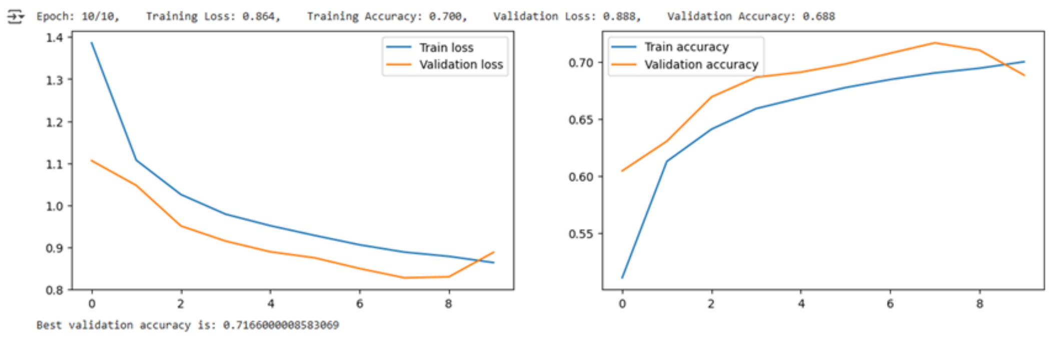
Question 1 report

Elliot Jones

April 2025

1 Introduction

In this project our goal was to build a compact and robust convolutional neural network CNN for the CIFAR-10 image classification task and optimise its performance. We first began with implementing the code we produced within the final lab and ran this code as a reference for the CNN's base performance and used this as a comparative performance for our testing. Our initial CNN had 2 convolutional layers that fully flattened into a single linear layer, with around 23,000 learnable parameters. The CNN was trained with 50 batches, a learning rate of 0.002, Adam optimiser with no weight and normalised data. Using the whole dataset from CIFAR10, We found that after 10 epochs the CNN began to level off with a training accuracy and validation accuracy of 0.700 and 0.688, respectively, and a training loss and validation loss of 0.864 and 0.888 respectively. We can see from Figure 1 that the training and validation losses both fall smoothly and the gap between them remains relatively small, showing no major sign of overfitting.



```
Epoch: 10/10,    Training Loss: 0.864,    Training Accuracy: 0.700,    Validation Loss: 0.888,    Validation Accuracy: 0.688
```

```
Best validation accuracy is: 0.7166000008583069
```

Figure 1: Base Convoluted Neural Network model

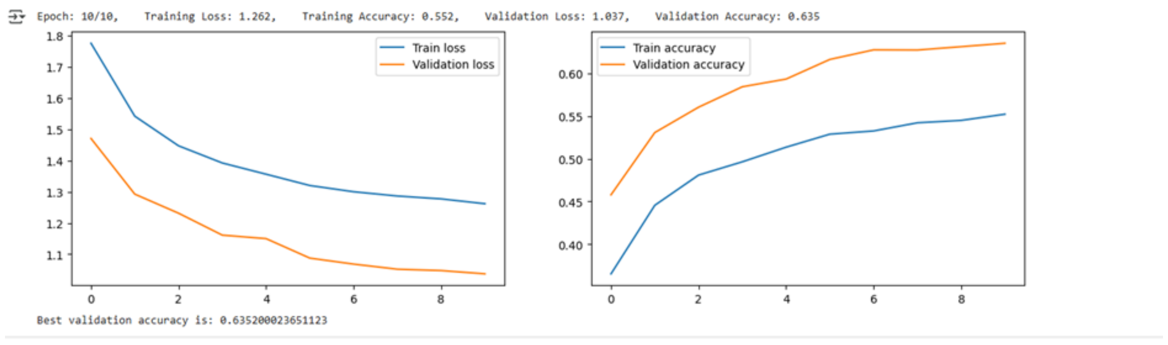
2 Development of CNN model

Before we started to improve our model, we noticed that there was a significant bottleneck within our code as our model was running off our Central Processing Unit (CPU) and not utilising the extensive number of cores from our Graphics Processing Unit (GPU) to speed up computations. We decided to change this within the code, ensuring that our model was utilising these resources so that our code ran much faster. Using the line of code below allowed our model to utilise our GPU when there was one available and the CPU otherwise.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

To improve our CNN we first started off with trying different architectures including, ReLU, ELU and Leaky ReLU, we found that using ELU within our model made it run a lot slower and yielded much worse validation and training accuracy. However, When using ReLU and Leaky ReLU we found significant improvement and that it was slight more beneficial to use Leaky ReLU over ReLU as we gained around an extra 1-2 % accuracy. This could be because Leaky ReLU gives a small non-zero slope in the negative region rather than zeroing out all negative inputs like normal ReLU. Because these negative activations still pass through as some gradient, more information can flow backward through the network thus yielding slightly better results.

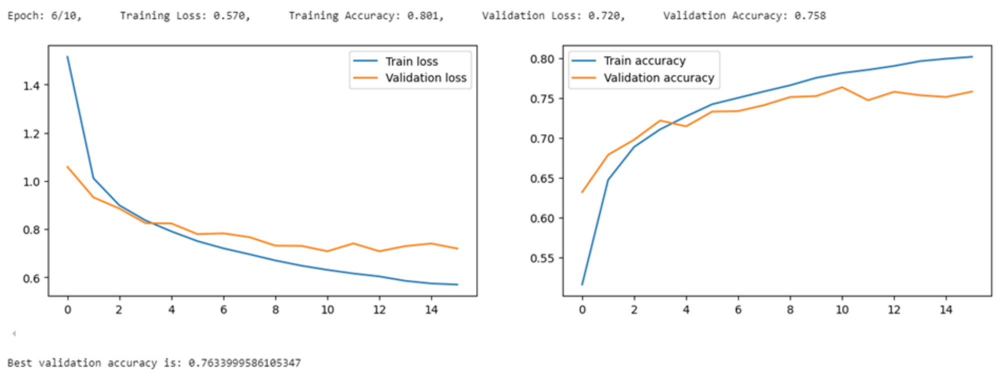
Next we decided to implement some different preprocessing techniques such as cropping, flipping, and rotating, to make our training more stable. We found that when using these augmentations on our CNN, the validation and loss curves decreased in a more linear fashion as shown in Figure 2, but yielded much lower validation and training accuracy of around 0.635 and 0.552 respectively, showing major signs of underfitting. This was most likely due to us making the training set a lot harder by implementing all of these techniques at once causing the capacity of our CNN to be a bottleneck in our model. Implementing these pre-processing models led our neural network to underfit massively as our two-layer network with small channel size was simply not complex enough to handle this level of augmented of data, causing the accuracy to be affected drastically.



```
Epoch: 10/10,    Training Loss: 1.262,    Training Accuracy: 0.552,    Validation Loss: 1.037,    Validation Accuracy: 0.635
Best validation accuracy is: 0.635200023651123
```

Figure 2: Base Convolutional Neural Network With Data Augmentation

From this, we decided to test different layers and channel sizes within our neural network, starting with increasing the number of channels in our layers from 32 to 64. To further our model, we then decided to add multiple layers within our CNN and tested our CNN multiple times with different channel sizes and different layers. This led us to a final model with five convolutional layers with the channel widths being 3-64-128-256-512-1024 along with batchnorm, maxpool and dropout on each followed by a final global average pool. This complexity in our design allowed us to handle the heavily augmented CIFAR-10 inputs while repeated pooling and dropout helped to prevent our model from overfitting. This led to instant results with our training behaviour becoming more stable and converging much faster as shown in Figure 3. From this we achieved a training accuracy of 0.0801 and a validation accuracy of 0.758 a significant increase from our original CNN.



```
Epoch: 6/10,    Training Loss: 0.570,    Training Accuracy: 0.801,    Validation Loss: 0.720,    Validation Accuracy: 0.758
Best validation accuracy is: 0.7633999586105347
```

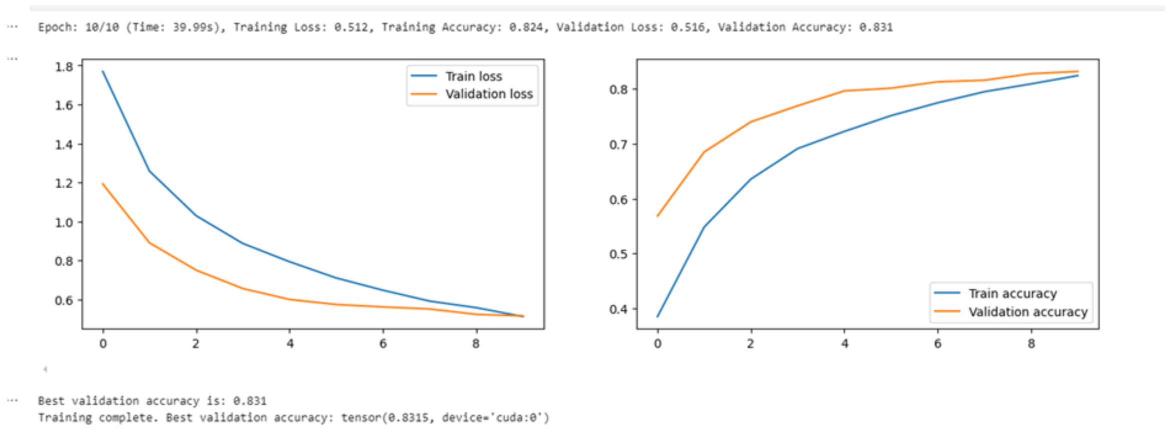
Figure 3: Augmented CNN With Architecture Change

After we had made our Convoluted Neural Network deeper we decided to perform a grid search with k-fold cross validation to choose a more optimised learning rate and potentially introduce weight decay into our Adam optimiser. Using 8 epochs and 3 folds to train, we decided to do a parameter search over multiple learning rates and weight decays as shown below.

```
 grid = {
    "lr":            [0.001, 0.002, 0.0015, 0.003],
    "weight_decay": [0, 0.001, 0.002, 0.003, 0.005, 0.01]
}
```

This K-fold cross-validation took extremely long to run, highlighting a computational bottleneck within our code due to the number of combinations of learning rates and weight decays we had to check. The best learning rate and weight decay were 0.001 and 0 respectively, and after plugging these into our CNN model we found that we had achieved a training accuracy of 0.824 and validation accuracy of 0.831 after 10 epochs.

With this learning rate, we saw that the training and validation accuracy converged very smoothly with no severe variations and that it had a fairly small training and validation gap throughout the training, suggesting no over-fitting as shown below in Figure 4.



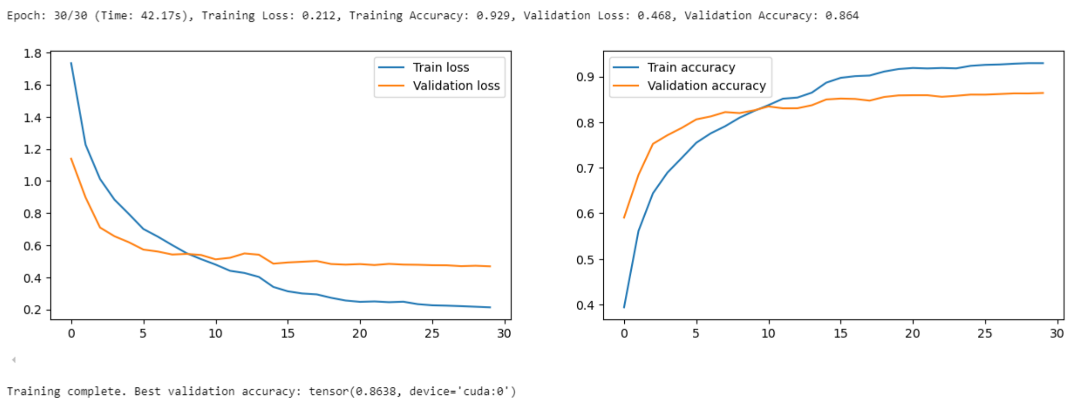
```
Epoch: 10/10 (Time: 39.99s), Training Loss: 0.512, Training Accuracy: 0.824, Validation Loss: 0.516, Validation Accuracy: 0.831
```

```
Best validation accuracy is: 0.831
Training complete. Best validation accuracy: tensor(0.8315, device='cuda:0')
```

Figure 4: CNN With Learning Rate And Weight Decay Optimised With K-Fold

3 Results From Our Final CNN Model

Having developed and improved our model to a good standard, we decided to implement our CNN with 30 epochs to see what our final model would look like and implemented a performance-based learning rate schedule to try to make progress after the plateau we experienced past epoch 10. From Figure 5, we can see below that after around 20 epochs our validation and training accuracy plateaued, and we were able to squeeze out 1-2% more using the Learning rate reduction, with the best values being 0.929 for training accuracy and 0.864 for validation accuracy. We also noticed that around epoch 30 there was a sizeable gap with our training accuracy and validation accuracy, suggesting that there was some weak overfitting. This generalisation bottleneck could potentially be due to our regularisation of our data not being strong enough for our model anymore. From this, we concluded that utilising data pre-processing techniques, different architectures and layers, and the Adam architecture that had the learning rate and weight decay optimised with K-fold cross validation allowed us to improve our CNN significantly from the initial model.



```
Epoch: 30/30 (Time: 42.17s), Training Loss: 0.212, Training Accuracy: 0.929, Validation Loss: 0.468, Validation Accuracy: 0.864
```

```
Training complete. Best validation accuracy: tensor(0.8638, device='cuda:0')
```

Figure 5: Final CNN With Learning Rate Reduction and 30 Epochs

4 Potential Future Improvements To Our CNN

Although a validation accuracy of 0.864 is considered a fairly good Neural network model there are a few more things that we could consider to implement in the future given the opportunity. We could first try to reduce the model's slight overfit in the later epochs by potentially increasing the dropout or implementing harsher pre-processing methods as this might allow our model to gain a small increase. During the development of our model, we implemented code to track the amount of time it took to complete each epoch and found that as we made our CNN more complex our run time increased drastically from around 20 seconds per epoch to 41 seconds, indicating a significant computational bottleneck. In the future, we could potentially spend more time optimising our code and CNN to decrease the run times by spending more time studying the effect of the size of batches and channels on computational time. To increase the performance of our CNN drastically we would look at using a pretrained backbone, training on a model that has already been trained on CIFAR10 data and then using our model to fine-tune the network so that it adapts its pretrained features to our task. This would allow our CNN to benefit massively by transferring the majority of the training and achieving faster convergence, allowing us to speed up our computation time and improve our CNN's performance at the same time. We could also expand on our architecture rather than using plain 3x3 convolutions, we could utilise ResNet bottleneck blocks, where each block uses a 1x1 $\mapsto$ 3x3 $\mapsto$ 1x1 pattern to reduce then restore channel dimensionality, drastically cutting the computation while preserving performance.