CS4287: Neural Computing

Assignment 1: Convolutional Neural Network (CNN)

Cormac O Connor (20227426), Sophia Keady (20231555), Leon Cullen (20274815)

1. DATA SET CIFAR-10

Introduction to CIFAR-10

The CIFAR-10 dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for benchmarking image classification algorithms.

Dataset Composition

CIFAR-10 consists of 60,000 32x32 color images in 10 different classes. The classes include airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images per class with 50,000 training images and 10,000 test images.

The Data Set:

a. Visualization of Some of the Key Attributes:

In our submitted Jupyter Notebook, we visualize the dataset using Matplotlib. This visualization helps in understanding the variety and characteristics of the images in the dataset. The code snippet below displays the first nine images from the training set, each belonging to different classes. This visual representation is good to get an understanding of what the data is visually which then grants insights into the data, such as the diversity of colors, the presence of noise, and the variability in object positioning and scaling within the images

```
# Plot first few images
for i in range(9):
    plt.subplot(330 + 1 + i)
    plt.imshow(train_images[i])
plt.show()
```

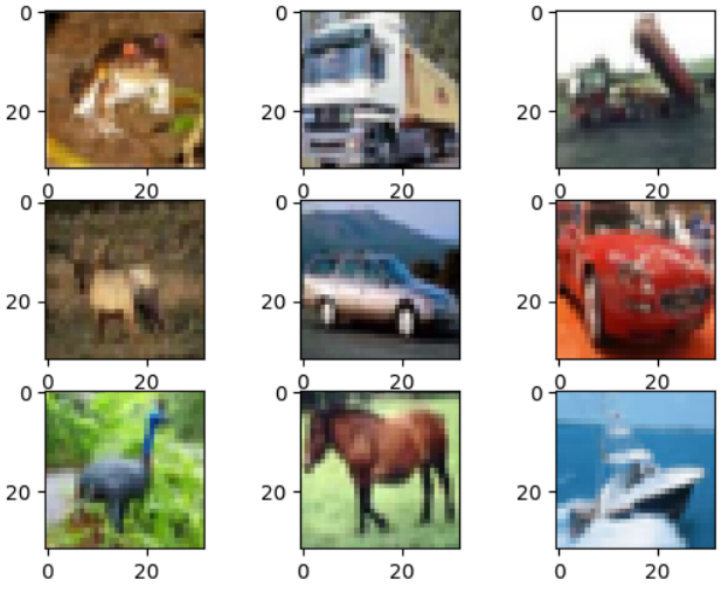

Figure 1.1: Examples of the images in the dataset

The below pairplot was made to visualize the distribution of the data which has been transformed from thousands of pixels to just 2 principal components PC1 and PC2. As normally pair plots wouldn't be feasible with high-dimensional data.

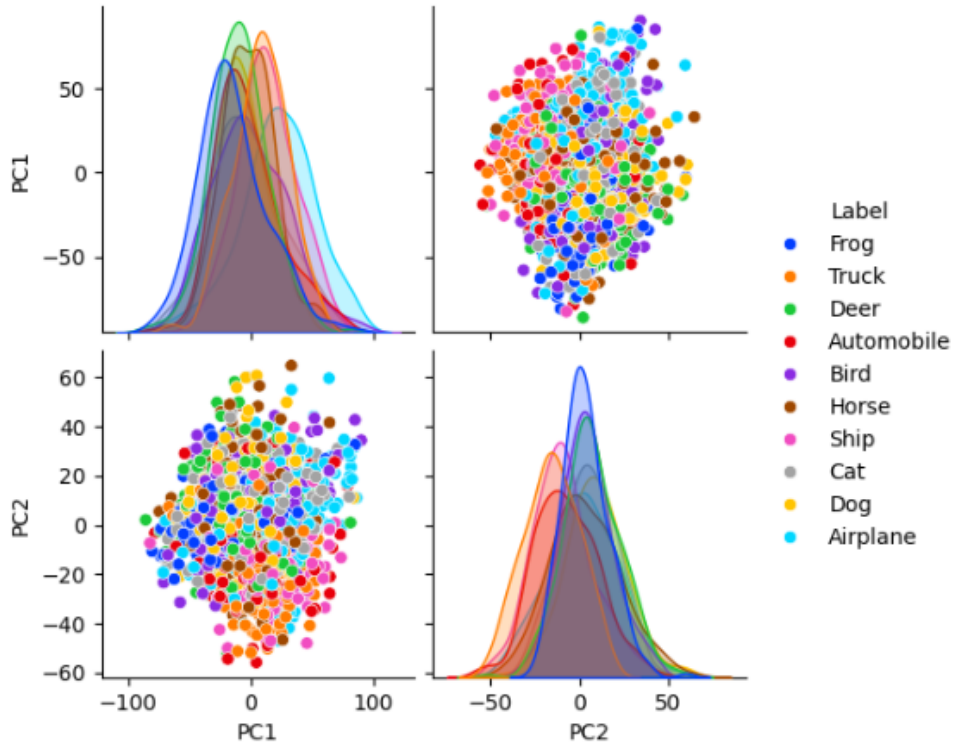

Figure 1.2: Visualizations of the dataset

b. Data Correlation and Feature Engineering

Data Correlation:

With CIFAR-10, we decided to focus on the image as a whole rather than dissecting correlations between pixels or colours, because our dataset is a collection of coloured images it doesn't follow traditional correlation analysis that you can use with tabular datasets. The importance for us is in the spatial relationship and patterns within the images. Our CNN has been designed to automatically learn and understand these patterns which represent different objects in images.

Feature Engineering:

Unlike other ML approaches where feature engineering is critical, CNNs are very capable at automatically extracting and learning the relevant features from raw images. This is one of the most important aspects of CNNs as it allows them to excel in image recognition tasks. The CNNs layers progressively learn more complex patterns the deeper the layers go, that translates to a significant advantage in complex image datasets like ours. While some data manipulation such as reshaping was required to get the CNN to work it's not as extreme.

c. Pre-processing Applied to the Data

Preparing the data for our model was a key factor in prioritizing its effectiveness. Here are the preprocessing techniques we decided on and applied to our dataset.

Normalization:

The first and most important step in our preprocessing was normalizing the images data. We scaled down our pixel values in the images by dividing each value by 255 and we went from ranges of 0-255 to 0-1. This normalization is key because it helps to bring consistency to the input data and also helps in speeding up the training process by making sure that the neural network's updates are not incredibly large and unstable.

One-Hot Encoding of Labels:

The next step in our preprocessing was transforming the dataset labels for compatibility with our model. First each image label in our dataset is represented as a single integer that denotes its class. We decided to convert these integers into the one-hot encoding format. This means for each label we created a 10 element array (reflecting our 10 classes), where every element is set to 0 except for the one that corresponds to the class of the image which is set to 1. For example if one of our images is tagged with '3' its label would transform to this array [0,0,0,1,0,0,0,0,0,0]. This allows our model to align with the output it predicts and work more smoothly with our categorical cross-entropy loss function.

2. The Network Structure and Hyperparameters

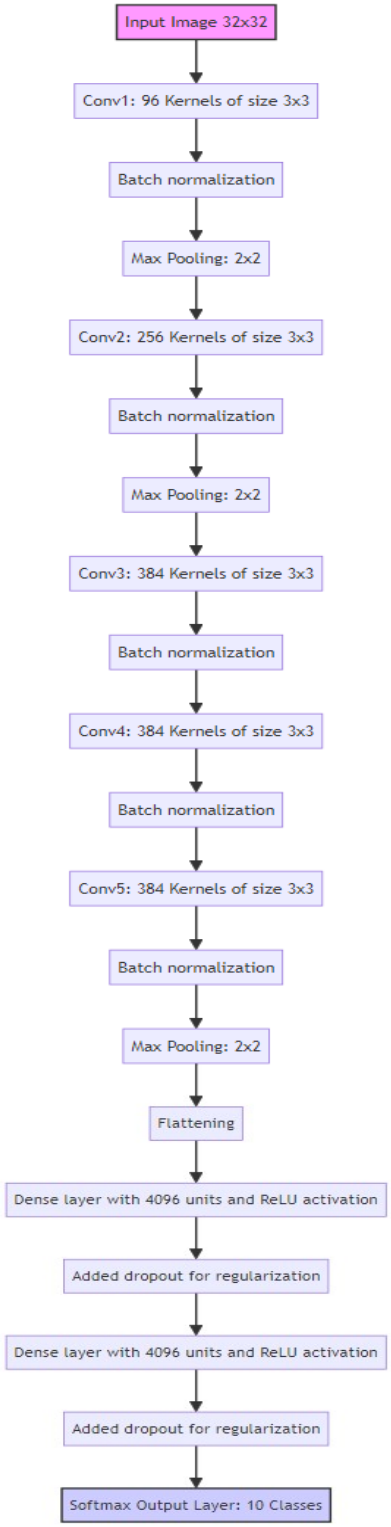

Figure 2.1: Diagram of Network Architecture

Weight Initialization:

Weight initialization played an important role in the training process and overall network performance. The TensorFlow/Keras framework, by default, uses Glorot uniform initializer (also known as Xavier uniform initializer) a method that sets the initial random weights of the model in a way that's just right for the layers with ReLU activation. This initializer is designed to keep the scale of the gradients consistent in all layers. By doing this across the layers, it helps in stabilizing the training process in deep neural networks.

Activation Function:

We chose the Rectified Linear Unit(ReLU) as our activation function because, for each neuron in our network we needed a way to decide whether or not it should be activated. This function only allows positive values to pass through and turns negative values to zero. This helps our network to learn more efficiently, train faster and steer clear of any potential issues like the vanishing gradient problem which is common in neural networks (meaning when the model stops learning as it should).

Batch Normalization:

To ensure that our model doesn't get too swayed by extreme values, we used batch normalization after each layer. This makes sure no neuron is unbalanced and it also normalizes the output of the previous layer adding more balance and in turn speeding up the learning process.

Regularization:

To prevent overfitting, our model includes dropout layers with a dropout rate of 0.5 after the fully connected layers. Dropout is a form of regularization that randomly sets a fraction of the input units to zero during training, this can help in making the model less sensitive to the specific weights of neurons. This is crucial to prevent overfitting and make sure that it performs well on new and unseen data also.

Hyperparameters:

These are the settings we tweaked before training began.

Optimizer: We chose Adam for its effectiveness in large datasets and parameters.

Loss Function: Categorical Crossentropy is employed as the loss function, which is appropriate for multi-class classification problems.

Epochs: The model is trained for 19 epochs, a value determined based on initial experiments to balance between sufficient training and computational efficiency.

Batch Size: The default batch size in Keras (32) is used, which is a common choice for a balance between memory usage and performance.

Transfer Learning:

We decided not to take advantage of transfer learning and opted to train our model from scratch instead which works better with our dataset.

3. The Cost/ Loss / Error / Objective function

The cost/loss/error/objective function quantifies the difference between the predicted outputs of our model and the actual targets and helps manage and adjust the weights accordingly.

We decided on the following during the model development:

Categorical cross entropy:

We have decided on using ‘categorical_crossentropy’ as our loss function, our reasoning for this is because it suits the CIFAR-10 dataset, where each image belongs to one and only one of our ten categories and as CIFAR-10 is a multi classification dataset which the ‘categorical_crossentropy’ is the appropriate loss function .

Categorical cross entropy works by evaluating how well our model's predictions align with the labels. It compares the expected output versus what the model predicts for each image. Categorical cross-entropy is an effective loss function for this scenario because it pairs well with one-hot encoding, a common method for representing categorical labels in neural networks. One-hot encoding transforms categorical labels into a format that is more suitable for classification tasks.

If the model correctly predicts the class of an image, the loss will be low. If the model incorrectly predicts the class of an image, the loss will spike, this means that there is still more learning to be done. Keeping in mind that ‘categorical_crossentropy’is rewarded from correct predictions and punished for incorrect predictions, that is another reason why our function is also suited for our dataset because our model can learn more efficiently and effectively from this process.

4. The Optimiser

Adam(Adaptive Moment Estimation):
In our implementation of a Convolutional Neural Network, using the AlexNet architecture for the CIFAR-10 dataset, the Adam optimizer plays a crucial role. Adam, which is short for Adaptive Moment Estimation, is an extension of stochastic gradient descent that has been proven to be reliable in both efficiency and effectiveness.

It stands out for its use of adaptive learning rates and momentum. When training CNNs like AlexNet, which consist of many complex layers and a substantial number of parameters, Adam helps in efficiently managing the complex weight space. For CIFAR-10, a dataset with varied and small images, Adam works well in the faster and more effective convergence of our network compared to traditional gradient descent methods. This results in a quicker and more accurate training process, which is important for handling the intricate patterns and features present in the CIFAR-10 image data.

5. Cross Fold Validation

Cross Fold Validation:
Our implementation of k-fold cross-validation is important for assessing the performance of our AlexNet model on the CIFAR-10 dataset. We decided on using StratifiedKFold from scikit-learn; we also ensure that each fold of the validation process is accurate compared to the overall dataset, maintaining the same proportion of each class as in the full dataset. This method helps the reliability of our model's performance metrics, as it helps mitigate the risks of overfitting and ensures that the model's effectiveness is not overly dependent on a subset of the data.

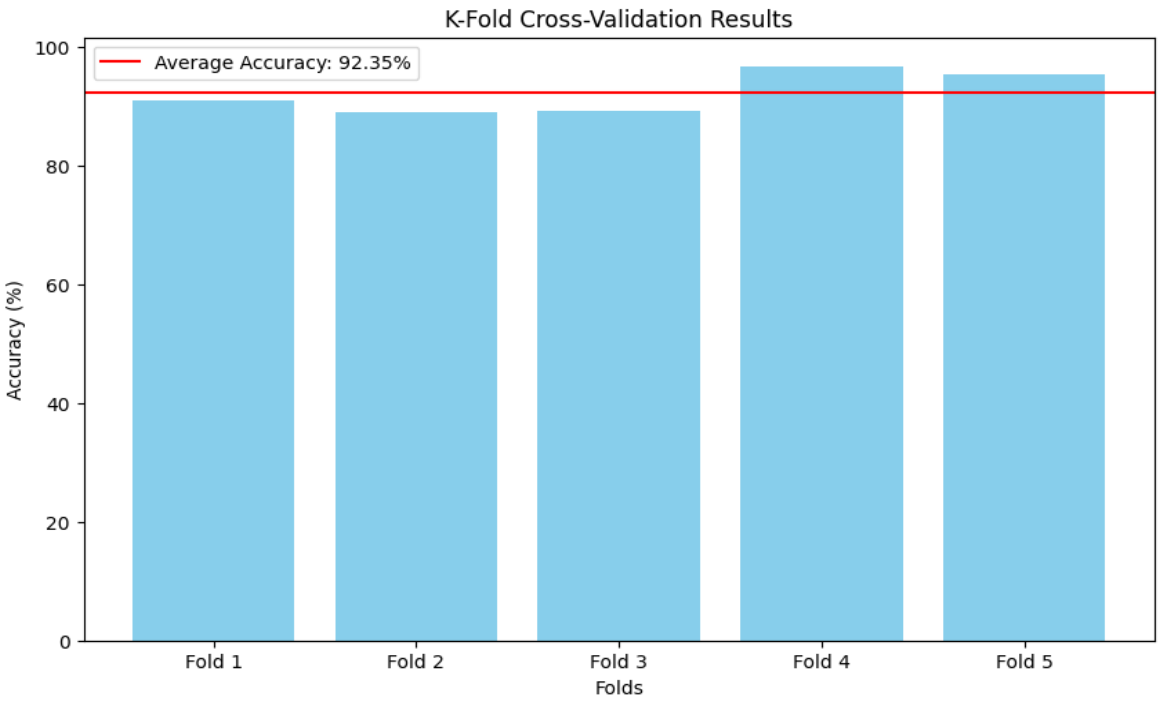

Figure 5.1: Visualizations for the K-Fold Cross-Validation

This is important as overfitting in multi-classifications neural networks are common. We opted for the approach of re-loading the pre-trained model for each fold and evaluating it on a different subset of data; this is the standard practice in cross-validation, this works as a way of testing our models performance against data it hasn't fully adapted too. This process provides an in-depth understanding of the model's generalization capabilities.

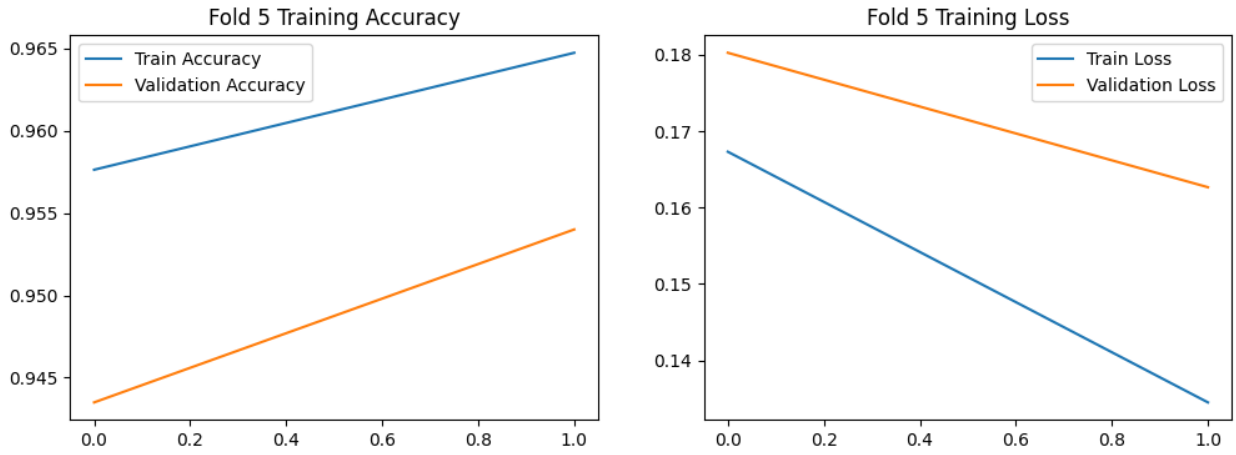

Figure 5.2: Accuracy and loss graphs for Fold 5 of the cross-validation

The success of our model in the cross-fold validation process is highly significant. It indicates that the model is not overfitted, which is a common challenge as mentioned. Overfitting occurs

when a model learns the details and noise in the training data to an extent that it negatively impacts the performance of the model on new data as a result losing the ability to generalize data as the parameters have become too focused on the original training data .

By performing well across different folds, each representing unique subsets of data, our model demonstrates its ability to generalize well. Generalization is the ability of a model to adapt to new, previously unseen data, drawn from the same distribution as the one used to create the model. This is critical when applying the model, as it suggests that the model will perform reliably when exposed to real-world data, maintaining accuracy outside the controlled environment of the training dataset. This is further demonstrated by our results below:

```
Epoch 1/2
1250/1250 [==============================] - 27s 21ms/step - loss: 0.1664 - accuracy: 0.9579 -
val_loss: 0.2288 - val_accuracy: 0.9359
Epoch 2/2
1250/1250 [==============================] - 26s 21ms/step - loss: 0.1398 - accuracy: 0.9641 -
val_loss: 0.2925 - val_accuracy: 0.9102
Score for fold 1: 91.01999998092651%
Epoch 1/2
1250/1250 [==============================] - 26s 20ms/step - loss: 0.1603 - accuracy: 0.9596 -
val_loss: 0.1537 - val_accuracy: 0.9534
Epoch 2/2
1250/1250 [==============================] - 21s 17ms/step - loss: 0.1452 - accuracy: 0.9634 -
val_loss: 0.3616 - val_accuracy: 0.8918
Score for fold 2: 89.17999863624573%
Epoch 1/2
1250/1250 [==============================] - 21s 17ms/step - loss: 0.1637 - accuracy: 0.9585 -
val_loss: 0.2237 - val_accuracy: 0.9428
Epoch 2/2
1250/1250 [==============================] - 21s 16ms/step - loss: 0.1399 - accuracy: 0.9643 -
val_loss: 0.4040 - val_accuracy: 0.8930
Score for fold 3: 89.30000066757202%
Epoch 1/2
1250/1250 [==============================] - 21s 17ms/step - loss: 0.1645 - accuracy: 0.9583 -
val_loss: 0.2166 - val_accuracy: 0.9383
Epoch 2/2
1250/1250 [==============================] - 21s 17ms/step - loss: 0.1471 - accuracy: 0.9627 -
val_loss: 0.1107 - val_accuracy: 0.9684
Score for fold 4: 96.84000015258789%
Epoch 1/2
1250/1250 [==============================] - 21s 17ms/step - loss: 0.1673 - accuracy: 0.9576 -
val_loss: 0.1802 - val_accuracy: 0.9435
Epoch 2/2
1250/1250 [==============================] - 20s 16ms/step - loss: 0.1345 - accuracy: 0.9647 -
val_loss: 0.1626 - val_accuracy: 0.9540
Score for fold 5: 95.39999961853027%
```

Average score across all folds: 92.34799981117249%

We did experiment with the cross-fold validation parameters such as epochs which resulted in consistent accuracy of over 90% regardless of the amount of iterations.

6. Results

Results:

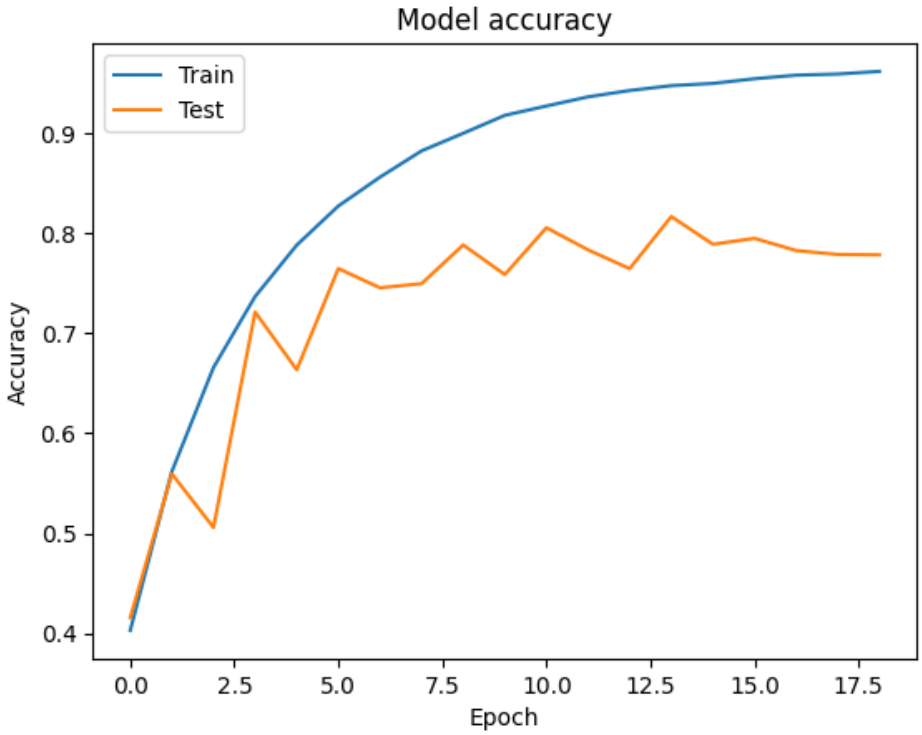

Figure 6.1: Accuracy chart for the model

Our model over a total of 19 achieved an accuracy of 96% on the training data with a val-accuracy of 80%.

Epoch 16/19
1563/1563 [==============================] - 28s 18ms/step - loss: 0.1521 - accuracy: 0.9546 - val_loss: 1.2009 - val_accuracy: 0.7775
Epoch 17/19
1563/1563 [==============================] - 28s 18ms/step - loss: 0.1556 - accuracy: 0.9551 - val_loss: 1.0056 - val_accuracy: 0.7903
Epoch 18/19

1563/1563 [==============================] - 28s 18ms/step - loss: 0.1498 - accuracy: 0.9561 - val_loss: 1.1297 - val_accuracy: 0.8014
Epoch 19/19
1563/1563 [==============================] - 28s 18ms/step - loss: 0.1442 - accuracy: 0.9589 - val_loss: 0.8944 - val_accuracy: 0.7926

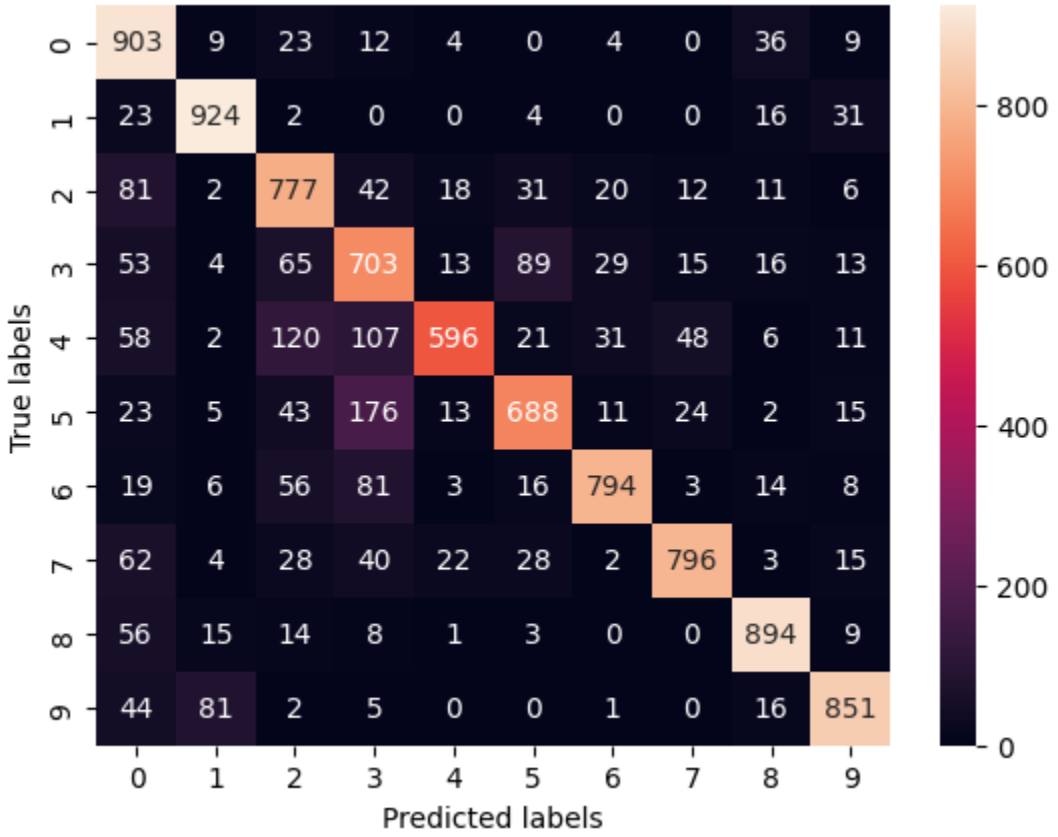

Figure 6.2: Confusion Matrix for the model

As previously mentioned our cross-fold validation worked well and we achieved an accuracy rate of 96%.

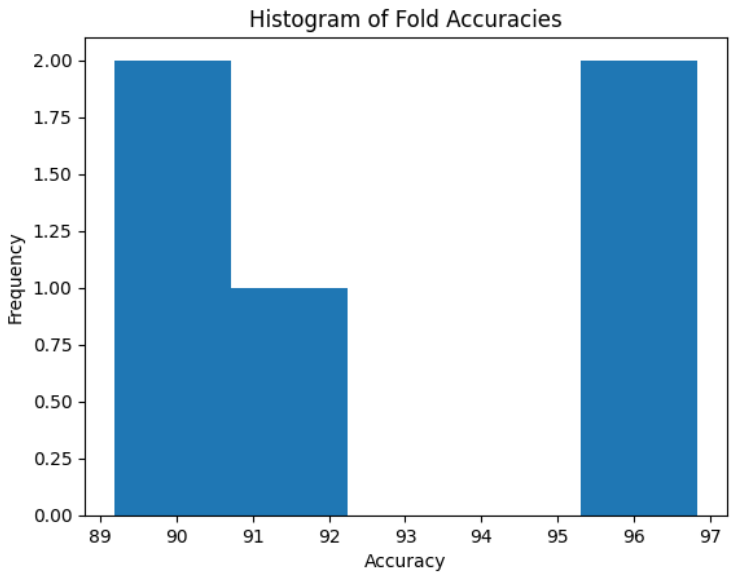

Figure 6.3: Histogram of fold accuracies

This means our CNN correctly learned the Cifar10 dataset and could correctly distinguish and reliably predict correctly what was in the image.

7. Evaluation of Results

The model performed exceptionally well on the Cifar10 dataset, managing to achieve an accuracy rate of over 95% with an extremely low loss. This high accuracy indicates that it correctly classified the majority of the samples. Such results suggest that the model is well-trained and is likely to perform reliably on data outside of the testing or validation sets.

We also assessed our model for the potential issues of overfitting and underfitting. As shown in the cross fold section the testing loss did not continuously increase and the validation loss did not increase, it's evident that our model is not overfitted to the dataset. Furthermore, the model is not underfitted; if it were, both the training and validation losses would be higher.

The Cross Fold Validation results, as discussed earlier, further corroborate these findings. Showing and verifying that our model works as expected and is able to perform its task.

8. Impact of varying hyperparameters

Varying hyperparameters like the number of filters, kernel size, learning rate, batch size etc. The first variation we decided to make was based on the kernel size and what would happen if it was increased/decreased.
When trying to include the following as their own models in the notebook they would break the jupyter notebook by giving us resource error where our model wasn't able to run on the computational power we had available; the hyperparameters we changed only required changing variables already defined in our model.

Kernel Size:

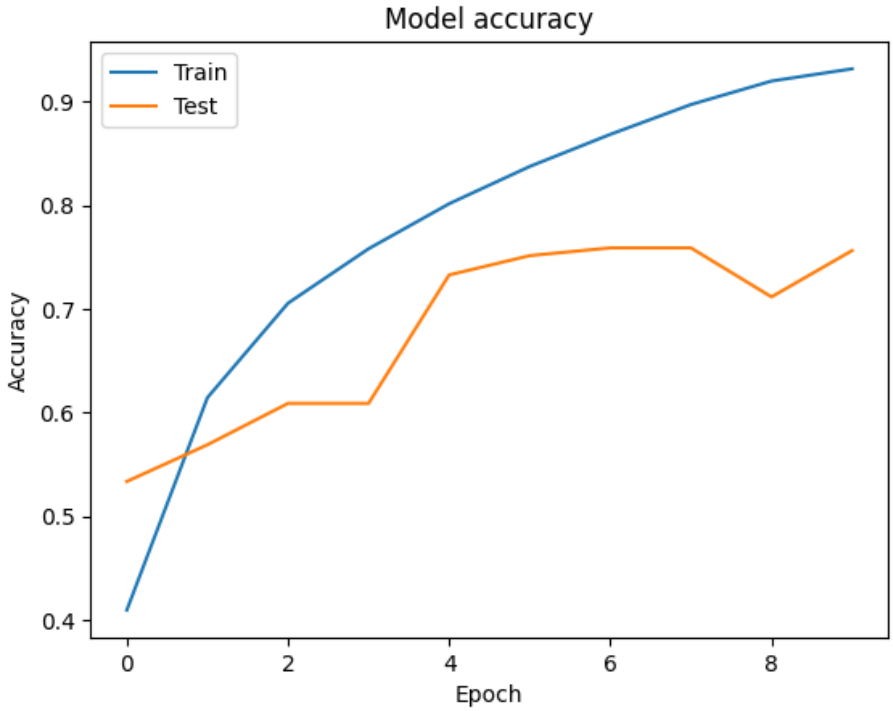

Figure 8.1: Accuracy graph showing kernel size 5x5

Kernel Size 5x5:
When increasing the kernel size some patterns are noticed while the initial learning rate is quite high it peaks quicker and struggles to pick out smaller details and maintains a 77% accuracy and doesn't improve quickly. It's also far more computationally intense compared to 3x3 taking an additional 8 seconds to go through an epoch.

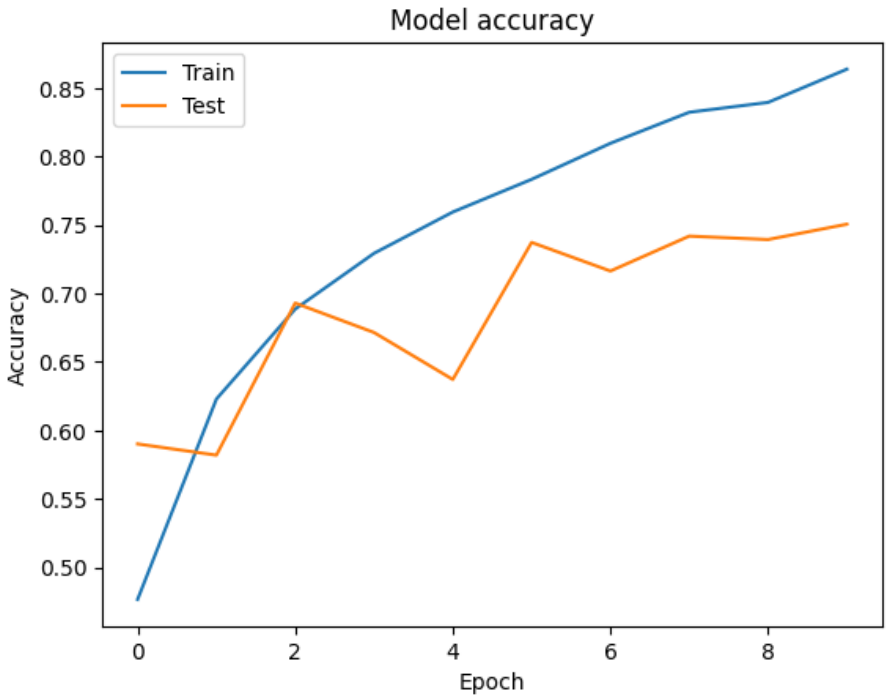

Figure 8.2: Accuracy graph for kernel size 2x2

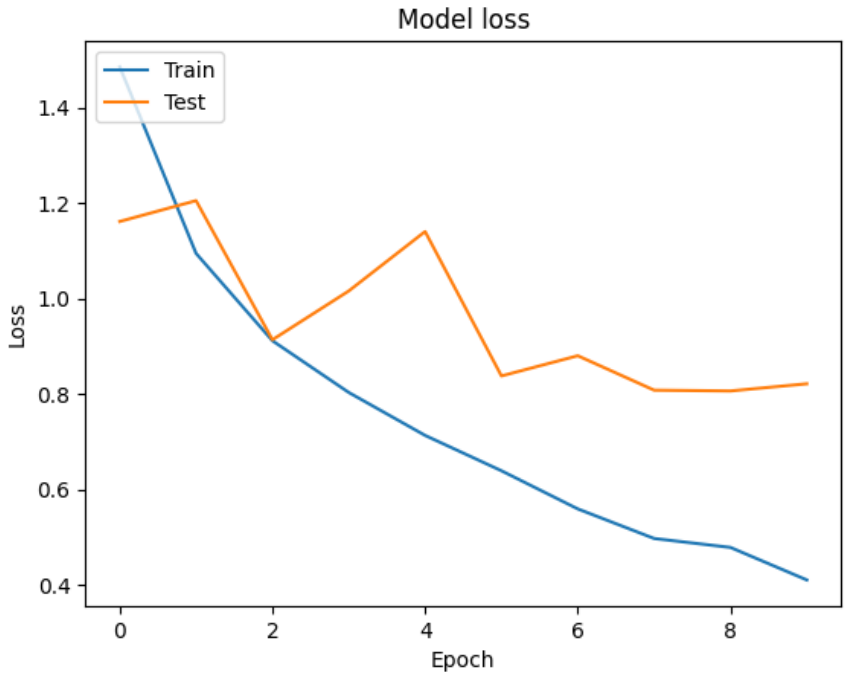

Figure 8.3: Loss graph for kernel size 2x2

Kernel Size 2x2:

Next we tried 2x2 the 2x2 kernel size went way quicker running taking 17 seconds to complete an iteration. While in the training data it appears to be performing well the accuracy of the test data

is worrying it seems to be rather inconsistent and not performing as effectively compared to the other kernel sizes this suggests that smaller kernel sizes may not be as good at generalization. It's also worth noting that the model had very high loss

Adjusting the learning rate:

Next we decided to increase the learning rate. The default rate is 0.001 here's what happened when we increased it to 0.1.

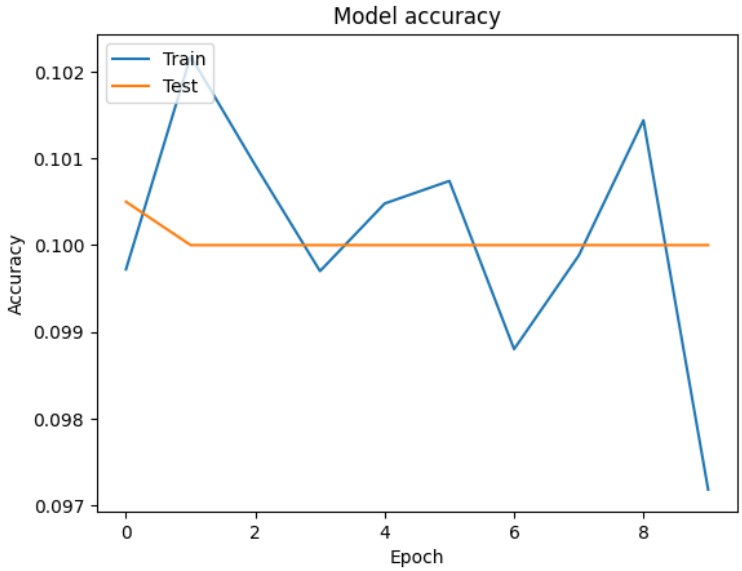

Figure 8.4: Accuracy graph for Learning Rate = 0.1

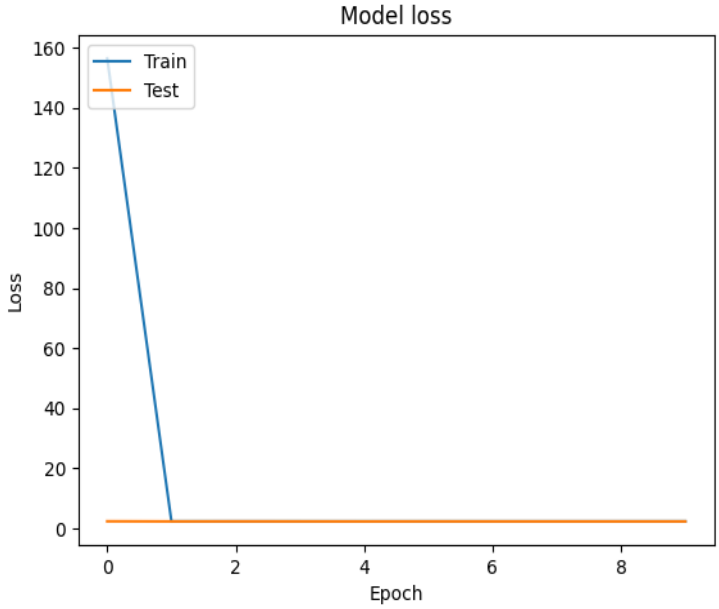

Figure 8.5: Loss graph for Learning Rate = 0.1

As seen in the graphs above, increasing the learning rate too much causes a complete fall off of the model's ability to learn. There are three main reasons for this Overshooting Minima, failing to converge and gradient noise.

Overshooting Minima:
A high learning rate can cause the optimizer to take very large steps in the parameter space. This can lead to overshooting the optimal values (minima) of the loss function.

Failing to converge:
With a high learning rate, the model might fail to converge to a stable solution. The updates to weights are too large to settle down into deeper, narrower parts of the loss landscape where the optimal solutions often lie.

Gradient Noise:
At a high learning rate, the gradient updates can be very noisy. This noise can dominate the signal, making it hard for the optimizer to find the direction of the true gradient. This is the main cause of some of the erratic iterations.

We settled on 0.1 for our learning rate as it performed consistently well.

Epochs:

While harder to vary for investigation due to the computational intensity of AlexNet processing images. Epochs during testing that were over 20 tended to overfit the data and attempts to implement techniques to bring down overfitting seem to get less effective as the amount of iterations gets higher.

Dropout Rate:
Dropout rate as a concept is fairly simple: neurons are randomly selected and temporarily removed “Dropped out”. This means they stop contributing to downstream neurons. Our model uses a rather high dropout. The reasoning for this was because we found out when implementing dropout into our model running at higher epochs with a high dropout tended to produce a consistent result and made the model quite good at generalization which is a key goal when training a CNN on categorizing images as a CNN that cannot recognise anything outside the training data. Higher dropout rates resulted in underfitting where learning ability was impacted.