MLP & CNN CIFAR-10 IMAGE CLASSIFIER

Submitted by

Tan RongDe Darius (A0146654Y)

Neural Networks Project

EE4305: Introduction to Fuzzy / Neural Systems

AY 2018/2019 Semester 1

Table of Contents

1. Literature Survey

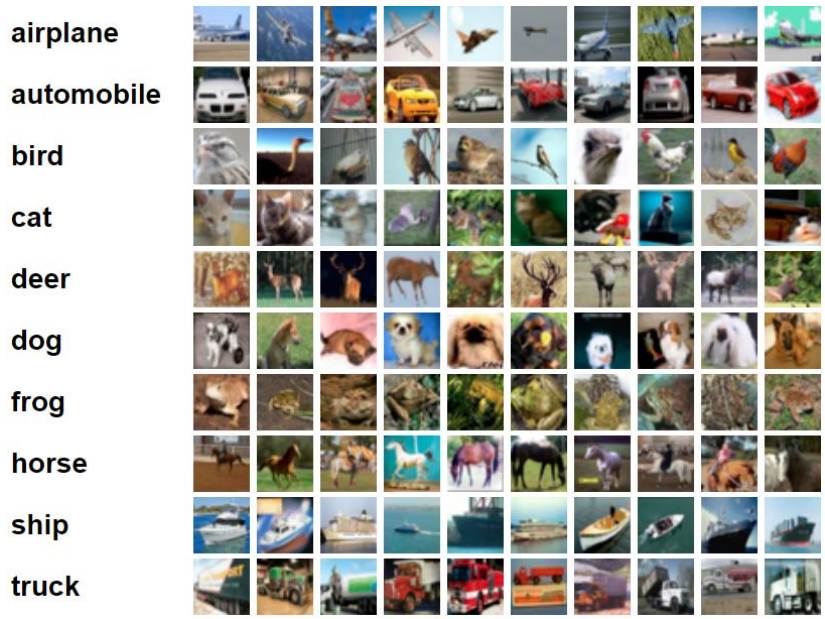


Figure 1. CIFAR-10 Dataset Classes

CIFAR-10 is a set of 60,000 32 x 32 colour images with 10 mutually exclusive object classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. This dataset is a labelled subset of the 80 million tiny images dataset collected by Krizhevsky [1]. In this project, I will be attempting to implement a CIFAR-10 Image Classifier using the traditional Multi-Layer Perceptron (MLP) architecture as well as a Convolution Neural Network (CNN).

In the list of state-of-the-art classifiers compiled by Benenson in 2016 [2], he listed the best performing algorithms in classifying well known datasets including CIFAR-10. The list for CIFAR-10 consists of either methods in improving on existing CNN architectures through new or improved hyperparameters or novel CNN architectures in classifying the CIFAR-10 dataset. Methods to improve existing architectures includes new techniques regarding performing optimisation [3], regularisation [4], max pooling [5], or new activation functions [6]. Novel CNN architectures include designs such as GoogLeNet (2014) [7], VGGNet (2014) [8] and ResNet (2015) [9] that were the top performers in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [10] in 2015 and 2014. For fully connected "permutation invariant" networks without convolution, the Z-Lin network was developed and proved to be state-of-the-art for fully connected networks at approximately 70% classification accuracy on CIFAR-10 [11].

GoogLeNet or Inception was the winner of ILSVRC 2014 with a top-5 error rate of 6.67%, close to human level performance. It uses a 22 layer deep CNN with 4M parameters. VGGNet on the other hand was the runner up with 7.1% top-5 error rate, implemented using 16 convolutional layers with 138M parameters, hence taking the developers of the net 2-3 weeks to complete it training. The ResNet architecture won the ILSVRC

2015 on the ImageNet dataset with a 3.57% top-5 error rate and had an error rate of 6.43% for CIFAR-10. It is a Residual Neural Network consisting of 152 layers while having lower complexity than VGGNet.

However, these are older methods compiled in 2016, the latest advancements in classifying the CIFAR-10 dataset include methods such as automatic data augmentation by AutoAugment [12], regularisation with Shake-Shake [13], or new architectures such as SimpNet [14]. In these cases, the new methods were developed to address certain limitations in previous designs. For AutoAugment, it was developed to automate the iterative process of determining suitable data augmentation algorithms for the target dataset, ensuring that the selected data augmentation policy suits the dataset of interest [12]. Shake-Shake was developed as a new regularisation method for multi-branch networks to prevent overfitting on small datasets [13]. SimpNet on the other hand, was developed to provide an accurate but light-weight architecture to address the high computation and memory overheads of CNNs such as ResNet and VGGNet for application on real world embedded systems [14].

Such image recognition techniques pushes the boundaries of technology and they can be applied in fields such as social media, advertising or even national security. In social media, the applications of such techniques has been prominent for quite some time with Facebook's DeepFace allowing users to automatically be tagged in photos uploaded onto Facebook. For advertising, such technologies can be used for targeted advertisements such as to specific age groups or genders, which image recognition techniques can identify through training with datasets with those specific classes. In security, image recognition has also been increasingly prevalent such as in China, where the state has implemented national facial recognition systems to monitor and keep track of its citizens.

With the rate at which new image recognition technologies surface in the development of neural networks and machine learning, we are already in an age where machines are able to identify and recognise images with a higher accuracy than humans can. It is indeed an exciting time to enter into the field of artificial intelligence and thus, experimenting with MLP and CNN classifiers using Keras and TensorFlow such as in this project will prove to be beneficial in my technical knowledge and understanding of neural networks.

2. Multi-Layer Perceptron (MLP) Classifier

a. Sample MLP Classifier

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 512)               1573376
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               131328
_________________________________________________________________
activation_5 (Activation)    (None, 256)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                2570
_________________________________________________________________
activation_6 (Activation)    (None, 10)                0
=================================================================
Total params: 1,707,274
Trainable params: 1,707,274
Non-trainable params: 0
```

Figure 2. Number and types of layers in the example MLP classifier

```
Epoch 100/100
50000/50000 [==============================] - 3s 56us/step - loss: 1.0169 - acc: 0.6388 - val_loss: 1.2690 - val_acc: 0.5572
Training Set Evaluation:
Test loss: 1.269009328842163
Test acc: 0.5572
```

Figure 3. Test results after training MLP classifier

In the sample MLP classifier given for this assignment, the validation dataset was simply taken from the test dataset, which is not ideal. Instead, the validation set should have been from the training set and the test dataset should be reserved for the testing of the actual accuracy of the model against images that the classifier has not seen before.

The architecture used in this model consisted of 3 fully connected layers (dense) that includes an output connected layer of 10 neurons, which represents the 10 classes of CIFAR-10. These fully connected layers were followed by Rectified Linear Unit (ReLU) activation layers for the first 2 dense layers, where ReLU(x) = max(x,0) and a softmax activation layer to return the array of 10 probability scores for the output.

The dataset was split into mini-batch sizes of 128 for mini-batch gradient descent and the optimizer for this classifier is using the mini batch stochastic gradient descent optimiser, with categorical crossentropy loss as there are multiple classes for CIFAR-10.

After training the classifier over 100 epochs, the results are as seen in Figure 2, where the test accuracy and validation accuracy are identical at 0.5572, since they are the same dataset, and the final training accuracy to be 0.6388. From this result, it is clear that there is overfitting, as the model performs much

better on the training sets as compared to the test and validation sets as it is unable to generalise the results from the training data when implemented with the test data.

After the training of the sample classifier, I began to experiment with the different hyperparameters to see their impact on the final accuracy of the classifier. Further comparisons were executed with all else remaining equal to the sample classifier other than the parameter under test.

b. Effects of splitting training set into training and validation sets

The first change that I made was to change the dataset split from using the test data as the validation set to splitting the training set into training and validation sets. This would then split the dataset into 3 portions, training, validation and test sets. The training set is what the model see and learns from over the epochs, the validation set is what the model will see for us to validate its accuracy and to change hyperparameters accordingly and the test set is the final evaluation of the model when the model has been completely trained.

```
Epoch 100/100
35000/35000 [==============================] - 2s 64us/step - loss: 1.0510 - acc: 0.6298 - val_loss: 1.3688 - val_acc: 0.5243
Evaluation with Test Set:
Test loss: 1.3602779069900512
Test acc: 0.5185
```

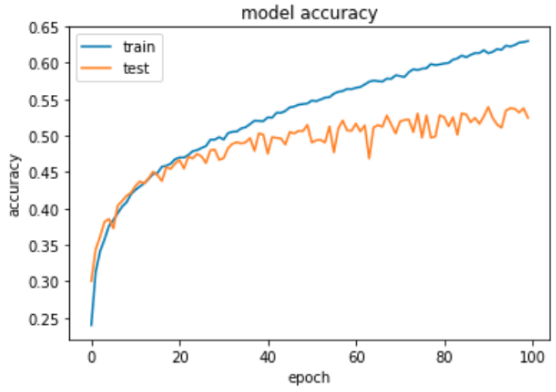


Figure 4. Test with 70-30 split of training and validation

Figure 3 shows the results where the training set was split into 70% for training, 30% for validation. In the raw number of images, the datasets consisted of 35,000 for training, 15,000 for validation and 10,000 for testing. When compared to the sample classifier, the training, validation and test accuracy all reduced as the model was trained on lesser number of images with all other hyperparameters remaining constant.

After that test in Figure 3, I increased the ratio between training and validation sets to a 80-20 split with 40,000 training images, 10,000 for validation and another 10,000 for testing. This was done to test if

allowing the model to train on a larger training set would improve its accuracy. As seen in Figure 4, this hypothesis holds true.

```
Epoch 100/100
40000/40000 [==============================] - 3s 63us/step - loss: 1.0363 - acc: 0.6329 - val_loss: 1.3474 - val_acc: 0.5290
Evaluation with Test Set:
Test loss: 1.3253343614578248
Test acc: 0.5365
```

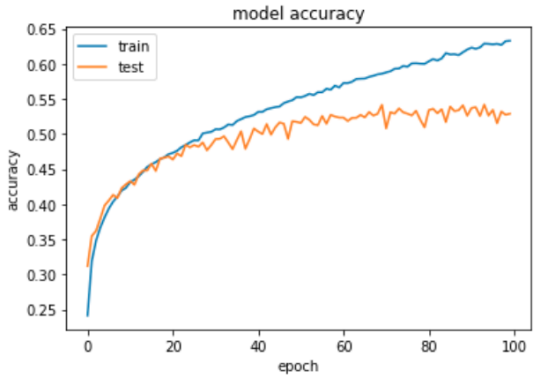


Figure 5. Test with 80-20 split of training and validation

Summary:

Dataset Split	Training Loss	Test Loss	Training Accuracy	Test Accuracy
Sample Classifier	1.0169	1.269	0.6388	0.5572
70-30	1.0510	1.36	0.6298	0.5185
80-20	1.0363	1.325	0.6329	0.5365

Table 1. Comparison of results for splitting dataset

c. Effects of varying number of hidden layers

To test the effect of the number of hidden layers, I first removed the fully connected hidden dense layer that was given in the sample classifier as seen in Figure 5. This caused the training, validation and test accuracies to fall as seen in Figure 6.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 512)               1573376
_________________________________________________________________
activation_7 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_8 (Activation)    (None, 10)                0
=================================================================
Total params: 1,578,506
Trainable params: 1,578,506
Non-trainable params: 0
```

Figure 6. Reducing number of hidden layers

```
Epoch 100/100
50000/50000 [==============================] - 3s 51us/step - loss: 1.0871 - acc: 0.6284 - val_loss: 1.3388 - val_acc: 0.5310
Training Set Evaluation:
Test loss: 1.3388218925476074
Test acc: 0.531
```

Figure 7. Test results from reducing number of hidden layers

When I increased the number of hidden layers by 1 layer by adding an additional dense fully connected layer with ReLU activation and dropout as seen in Figure 7, there was a slight, almost negligible increase in training, validation and test accuracy seen in Figure 8

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_16 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_17 (Dense)             (None, 256)               131328
_________________________________________________________________
activation_17 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_12 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_18 (Dense)             (None, 256)               65792
_________________________________________________________________
activation_18 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_13 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_19 (Dense)             (None, 10)                2570
_________________________________________________________________
activation_19 (Activation)   (None, 10)                0
=================================================================
Total params: 1,773,066
Trainable params: 1,773,066
Non-trainable params: 0
```

Figure 8. Additional dense layer with ReLU activation and dropout

```
Epoch 100/100
50000/50000 [==============================] - 3s 57us/step - loss: 0.9992 - acc: 0.6394 - val_loss: 1.2608 - val_acc: 0.5583
Training Set Evaluation:
Test loss: 1.260782372188568
Test acc: 0.5583
```

Figure 9. Results for test evaluation for additional dense layer

When I added an additional hidden layer on top of that in Figure 7, as seen in Figure 9, the validation and test accuracy after 100 epochs fell as compared to the original sample classifier.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_11 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_7 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_12 (Dense)             (None, 256)               131328
_________________________________________________________________
activation_12 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_8 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 256)               65792
_________________________________________________________________
activation_13 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_14 (Dense)             (None, 256)               65792
_________________________________________________________________
activation_14 (Activation)   (None, 256)               0
_________________________________________________________________
dropout_10 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                2570
_________________________________________________________________
activation_15 (Activation)   (None, 10)                0
=================================================================
Total params: 1,838,858
Trainable params: 1,838,858
Non-trainable params: 0
```

Figure 10. Model with 2 additional dense layers

```
Epoch 100/100
50000/50000 [==============================] - 3s 70us/step - loss: 1.0184 - acc: 0.6348 - val_loss: 1.2952 - val_acc: 0.5447
Training Set Evaluation:
Test loss: 1.2951862770080567
Test acc: 0.5447
```

Figure 11. Test results from 2 additional dense layers

Summary:

Number of hidden layers	Training Loss	Test Loss	Training Accuracy	Test Accuracy
0	1.0871	1.3388	0.6348	0.531
1 (Sample Classifier)	1.0169	1.269	0.6388	0.5572
2	0.9992	1.2607	0.6394	0.5583
3	1.0184	1.295	0.6348	0.5447

Table 2. Comparison of results for varying number of hidden layers

d. Effects of varying number of neurons in hidden layers

As seen in Figure 11, the number of neurons in the hidden dense layer was reduced from 256 to 128 as compared to the sample classifier. This caused the training, validation and test accuracy to reduce as seen in Figure 12.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_26 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_26 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_18 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_27 (Dense)             (None, 128)               65664
_________________________________________________________________
activation_27 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_19 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_28 (Dense)             (None, 10)                1290
_________________________________________________________________
activation_28 (Activation)   (None, 10)                0
=================================================================
Total params: 1,640,330
Trainable params: 1,640,330
Non-trainable params: 0
```

Figure 12. Model with reduced number of neurons in the hidden dense layer

```
Epoch 100/100
50000/50000 [==============================] - 3s 62us/step - loss: 1.0334 - acc: 0.6352 - val_loss: 1.2778 - val_acc: 0.5500
Training Set Evaluation:
Test loss: 1.2777681676864625
Test acc: 0.55
```

Figure 13. Test results for reduced number of neurons

When the number of neurons in the hidden dense layer was increased from 256 to 512 as seen in Figure 13, the training accuracy increased as compared to the sample classifier while the validation and test accuracy reduced. This could be due to overfitting as the classifier has higher training accuracy as compared to validation and test accuracy.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_23 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_16 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_24 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_24 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_17 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_25 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_25 (Activation)   (None, 10)                0
=================================================================
Total params: 1,841,162
Trainable params: 1,841,162
Non-trainable params: 0
```

Figure 14. Model with increased number of neurons in the hidden dense layer

```
Epoch 100/100
50000/50000 [==============================] - 3s 56us/step - loss: 0.9915 - acc: 0.6497 - val_loss: 1.2769 - val_acc: 0.5521
Training Set Evaluation:
Test loss: 1.276863482284546
Test acc: 0.5521
```

Figure 15. Test results for increased number of neurons

Another experiment that was done was reducing the number of neurons in the hidden dense layers while increasing the number of layers by adding another dense layer with ReLU activation and dropout as seen in Figure 15.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_33 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_33 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_23 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_34 (Dense)             (None, 128)               65664
_________________________________________________________________
activation_34 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_24 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_35 (Dense)             (None, 128)               16512
_________________________________________________________________
activation_35 (Activation)   (None, 128)               0
_________________________________________________________________
dropout_25 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_36 (Dense)             (None, 10)                1290
_________________________________________________________________
activation_36 (Activation)   (None, 10)                0
=================================================================
Total params: 1,656,842
Trainable params: 1,656,842
Non-trainable params: 0
```

Figure 16. Model with additional hidden layer with reduced number of neurons

```
Epoch 100/100
50000/50000 [==============================] - 3s 56us/step - loss: 1.0391 - acc: 0.6319 - val_loss: 1.2776 - val_acc: 0.5517
Training Set Evaluation:
Test loss: 1.2775952409744262
Test acc: 0.5517
```

Figure 17. Test results with additional hidden layer with reduced number of neurons

The results of the training can be seen in Figure 16, where the accuracy across training, validation and test all reduced as compared to the sample case.

Summary:

Number of neurons in hidden layer	**Training Loss**	**Test Loss**	**Training Accuracy**	**Test Accuracy**
128	0.9915	1.2768	0.6497	0.5521
256 (Sample Classifier)	1.0169	1.269	0.6388	0.5572
512	0.9915	1.2768	0.6497	0.5521
128 with additional layer	1.0391	1.2775	0.6319	0.5517

Table 3. Comparison of results for varying number neurons in hidden layers

e. Effects of different parameter update rules

Another hyperparameter that can be modified would be the optimiser used in the model [15], where for the sample classifier, the mini batch Stochastic Gradient Descent optimiser was used.

The first optimiser that was tried was the mini batch Stochastic Gradient Descent (SGD) with classical momentum, where momentum is a technique to accelerate the gradient descent across iterations [16].

```
Epoch 100/100
50000/50000 [==============================] - 3s 58us/step - loss: 0.8777 - acc: 0.6826 - val_loss: 1.2889 - val_acc: 0.5627
Training Set Evaluation:
Test loss: 1.2888936428070068
Test acc: 0.5627
```

Figure 18. Test results using the SGD with momentum optimiser

Figure 17 shows the results after 100 epochs using the SGD with momentum optimiser with 0.01 learning rate and 0.9 momentum. This resulted in higher training, validation and test accuracies as compared to the sample classifier. However, the training accuracy is larger than the test accuracy more so than in the sample classifier test results. This suggests a higher degree of overfitting in this case.

Another form of momentum would be Nesterov's Accelerated Gradient (NAG) [16]. In Keras, NAG is implemented as mini batch SGD with Nesterov momentum. Figure 18 shows the results after training with the mini batch SGD with Nesterov momentum optimiser with 0.01 learning rate and 0.9 momentum.

```
Epoch 100/100
50000/50000 [==============================] - 3s 62us/step - loss: 0.8014 - acc: 0.7108 - val_loss: 1.3218 - val_acc: 0.5596
Training Set Evaluation:
Test loss: 1.321787141609192
Test acc: 0.5596
```

Figure 19. Test results using the SGD with Nesterov momentum optimiser

From the results shown in Figure 18, it shows that the SGD with Nesterov momentum optimiser allowed the model to attain the highest training accuracy thus far at 0.7108. However, the validation and test accuracy remains low at 0.5596, which shows overfitting again.

In the previous optimisers, their learning rate is always constant and remains fixed throughout all the epochs. However, there are optimisers that have adaptive learning rate methods such as Adagrad, RMSprop and Adam. These adaptive learning rate methods brings the added benefit of removing the need to tune the learning rate by simply using the default value.

For RMSprop, the results of the test can be seen from Figure 19 below, where the accuracies of the training, validation and test datasets are lower than that of the sample classifier. However, it is also important to note that the difference between the training accuracy and test accuracy is much smaller than that of the sample classifier. This suggests that if the model is allowed to train over a longer duration with more

epochs or if it is exposed to a larger dataset, the test accuracy could be much higher due to the lower degree of overfitting.

```
Epoch 100/100
50000/50000 [==============================] - 3s 66us/step - loss: 1.3549 - acc: 0.5232 - val_loss: 1.4674 - val_acc: 0.4923
Training Set Evaluation:
Test loss: 1.467399157333374
Test acc: 0.4923
```

Figure 20. Test results using RMSprop

The next adaptive learning rate method used was Adaptive Moment Estimation (Adam), where Adam stores both exponentially decaying average of past gradients and past squared gradients and can be considered to be a combination of RMSprop and SGD with classical momentum [15].

```
Epoch 100/100
50000/50000 [==============================] - 4s 74us/step - loss: 1.3302 - acc: 0.5224 - val_loss: 1.4117 - val_acc: 0.5014
Training Set Evaluation:
Test loss: 1.411689256286621
Test acc: 0.5014
```

Figure 21. Test results using the Adam optimiser

Similar to the model using the RMSprop optimiser, the Adam optimiser made the training, validation and test accuracies lower, while reducing the difference between training accuracy and test accuracy as seen in Figure 20. With additional training, the model could possibly attain higher accuracies.

As mentioned previously, Adam can be seen as a combination of RMSprop and SGD with classical momentum. Nadam on the other hand, is a combination of Adam and NAG.

```
Epoch 100/100
50000/50000 [==============================] - 4s 84us/step - loss: 1.4604 - acc: 0.4744 - val_loss: 1.4816 - val_acc: 0.4734
Training Set Evaluation:
Test loss: 1.4815608909606934
Test acc: 0.4734
```

Figure 22. Test results using the Nadam optimizer

As seen in Figure 21, the classifier with Nadam optimiser had low training, validation and test accuracies but the difference between the training and test accuracy is low. This means that there is low variance, but there is high bias as the training and test losses are high. However, with additional training, it could lead to increased test accuracies.

Summary:

Optimiser Used	**Training Loss**	**Test Loss**	**Training Accuracy**	**Test Accuracy**
SGD (Sample Classifier)	1.0169	1.269	0.6388	0.5572
SGD w momentum	0.8777	1.2888	0.6826	0.5627
SGD w NAG	0.8014	1.3217	0.7108	0.5596
RMSprop	1.3549	1.4673	0.5232	0.4923
Adam	1.3302	1.4116	0.5224	0.5014
Nadam	1.4604	1.4815	0.4744	0.4734

Table 4. Comparison of results for varying optimiser used

f. Effects of different activation functions

Among the different activation functions such as sigmoid, tanh and ReLU, ReLU is often used as it is less computationally demanding as compared to the other functions due to simpler mathematical operations.

For the experiment using sigmoid activation functions in the model, the results can be seen in Figure 22 below, where the accuracies are all the lowest thus far with the largest amount of loss as well. The tanh activation function results are in Figure 23.

```
Epoch 100/100
50000/50000 [==============================] - 3s 59us/step - loss: 1.7443 - acc: 0.3773 - val_loss: 1.7044 - val_acc: 0.3958
Training Set Evaluation:
Test loss: 1.704427645111084
Test acc: 0.3958
```

Figure 23. Test results using the sigmoid activation function

```
Epoch 100/100
50000/50000 [==============================] - 3s 60us/step - loss: 1.3477 - acc: 0.5214 - val_loss: 1.3649 - val_acc: 0.5142
Training Set Evaluation:
Test loss: 1.3648607599258422
Test acc: 0.5142
```

Figure 24. Test results using the tanh activation function

Summary:

Activation Function	Training Loss	Test Loss	Training Accuracy	Test Accuracy
Sigmoid	1.7443	1.7044	0.3773	0.3958
Tanh	1.3477	1.3648	0.5214	0.5142
ReLU (Sample Classifier)	1.0169	1.269	0.6388	0.5572

Table 5. Comparison of results for varying activation function

g. Effects of different objective / loss functions

In Keras, loss functions such as mean squared error, categorical crossentropy and binary crossentropy are available. When tested with the mean squared error loss function, the results can be seen in Figure 24 while for binary crossentropy, erroneous results were obtained.

```
Epoch 100/100
50000/50000 [==============================] - 3s 63us/step - loss: 0.0752 - acc: 0.3867 - val_loss: 0.0737 - val_acc: 0.4092
Evaluation with Test Set:
Test loss: 0.07372189099788666
Test acc: 0.4092
```

Figure 25. Test results using the mean squared error loss function

Summary:

Loss Function	Training Loss	Test Loss	Training Accuracy	Test Accuracy
Categorical Crossentropy (Sample Classifier)	1.0169	1.269	0.6388	0.5572
Mean Squared Error	0.0752	0.0737	0.3867	0.4092

Table 6. Comparison of results for varying loss function

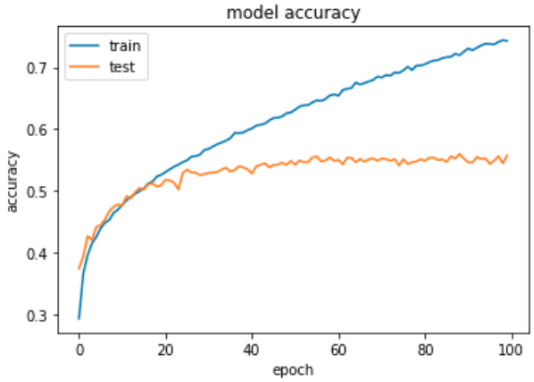
h. Improved MLP Classifier

My improved MLP classifier consists of 40,000 images in the training dataset, 10,000 in validation set and 10,000 in test set. Initially, after much testing, I was still unable to achieve a test accuracy of more than the sample classifier's 55.72% test accuracy. For example, in Figure 25, the model that I used consisted of 2 hidden dense layers with 512 neurons with the SGD with NAG optimiser with 0.01 learning rate and 0.9 momentum.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_71 (Dense)             (None, 512)               1573376
_________________________________________________________________
activation_71 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_49 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_72 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_72 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_50 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_73 (Dense)             (None, 512)               262656
_________________________________________________________________
activation_73 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_51 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_74 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_74 (Activation)   (None, 10)                0
=================================================================
Total params: 2,103,818
Trainable params: 2,103,818
Non-trainable params: 0
_________________________________________________________________
Training.....
Train on 40000 samples, validate on 10000 samples
```

Figure 26. Model with 2 additional dense hidden layers with 512 neurons

```
Epoch 100/100
40000/40000 [==============================] - 3s 77us/step - loss: 0.7131 - acc: 0.7429 - val_loss: 1.4315 - val_acc: 0.5569
Training Outcome:
```

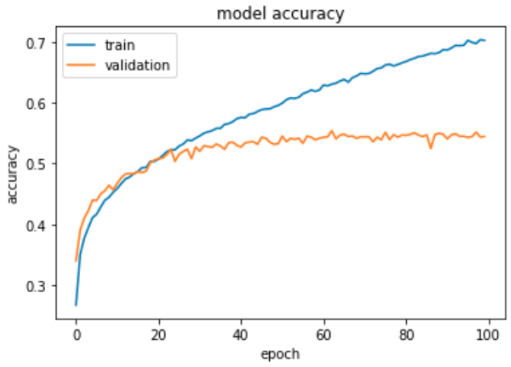



```
Evaluation with Test Set:
Test loss: 1.4149921003341674
Test acc: 0.5506
```

Figure 27. Test results of new model

As seen from Figure 26, the validation accuracy plateaued at approximately 0.53 accuracy while the training accuracy continued to rise. This shows that the model is overfitting and methods to reduce overfitting include increasing the size of the dataset, performing data augmentation, adding regularisation or to change the model architecture. However, it is not possible to increase the size of the CIFAR-10 dataset as it is a fixed set of data. For data augmentation, I attempted to implement width and height shifting together with horizontal flips for the images in CIFAR-10, but that proved to be too computationally intensive, with each epoch taking up to 100s to run. As seen in Figure 27, I increased the rate of the dropout layer from 0.2 to 0.5 and the training accuracy reduced to 70.25% but the test accuracy was still l at 54.04%.

```
Epoch 100/100
40000/40000 [==============================] - 4s 112us/step - loss: 0.8280 - acc: 0.7025 - val_loss: 1.4253 - val_acc: 0.5442
Training Outcome:
```

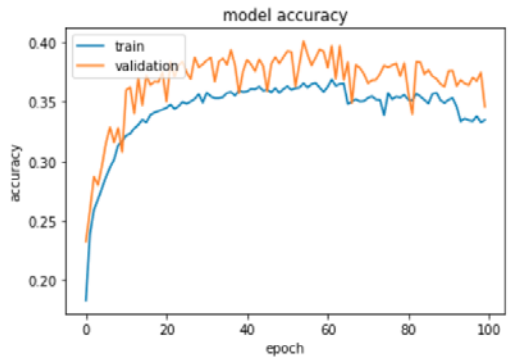



```
Evaluation with Test Set:
Test loss: 1.4231988216400147
Test acc: 0.5404
```

Figure 28. Increasing the dropout rate

When tested with different optimisation methods such as Nadam, which had the least amount of overfitting as seen in Table 4, the overfitting was reduced but the training and test accuracies were lower as well as seen in Figure 28.

```
Epoch 100/100
40000/40000 [==============================] - 5s 124us/step - loss: 1.8243 - acc: 0.3346 - val_loss: 1.8285 - val_acc: 0.3457
Training Outcome:
```




```
Evaluation with Test Set:
Test loss: 1.8163311065673828
Test acc: 0.3531
```

Figure 29. Using Nadam optimiser instead

If given sufficient time and resources, I would implement data augmentation and attempt to increase the data size to reduce the overfitting in the model. This would allow the model to generalise to the test dataset more effectively.

i. Comparison with other algorithms

When comparing with other state-of-the-art algorithms, most CIFAR-10 classifiers are not based on fully connected layers in a multilayer perceptron architecture and hence in comparison with the list compiled by Benenson [2], my results come in last among those state-of-the-art models. However, the paper by Lin and Memisevic [11] attempted to push the limits of fully connected networks without using convolution.

In their paper, the CIFAR-10 dataset was pre-processed by normalising the contract and centering to zero mean, before implementing PCA whitening. They used zero bias autoencoders (ZAE) instead of ReLU as the activation function and configured the network as 4000ZAE – 1000Linear – 4000ZAE with SGD optimiser, achieving an accuracy of 65.76%. After that test, they added data augmentation by flipping, rotating and shifting the original data and trained the model using the same architecture, which pushed the accuracy to 78.62%. Lin and Memisevic then claimed that this performance is the highest achieved by any fully connected network. In comparison to my classifier, without data augmentation, my results are approximately 10% behind that of Lin and Memisevic's model.

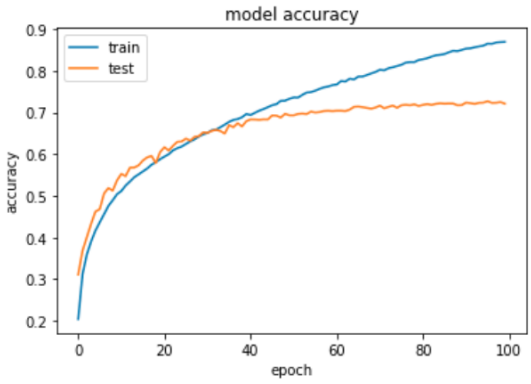
3. Convolutional Neural Network (CNN) Classifier

a. Sample CNN Classifier

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_102 (Activation)  (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_103 (Activation)  (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_70 (Dropout)         (None, 15, 15, 32)        0
_________________________________________________________________
flatten_11 (Flatten)         (None, 7200)              0
_________________________________________________________________
dense_98 (Dense)             (None, 512)               3686912
_________________________________________________________________
activation_104 (Activation)  (None, 512)               0
_________________________________________________________________
dropout_71 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_99 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_105 (Activation)  (None, 10)                0
=================================================================
Total params: 3,702,186
Trainable params: 3,702,186
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
```

Figure 30. Sample CNN Classifier model

```
Epoch 100/100
50000/50000 [==============================] - 9s 185us/step - loss: 0.3635 - acc: 0.8698 - val_loss: 0.8639 - val_acc: 0.7214
```




```
Test loss: 0.8638767498970031
Test acc: 0.7214
```

Figure 31. Test results for sample CNN classifier

For the sample CNN classifier, similar to the sample MLP classifier, the validation dataset was taken from the test dataset which is not ideal. For all future tests, the validation set will come from the training set in the form of 40,000 images for training, 10,000 for validation and 10,000 for testing.

The dataset was not pre-processed and did not have any data augmentation applied. The model is made of 1 2D convolutional input layer with ReLU activation, 1 2D convolutional hidden layer with ReLU activation, 2D Max Pooling with dropout, 1 dense fully connected hidden layer with ReLU activation and

dropout and a fully connected output layer with softmax activation as seen in Figure 29. The model was then compiled with the categorical crossentropy loss function with SGD optimiser.

The model ran for 100 epochs and the batch size was set to 128. For future tests a batch size of 64 will be used instead as there has been evidence that smaller batch sizes are beneficial in the training of CNN classifiers [17].

The test results from the sample classifier can be seen in Figure 30, where the training accuracy was at 86.98% and the test and validation accuracies at 72.14%. This result is significantly higher than any implementation of a MLP classifier thus far, higher than even the state-of-the-art MLP classifier without data augmentation mentioned previously.

For the following tests, models are run over 20 epochs for the initial test as from the first 20 epochs, it provides enough clarity as to the final test accuracy even if the model is trained over more epochs.

b. Effects of different parameter update rules

From the MLP classifier experiments, SGD with Nesterov Momentum can be seen to be superior to SGD with classical momentum. Therefore, the first experiment for varying the optimizer was using SGD with Nesterov momentum, with 0.01 learning rate and 0.9 momentum.

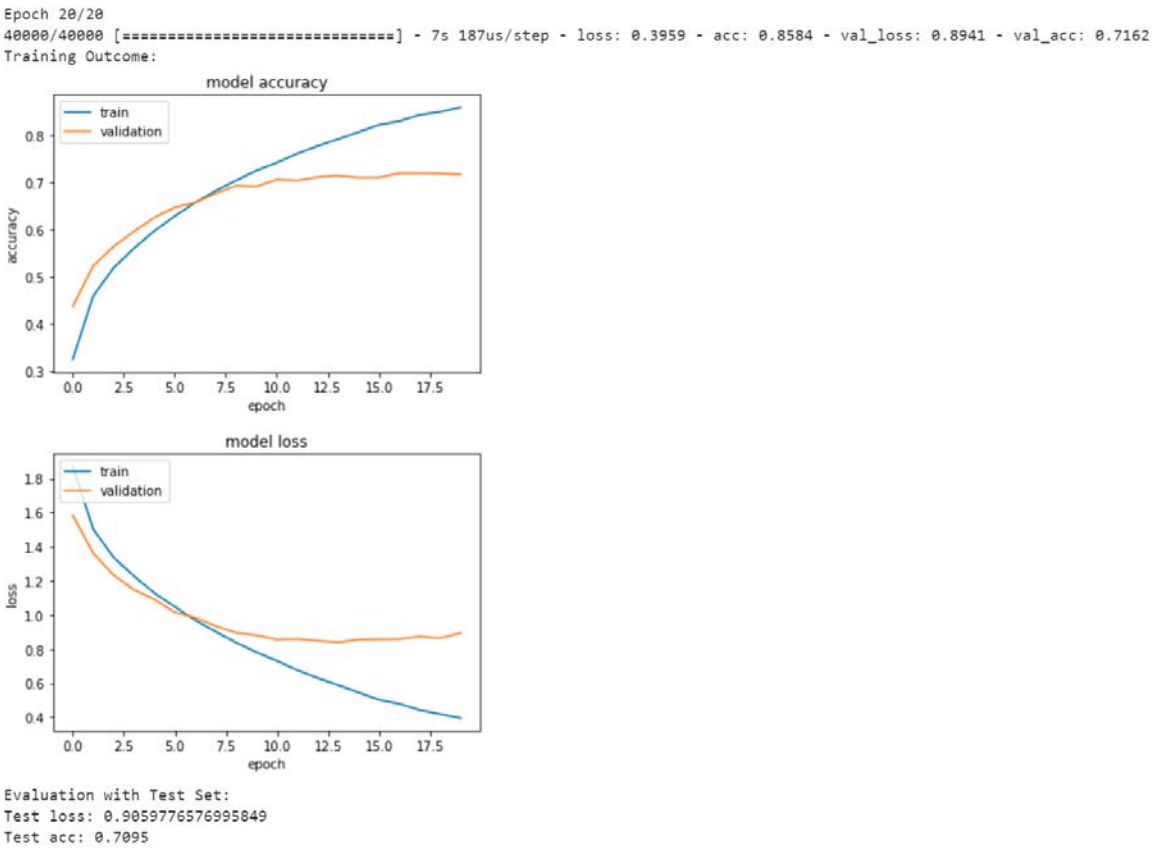


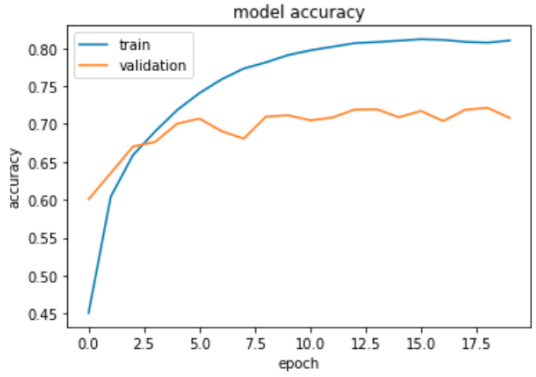
Figure 32. Test results using SGD with Nesterov momentum

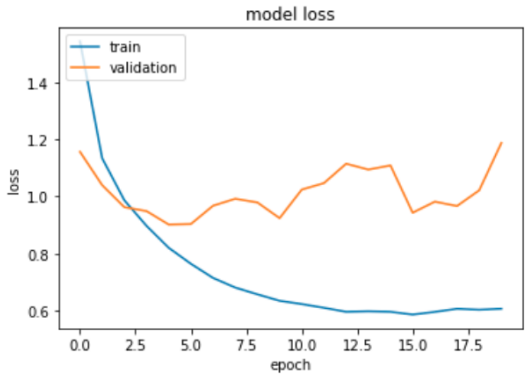
As seen from Figure 31, the validation accuracy plateaued at around 0.70 after 10 epochs, where the model started to overfit with the training accuracy higher than the test accuracy.

After testing with constant learning rate optimisers, I used adaptive learning rate methods such as RMSprop, Adam and Nadam, similar to that in the MLP experiments.

For the RMSprop and Adam optimiser in Figure 32 and 33 respectively, the validation accuracy stagnates at 0.70, similar to that in SGD with NAG. However, the training accuracy in the Adam optimiser shows an increasing rate that does not stagnate after 20 epochs, with a final training accuracy of 0.9115. This suggests that there is room for the model to train over more epoch with added normalisation to reduce the overfitting. This effect of the training accuracy can be seen in the Nadam optimiser in Figure 34, where over 100 epochs, the training accuracy increases to 0.96 while the validation accuracy stagnates at 0.70. If additional regularisation methods such as dropout or batch normalisation are added, it could lead to better validation and test accuracies.

```
Epoch 20/20
40000/40000 [==============================] - 12s 307us/step - loss: 0.6068 - acc: 0.8100 - val_loss: 1.1876 - val_acc: 0.7080
Training Outcome:
```

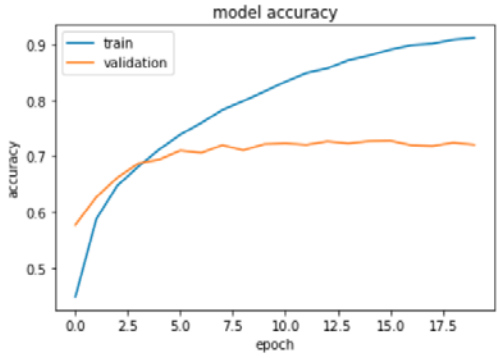


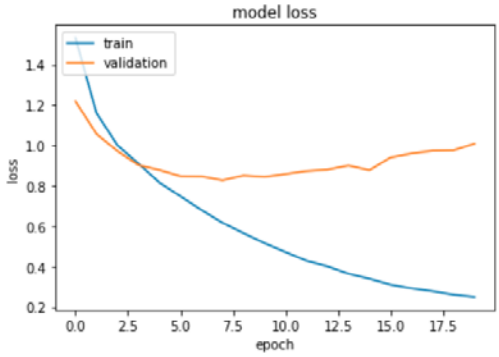



```
Evaluation with Test Set:
Test loss: 1.2132640821456908
Test acc: 0.6978
```

Figure 33. Test results using RMSprop

```
Epoch 20/20
40000/40000 [==============================] - 9s 232us/step - loss: 0.2496 - acc: 0.9115 - val_loss: 1.0088 - val_acc: 0.7205
Training Outcome:
```

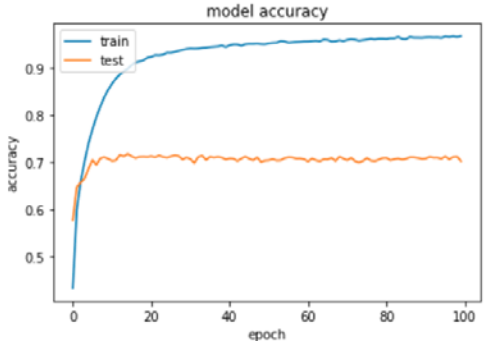


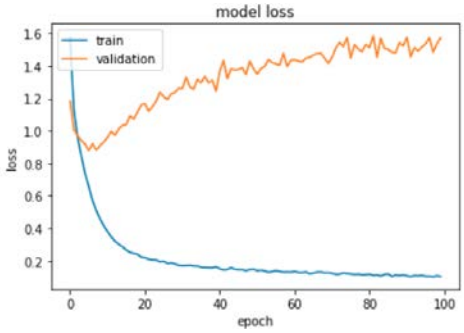



```
Evaluation with Test Set:
Test loss: 1.0076559057235719
Test acc: 0.7162
```

Figure 34. Test results using Adam

```
Epoch 100/100
40000/40000 [==============================] - 9s 232us/step - loss: 0.1025 - acc: 0.9678 - val_loss: 1.5715 - val_acc: 0.7028
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 1.5296847207069397
Test acc: 0.7061
```

Figure 35. Test results using Nadam

Summary:

Optimiser Used	Training Loss	Test Loss	Training Accuracy	Test Accuracy
SGD (Sample Classifier)	0.3635	0.8638	0.8698	0.7214
SGD with NAG	0.3959	0.9059	0.8584	0.7095
RMSprop	0.6068	0.8100	1.2132	0.6978
Adam	0.2496	1.0076	0.9115	0.7162
Nadam	0.1025	1.5296	0.9678	0.7061

Table 7. Comparison of results for varying optimiser in CNN architecture

c. Effects of adding batch normalisation

From the previous experiments, there were overfitting issues with the all the optimisers used. Therefore, I attempted to address these issues using batch normalisation on top of the already existing dropout layers provided in the sample model as seen in Figure 35. Batch normalisation works by normalising the outputs of the previous layer by subtracting the batch mean and dividing the batch standard deviation, increasing the stability of the neural network while adding slight regularisation effects [18].

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_9 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 30, 30, 32)        128
_________________________________________________________________
activation_10 (Activation)   (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_5 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 7200)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               3686912
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048
_________________________________________________________________
activation_11 (Activation)   (None, 512)               0
_________________________________________________________________
dropout_6 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_12 (Activation)   (None, 10)                0
=================================================================
Total params: 3,704,362
Trainable params: 3,703,274
Non-trainable params: 1,088
```

Figure 36. Adding batch normalisation to the sample model

Figure 36 shows the results from adding batch normalisation with a RMSprop optimised model. From the results, the validation accuracy can be seen to be more erratic as compared to without batch normalisation. However, the final test accuracy is higher at 0.7289 than when batch normalisation was not used as in

Figure 32. Similarly, the result for the Adam optimiser shows the same trend with the test accuracy increasing to 0.7321 as compared to 0.7162 without batch normalisation.

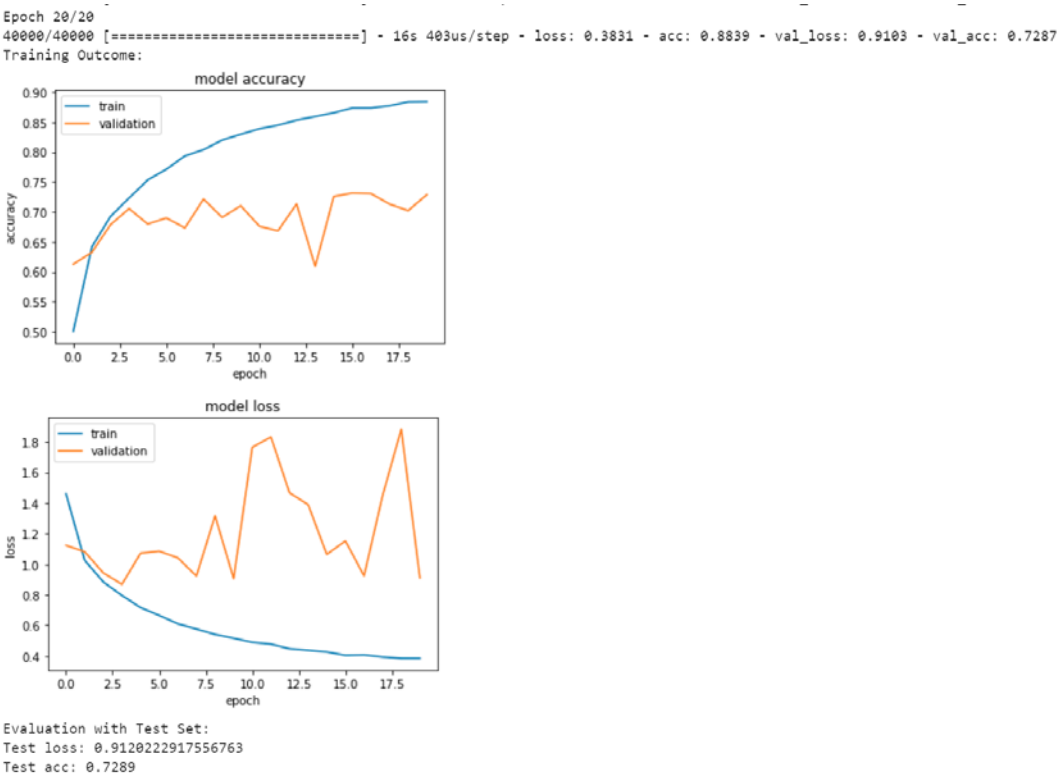


Figure 37. Test results using RMSprop with batch normalisation layers

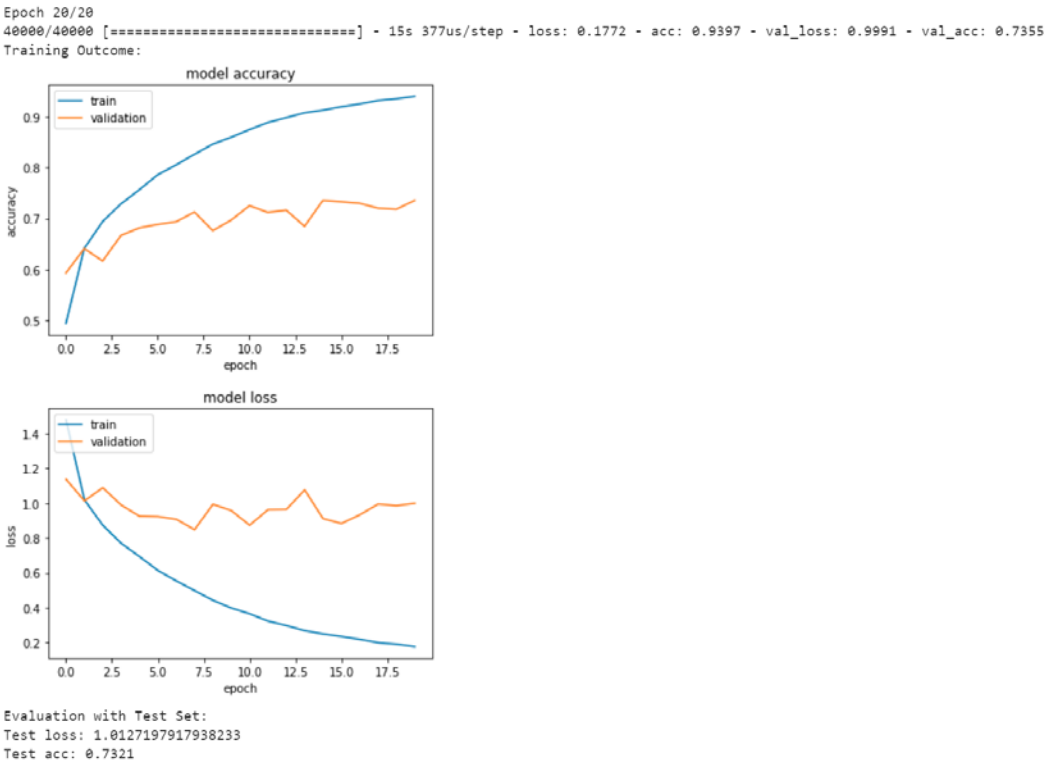
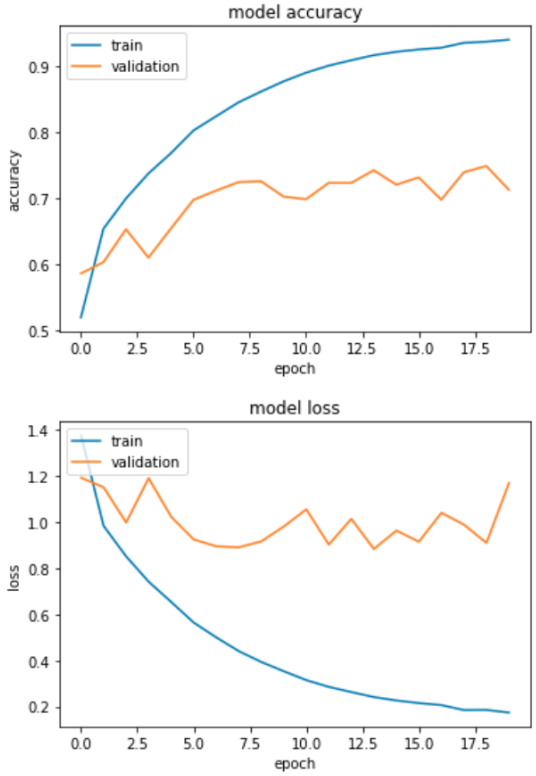


Figure 38. Test results using Adam with batch normalisation layers

```
Epoch 20/20
40000/40000 [==============================] - 15s 387us/step - loss: 0.1736 - acc: 0.9391 - val_loss: 1.1693 - val_acc: 0.7126
Training Outcome:
```




```
Evaluation with Test Set:
Test loss: 1.2399604826450348
Test acc: 0.7013
```

Figure 39. Test results using Nadam with batch normalisation layers

Summary:

Optimiser Used	**Training Loss**	**Test Loss**	**Training Accuracy**	**Test Accuracy**
SGD (Sample Classifier)	0.3635	0.8638	0.8698	0.7214
RMSprop	0.6068	0.8100	1.2132	0.6978
RMSprop (with batch normalisation)	0.3831	0.9120	0.8839	0.7289
Adam	0.2496	1.0076	0.9115	0.7162
Adam (with batch normalisation)	0.1772	1.0127	0.9397	0.7321
Nadam	0.1025	1.5296	0.9678	0.7061
Nadam (with batch normalisation)	0.1736	1.2399	0.9391	0.7013

Table 8. Comparison of results for adding batch normalisation in CNN architecture

d. Effects of varying number of hidden layers

The next hyperparameter that I experimented with was the number of hidden layers. In this case with the CNN model, I only added a single additional 2D convolutional layer with batch normalisation, ReLU activation, 2D Max Pooling and drop out layers as seen in Figure 39.

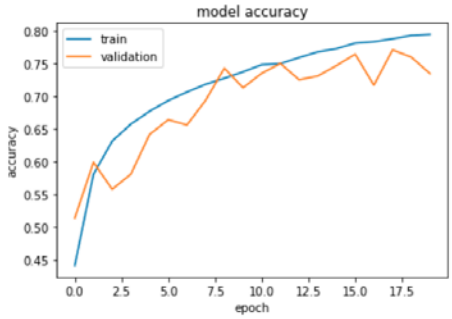
Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 32, 32, 32)	896
activation_17 (Activation)	(None, 32, 32, 32)	0
conv2d_10 (Conv2D)	(None, 30, 30, 32)	9248
batch_normalization_7 (Batch	(None, 30, 30, 32)	128
activation_18 (Activation)	(None, 30, 30, 32)	0
max_pooling2d_5 (MaxPooling2	(None, 15, 15, 32)	0
dropout_9 (Dropout)	(None, 15, 15, 32)	0
conv2d_11 (Conv2D)	(None, 13, 13, 32)	9248
batch_normalization_8 (Batch	(None, 13, 13, 32)	128
activation_19 (Activation)	(None, 13, 13, 32)	0
max_pooling2d_6 (MaxPooling2	(None, 6, 6, 32)	0
dropout_10 (Dropout)	(None, 6, 6, 32)	0
flatten_5 (Flatten)	(None, 1152)	0
dense_9 (Dense)	(None, 512)	590336
batch_normalization_9 (Batch	(None, 512)	2048
activation_20 (Activation)	(None, 512)	0
dropout_11 (Dropout)	(None, 512)	0
dense_10 (Dense)	(None, 10)	5130
activation_21 (Activation)	(None, 10)	0

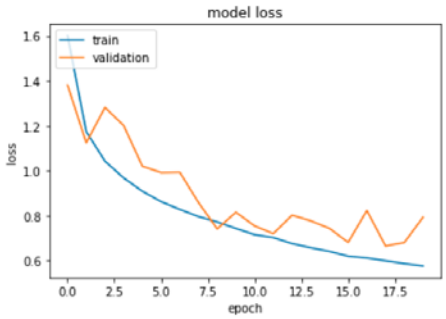
Total params: 617,162
Trainable params: 616,010
Non-trainable params: 1,152

Figure 40. CNN model with additional 2D Convolutional layer

When testing with the adam optimiser the results in Figure 40 as compared to that in Figure 37 shows a lower amount of overfitting, with the training accuracy reducing to 0.79 and the test accuracy remaining similar at 0.7313.

```
Epoch 20/20
40000/40000 [==============================] - 12s 291us/step - loss: 0.5769 - acc: 0.7941 - val_loss: 0.7928 - val_acc: 0.7346
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.8149439834594726
Test acc: 0.7313
```

Figure 41. Test results for Adam optimiser with additional convolution layer and batch normalisation

I then increased the number of training epochs to see any further improvements on the model with additional training. From the results in Figure 41, the test accuracy increased by a large margin, with the validation and training accuracies seen in the graph to be still continuing to rise, suggesting that over larger epochs, the accuracy is likely to increase even further.

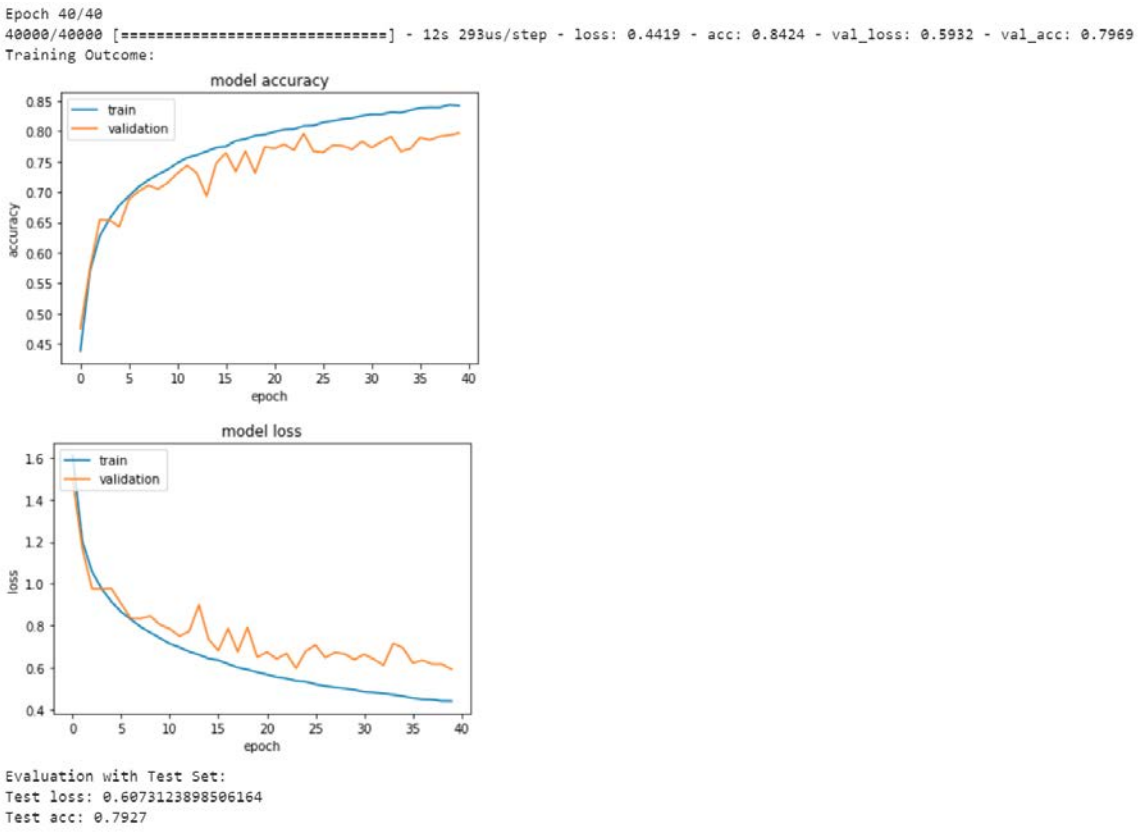
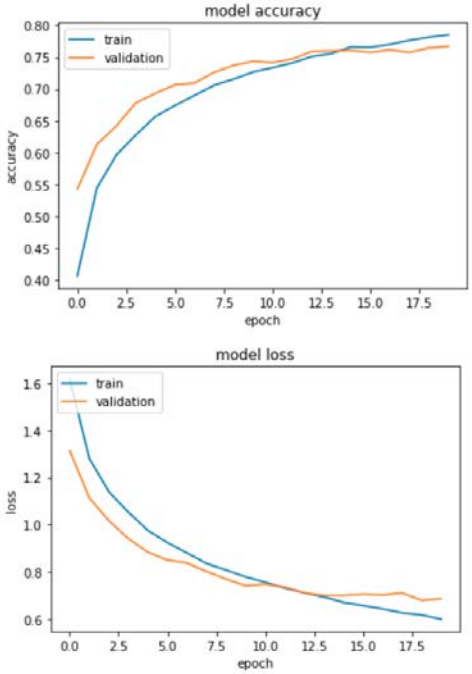


Figure 42. Test results for Adam optimiser with additional convolution layer and batch normalisation 40 epochs

As batch normalisation causes the validation accuracy to be more volatile, I experimented with the additional convolution layer with the Adam optimiser without batch normalisation. From the results seen in Figure 42, the validation accuracy graph does smoothen out, and the test accuracy increases to 0.76. However, the graph shows that the training accuracy has started to increase more so than the validation accuracy and if the model is trained over more epochs, the model is likely to overfit, as seen in Figure 43. This hence suggests that batch normalisation would allow the model to be trained over larger number of epochs to achieve higher test accuracies.

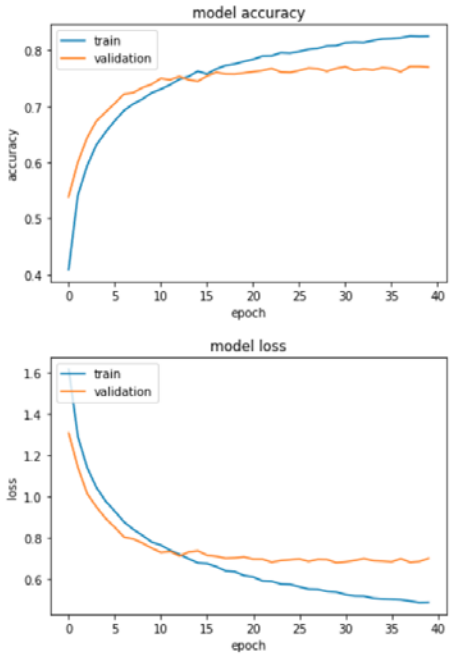
```
Epoch 20/20
40000/40000 [==============================] - 10s 262us/step - loss: 0.6011 - acc: 0.7851 - val_loss: 0.6871 - val_acc: 0.7667
Training Outcome:
```




```
Evaluation with Test Set:
Test loss: 0.6894058220863343
Test acc: 0.7658
```

Figure 43. Test results for Adam optimiser with additional convolution layer

```
Epoch 40/40
40000/40000 [==============================] - 10s 249us/step - loss: 0.4894 - acc: 0.8253 - val_loss: 0.7021 - val_acc: 0.7702
Training Outcome:
```

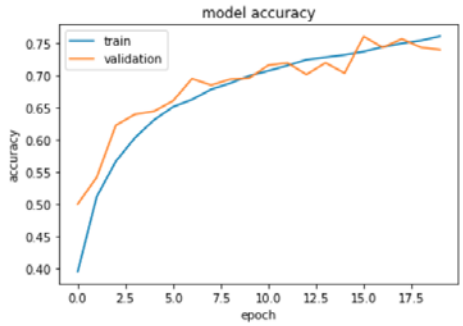


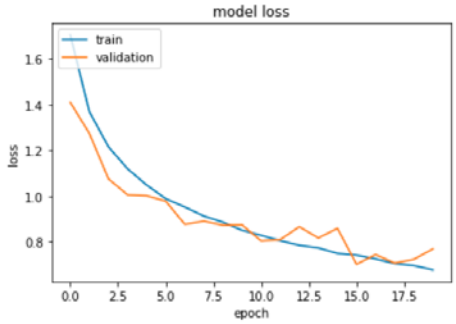

```
Evaluation with Test Set:
Test loss: 0.7217104276657105
Test acc: 0.762
```

Figure 44. Test results for Adam optimiser with additional convolution layer 40 epochs

I then tested the same setup with the SGD with Nesterov momentum and Nadam instead of Adam optimiser, with the results as seen in Figures 44, 45 and 48.

```
Epoch 20/20
40000/40000 [==============================] - 11s 273us/step - loss: 0.6766 - acc: 0.7601 - val_loss: 0.7685 - val_acc: 0.7391
Training Outcome:
```

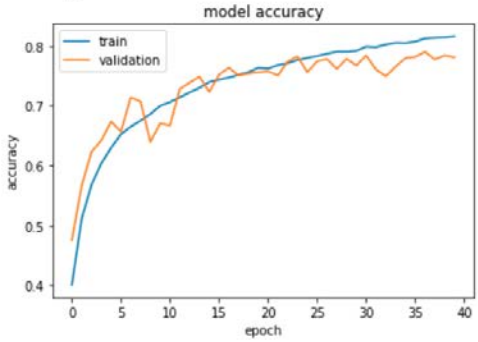


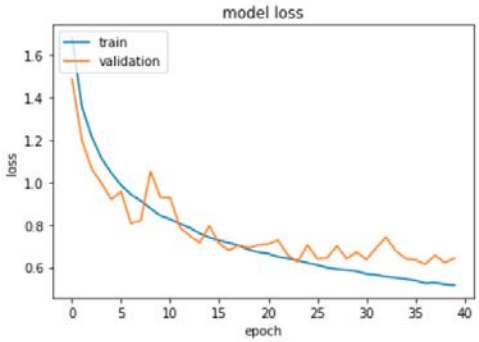



```
Evaluation with Test Set:
Test loss: 0.7899510277748107
Test acc: 0.7303
```

Figure 45. Test results for SGD w NAG optimiser with additional convolution layer and batch normalisation

```
Epoch 40/40
40000/40000 [==============================] - 11s 275us/step - loss: 0.5156 - acc: 0.8159 - val_loss: 0.6442 - val_acc: 0.7808
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.6557290950298309
Test acc: 0.7734
```

Figure 46. Test results for SGD w NAG optimiser with additional convolution layer and batch normalisation 40 epochs

Similarly, the results for SGD with Nesterov momentum and Nadam without batch normalisation can be seen in Figures 46, 47 and 49, where the results begin to show overfitting.

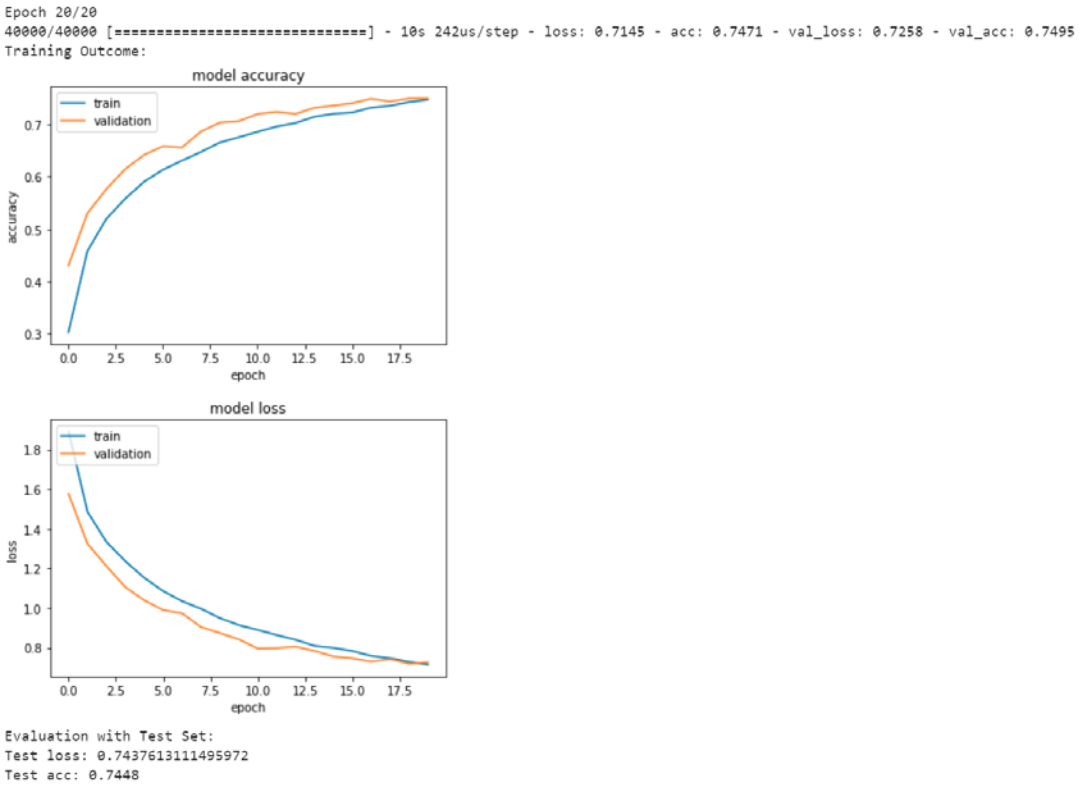


Figure 47. Test results for SGD w NAG optimiser with additional convolution layer

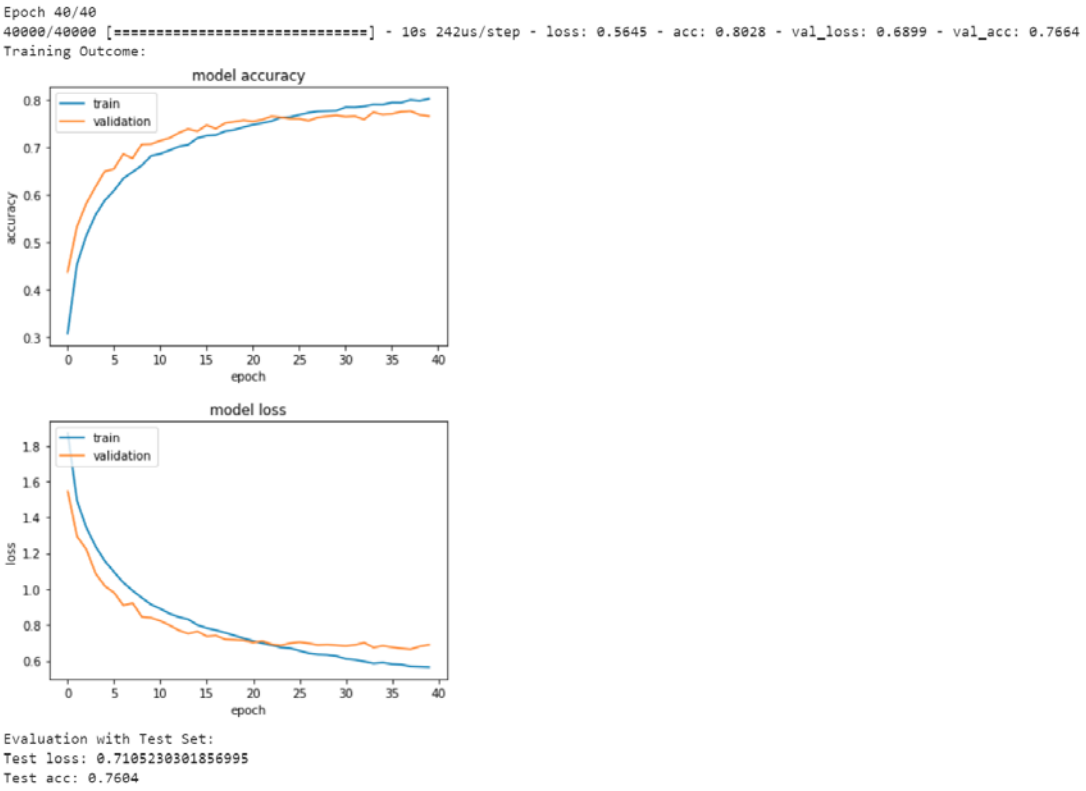
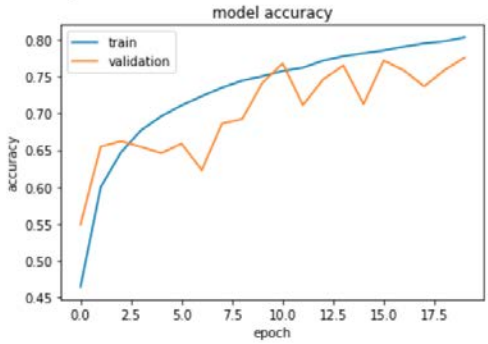
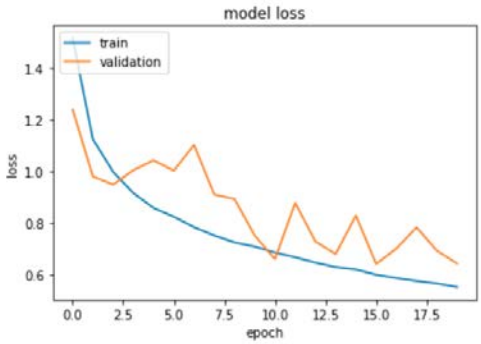


Figure 48. Test results for SGD w NAG optimiser with additional convolution layer 40 epochs

```
Epoch 20/20
40000/40000 [==============================] - 14s 344us/step - loss: 0.5536 - acc: 0.8031 - val_loss: 0.6437 - val_acc: 0.7757
Training Outcome:
```

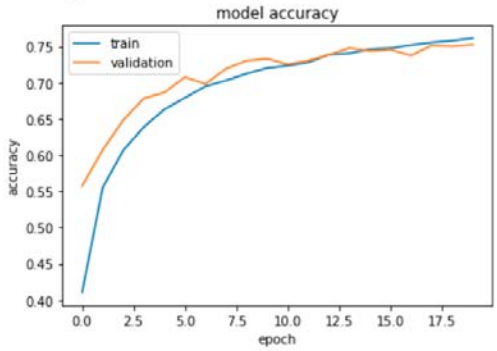


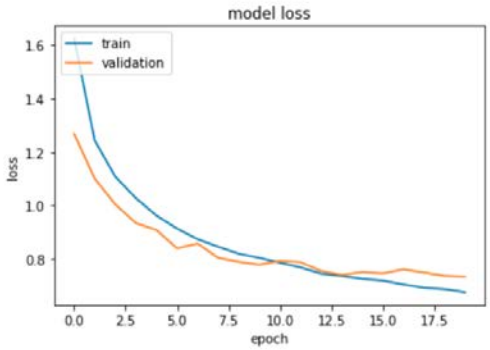



```
Evaluation with Test Set:
Test loss: 0.6640146744728088
Test acc: 0.771
```

Figure 49. Test results for Nadam optimiser with additional convolution layer and batch normalisation

```
Epoch 20/20
40000/40000 [==============================] - 12s 292us/step - loss: 0.6752 - acc: 0.7612 - val_loss: 0.7331 - val_acc: 0.7527
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.7465663311958313
Test acc: 0.75
```

Figure 50. Test results for Nadam optimiser with additional convolution layer

Summary:

Additional 2DConv Optimiser Used	Batch Norm Used?	Training Loss	Test Loss	Training Accuracy	Test Accuracy
SGD (Sample Classifier)	No	0.3635	0.8638	0.8698	0.7214
Adam (20 epochs)	Yes	0.5769	0.8149	0.7941	0.7313
Adam (40 epochs)	Yes	0.4419	0.6073	0.8424	0.7927
Adam (20 epochs)	No	0.6011	0.6894	0.7851	0.7658
Adam (40 epochs)	No	0.4894	0.7217	0.8253	0.762
SGD w NAG (20 epochs)	Yes	0.6766	0.7899	0.7601	0.7303
SGD w NAG (40 epochs)	Yes	0.5156	0.6557	0.8159	0.7734
SGD w NAG (20 epochs)	No	0.7145	0.7437	0.7471	0.7448
SGD w NAG (40 epochs)	No	0.5645	0.7105	0.8028	0.7604
Nadam (20 epochs)	Yes	0.5536	0.66401	0.8031	0.771
Nadam (20 epochs)	No	0.6752	0.7465	0.7612	0.75

Table 9. Comparison of results for additional convolution layer in CNN architecture

e. Effects of varying number of feature maps in the convolution layer

I attempted to increase the number of feature maps in the hidden convolution layer as seen in Figure 50 and used the Adam optimiser with batch normalisation. The results of the experiment shows a large degree of overfitting which would have to be managed using further regularisation methods or by reducing the complexity of the model.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_145 (Conv2D)          (None, 32, 32, 32)        896
_________________________________________________________________
activation_311 (Activation)  (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_146 (Conv2D)          (None, 30, 30, 64)        18496
_________________________________________________________________
batch_normalization_107 (Bat (None, 30, 30, 64)        256
_________________________________________________________________
activation_312 (Activation)  (None, 30, 30, 64)        0
_________________________________________________________________
max_pooling2d_84 (MaxPooling (None, 15, 15, 64)        0
_________________________________________________________________
dropout_155 (Dropout)        (None, 15, 15, 64)        0
_________________________________________________________________
flatten_49 (Flatten)         (None, 14400)             0
_________________________________________________________________
dense_167 (Dense)            (None, 512)               7373312
_________________________________________________________________
batch_normalization_108 (Bat (None, 512)               2048
_________________________________________________________________
activation_313 (Activation)  (None, 512)               0
_________________________________________________________________
dropout_156 (Dropout)        (None, 512)               0
_________________________________________________________________
dense_168 (Dense)            (None, 10)                5130
_________________________________________________________________
activation_314 (Activation)  (None, 10)                0
=================================================================
Total params: 7,400,138
Trainable params: 7,398,986
Non-trainable params: 1,152
```

Figure 51. Model with increased number of feature maps in convolution layer

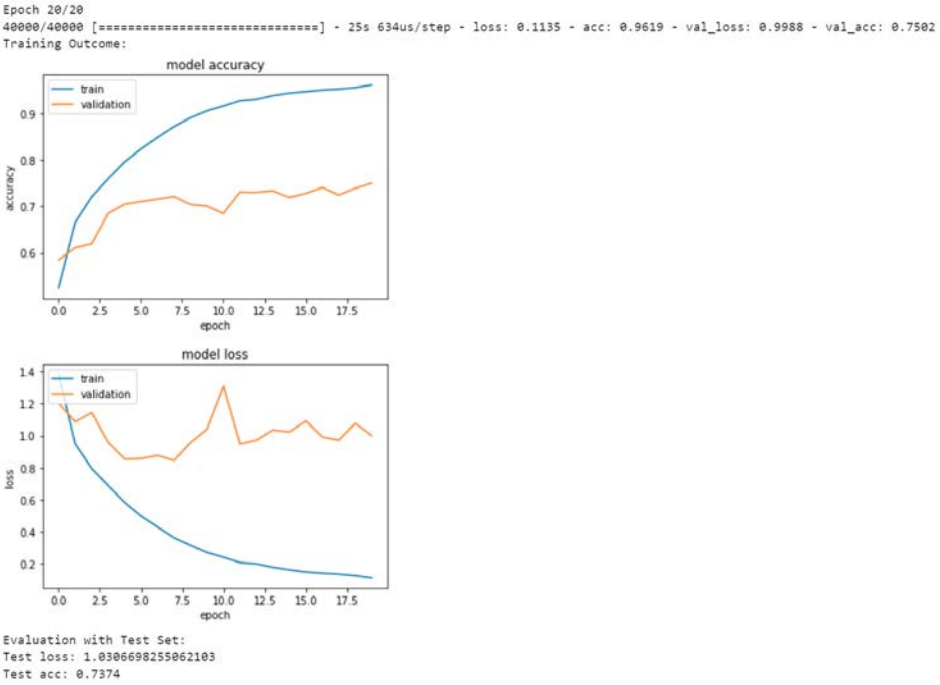


Figure 52. Test results for Adam optimiser with increased feature maps

Summary:

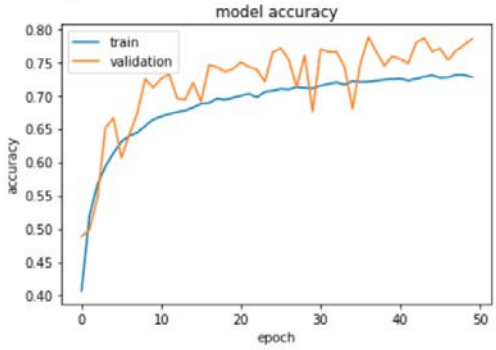
Number of feature maps in 2DConv layer	**Training Loss**	**Test Loss**	**Training Accuracy**	**Test Accuracy**
32 (Sample Classifier)	0.3635	0.8638	0.8698	0.7214
64	0.1135	1.0306	0.9619	0.7374

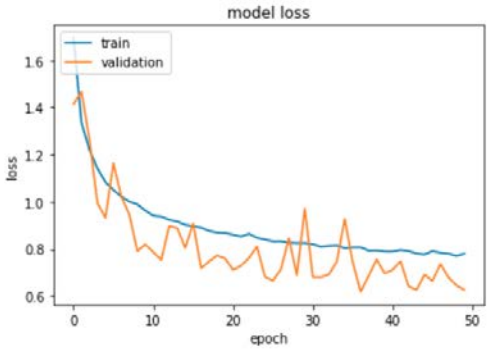
Table 10. Comparison of results for varying number of feature maps in 2D convolutional layer

f. Effects of data augmentation (Image translation & horizontal flips)

For the CNN classifier, I managed to implement data augmentation, which would allow the model to train over a larger dataset and to expose the classifier to a larger variety of images to increase its robustness during testing against unseen images. Figure 52 shows the results of data augmentation methods employed, namely width and height shifting, rotation as well as horizontal flips on the model as seen in Figure 40 (additional Convolutional layer, Adam optimiser with batch normalisation) over 50 epochs instead.

```
Epoch 50/50
1250/1250 [==============================] - 24s 19ms/step - loss: 0.7795 - acc: 0.7274 - val_loss: 0.6256 - val_acc: 0.7845
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.6405172392368317
Test acc: 0.7789
```

Figure 53. Test results for Adam optimiser with additional convolution layer, batch normalisation and data augmentation

Summary:

Addition of Data Augmentation	**Training Loss**	**Test Loss**	**Training Accuracy**	**Test Accuracy**
Sample Classifier	0.3635	0.8638	0.8698	0.7214
With data augmentation	0.7795	0.6405	0.7274	0.7789

Table 11. Comparison of results for adding data augmentation

g. Improved CNN Classifier

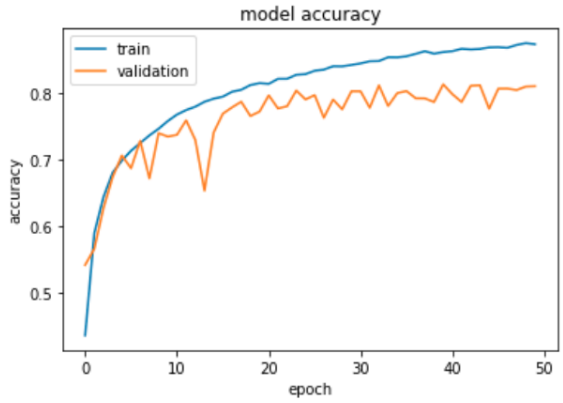
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_106 (Conv2D)          (None, 32, 32, 32)        896
_________________________________________________________________
activation_180 (Activation)  (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_107 (Conv2D)          (None, 30, 30, 32)        9248
_________________________________________________________________
batch_normalization_63 (Batc (None, 30, 30, 32)        128
_________________________________________________________________
activation_181 (Activation)  (None, 30, 30, 32)        0
_________________________________________________________________
conv2d_108 (Conv2D)          (None, 28, 28, 32)        9248
_________________________________________________________________
batch_normalization_64 (Batc (None, 28, 28, 32)        128
_________________________________________________________________
activation_182 (Activation)  (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_65 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
dropout_93 (Dropout)         (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_109 (Conv2D)          (None, 12, 12, 32)        9248
_________________________________________________________________
batch_normalization_65 (Batc (None, 12, 12, 32)        128
_________________________________________________________________
activation_183 (Activation)  (None, 12, 12, 32)        0
_________________________________________________________________
max_pooling2d_66 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
dropout_94 (Dropout)         (None, 6, 6, 32)          0
_________________________________________________________________
flatten_38 (Flatten)         (None, 1152)              0
_________________________________________________________________
dense_75 (Dense)             (None, 512)               590336
_________________________________________________________________
batch_normalization_66 (Batc (None, 512)               2048
_________________________________________________________________
activation_184 (Activation)  (None, 512)               0
_________________________________________________________________
dropout_95 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_76 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_185 (Activation)  (None, 10)                0
=================================================================
Total params: 626,538
Trainable params: 625,322
Non-trainable params: 1,216
```

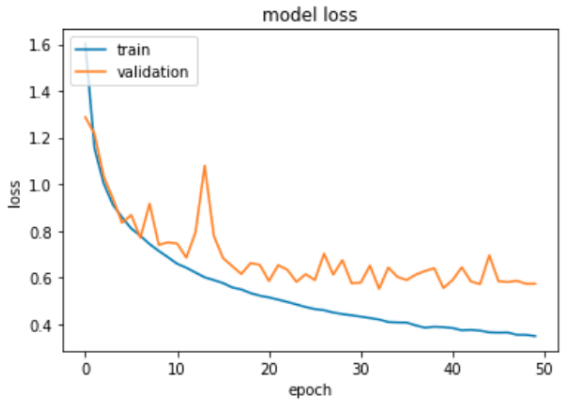
Figure 54. Improved CNN classifier model without data augmentation

In order to improve on the sample CNN classifier, the first approach that I took was implementing a classifier without data augmentation by increasing the number of convolution layers while maintaining adequate number of batch normalisation and dropout layers to balance the overfitting. After iterating through multiple experiments to get a high test accuracy and low overfitting, I then added data augmentation to increase the accuracy even more by training the classifier on augmented data. Lastly, I increased the number of training epochs to get higher accuracies by allowing the model to train over a longer period of time. All these models were optimised with the Adam optimiser.

The first model without data augmentation can be seen in Figure 53, with the results in Figure 54. I added batch normalisation layers between all convolution and activation layers, 1 set of Conv2D – BatchNorm – ReLU before the first MaxPooling2D layer, and 1 set of Conv2D – BatchNorm – ReLU – MaxPooling2D – Dropout layers before flattening to the dense layer.

```
Epoch 50/50
40000/40000 [==============================] - 16s 403us/step - loss: 0.3486 - acc: 0.8744 - val_loss: 0.5734 - val_acc: 0.8112
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.5764691339492798
Test acc: 0.8069
```

Figure 55. Test results for improved CNN classifier model without data augmentation

From the results seen above, the training accuracy seems to be still rising after 50 epochs, while the validation accuracy seems to have stagnated at approximately 0.8 which shows overfitting being present. However, with a test loss of 0.5764 and a test accuracy of 0.8069, it is already much better than the sample classifier's test results of 0.8638 loss and 0.7214 accuracy.

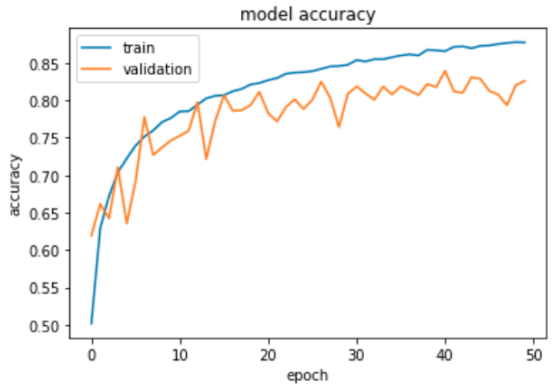
To improve further on the classifier, I decided to add data augmentation as seen in Figure 55.

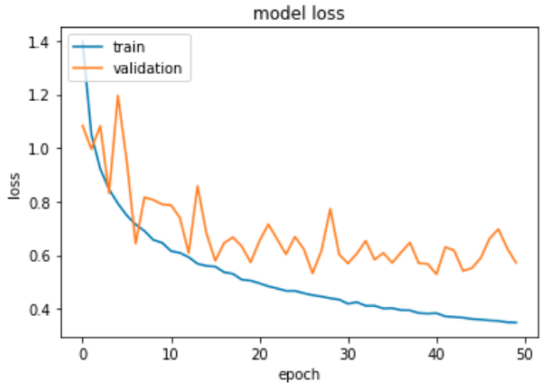
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_133 (Conv2D)          (None, 32, 32, 32)        896
_________________________________________________________________
activation_221 (Activation)  (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_134 (Conv2D)          (None, 30, 30, 32)        9248
_________________________________________________________________
batch_normalization_87 (Batc (None, 30, 30, 32)        128
_________________________________________________________________
activation_222 (Activation)  (None, 30, 30, 32)        0
_________________________________________________________________
conv2d_135 (Conv2D)          (None, 28, 28, 32)        9248
_________________________________________________________________
batch_normalization_88 (Batc (None, 28, 28, 32)        128
_________________________________________________________________
activation_223 (Activation)  (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_78 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_136 (Conv2D)          (None, 12, 12, 32)        9248
_________________________________________________________________
batch_normalization_89 (Batc (None, 12, 12, 32)        128
_________________________________________________________________
activation_224 (Activation)  (None, 12, 12, 32)        0
_________________________________________________________________
max_pooling2d_79 (MaxPooling (None, 6, 6, 32)          0
_________________________________________________________________
flatten_45 (Flatten)         (None, 1152)              0
_________________________________________________________________
dense_89 (Dense)             (None, 512)               590336
_________________________________________________________________
batch_normalization_90 (Batc (None, 512)               2048
_________________________________________________________________
activation_225 (Activation)  (None, 512)               0
_________________________________________________________________
dense_90 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_226 (Activation)  (None, 10)                0
=================================================================
Total params: 626,538
Trainable params: 625,322
Non-trainable params: 1,216
```

Figure 56. Improved CNN classifier model with data augmentation (50 epochs)

For the data augmentation, I added width and height shifts of 0.1, rotation of 10 degrees and horizontal flips to a data generator. However, when I experimented with using the exact same model as in Figure 53, there was underfitting, where the test and validation accuracies were consistently higher than the training accuracy. To mitigate this, I removed all of the dropout layers that were used previously as seen in Figure 55, and achieved the results seen in Figure 56.

```
Epoch 50/50
1250/1250 [==============================] - 26s 21ms/step - loss: 0.3491 - acc: 0.8774 - val_loss: 0.5720 - val_acc: 0.8259
Training Outcome:
```






```
Evaluation with Test Set:
Test loss: 0.5961983642816544
Test acc: 0.817
```

Figure 57. Test results for improved CNN classifier model with data augmentation (50 epochs)

The results show that the training and validation accuracies were still rising after 50 epochs, with slightly lesser overfitting than in the previous experiment without data augmentation. After 50 epochs, and a run time of approximately 25 minutes, the test results were 0.5961 loss and 0.817 accuracy.

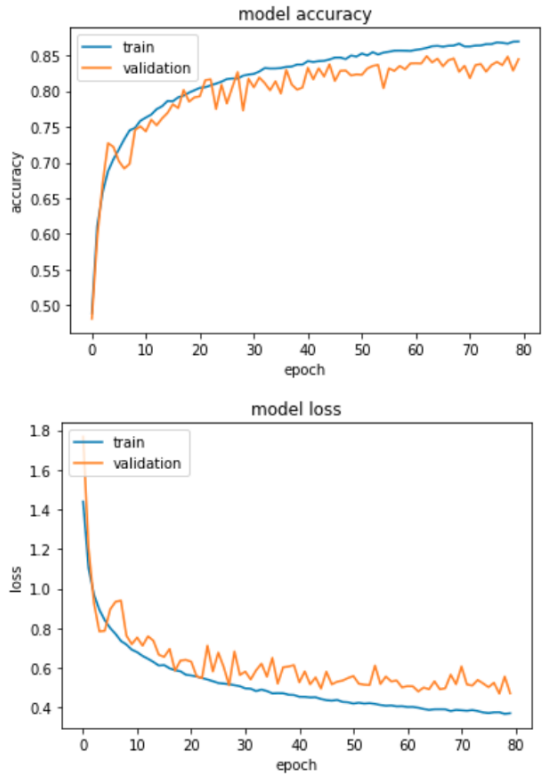
Layer (type)	Output Shape	Param #
conv2d_137 (Conv2D)	(None, 32, 32, 32)	896
activation_297 (Activation)	(None, 32, 32, 32)	0
conv2d_138 (Conv2D)	(None, 30, 30, 32)	9248
batch_normalization_99 (Batc	(None, 30, 30, 32)	128
activation_298 (Activation)	(None, 30, 30, 32)	0
conv2d_139 (Conv2D)	(None, 28, 28, 32)	9248
batch_normalization_100 (Bat	(None, 28, 28, 32)	128
activation_299 (Activation)	(None, 28, 28, 32)	0
max_pooling2d_80 (MaxPooling	(None, 14, 14, 32)	0
conv2d_140 (Conv2D)	(None, 12, 12, 32)	9248
batch_normalization_101 (Bat	(None, 12, 12, 32)	128
activation_300 (Activation)	(None, 12, 12, 32)	0
max_pooling2d_81 (MaxPooling	(None, 6, 6, 32)	0
flatten_46 (Flatten)	(None, 1152)	0
dense_161 (Dense)	(None, 512)	590336
batch_normalization_102 (Bat	(None, 512)	2048
activation_301 (Activation)	(None, 512)	0
dropout_150 (Dropout)	(None, 512)	0
dense_162 (Dense)	(None, 10)	5130
activation_302 (Activation)	(None, 10)	0

Total params: 626,538
Trainable params: 625,322
Non-trainable params: 1,216

Figure 58. Improved CNN classifier model with data augmentation (80 epochs)

I decided to allow the model to train over more epochs but the previous model with 50 epochs already had overfitting issues, if I were to increase the number of epochs, the overfitting would definitely be higher. Therefore, I added a dropout layer before the output layer as seen in Figure 57 to ensure that the overfitting would not become worse.

```
Epoch 80/80
1250/1250 [==============================] - 28s 23ms/step - loss: 0.3715 - acc: 0.8694 - val_loss: 0.4724 - val_acc: 0.8448
Training Outcome:
```




```
Evaluation with Test Set:
Test loss: 0.48672348487377165
Test acc: 0.842
```

Figure 59. Test results for improved CNN classifier model with data augmentation (50 epochs)

Finally, after 80 epochs and a run time of approximately 40 minutes, the results in Figure 55 was achieved. The training and validation accuracies closely follow each other as seen in the graph, with minimal amounts of overfitting. This shows that the dropout layer added managed to mitigate the overfitting issues. The final, best test results achieved out of all the experiments was 0.4867 loss and 0.842 accuracy.

Summary:

CNN Classifier	Training Loss	Test Loss	Training Accuracy	Test Accuracy
Sample Classifier	0.3635	0.8638	0.8698	0.7214
Improved Classifier, 50 epochs (without data augmentation)	0.3486	0.5764	0.8744	0.8069
Improved Classifier, 50 epochs (with data augmentation)	0.3491	0.5961	0.8774	0.817
Improved Classifier, 80 epochs (with data augmentation)	0.3715	0.4867	0.8694	0.842

Table 12. Comparison of results for improved CNN based CIFAR-10 classifier

For the final model, this specific architecture was chosen after multiple iterations through the different hyperparameters and this proved to be one of the most effective while striking a balance with run times. Due to the final model run time being more than 40 minutes, I did not iterate through the same model using different loss functions and optimisation methods. However, through the earlier experiments either

in the MLP architecture, I deduced that the loss function for this classifier should be categorical crossentropy regardless of the architecture as it seemed like the other loss functions such as binary crossentropy were not necessarily meant for a dataset with 10 classes. Regarding the optimisation methods, from earlier experiments with the CNN architecture, it seemed like the Adam optimiser with batch normalisation would be the most promising optimiser as the accuracy against number of epochs graph seemed to be in an increasing trend while other optimisers seemed to be plateauing.

An interesting observation through these experiments is that when batch normalisation was added, the validation losses and accuracies regardless of optimiser used or number of layers, would not be a smooth decrease or increase. Instead, the graph for both validation loss and accuracy would show spikes in their values.

Given sufficient time and resources to improve on my model, I would iterate my final model with different optimisers to see their impact and after that choose the most appropriate and continue increasing the number of epochs till overfitting is seen again before applying dropout or other regularisation methods. I would also try to implement a Recurrent Neural Network model separately to see how it performs with CIFAR-10.

h. Comparison with other algorithms

When comparing my results with that of other algorithms, my model did not perform too badly, having an accuracy higher than 11 other models among the 49 state-of-the-art models collected by Benenson [2].

This is due to the low number of layers used in this classifier as compared to the state-of-the-art, where models with millions of parameters and deeper architectures were trained over days and weeks such as in VGGNet [8] or GoogLeNet [7]. Furthermore, some of the methods included more advanced regularisation or activation functions such as Shake-shake regularisation [13].

Therefore, for my improved model, considering the amount of time that I trained my model for and the computational complexity it holds, it is still acceptable in its training accuracy.

4. References

[1] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, 2009, vol. 1.

[2] R. Benenson. (2016, 6 Nov). *What is the class of this image? Discover the current state of the art in objects classification.* Available: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

[3] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves," in *IJCAI*, 2015, vol. 15, pp. 3460-8.

[4] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International Conference on Machine Learning*, 2013, pp. 1058-1066.

[5] B. Graham, "Fractional max-pooling," *arXiv preprint arXiv:1412.6071*, 2014.

[6] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," *arXiv preprint arXiv:1412.6830*, 2014.

[7] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1-9.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770-778.

[10] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211-252, 2015.

[11] Z. Lin, R. Memisevic, and K. Konda, "How far can we go without convolution: Improving fully-connected networks," *arXiv preprint arXiv:1511.02580*, 2015.

[12] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "AutoAugment: Learning Augmentation Policies from Data," *arXiv preprint arXiv:1805.09501*, 2018.

[13] X. Gastaldi, "Shake-shake regularization," *arXiv preprint arXiv:1705.07485*, 2017.

[14] S. H. Hasanpour, M. Rouhani, M. Fayyaz, M. Sabokrou, and E. Adeli, "Towards Principled Design of Deep Convolutional Networks: Introducing SimpNet," *arXiv preprint arXiv:1802.06205*, 2018.

[15] S. Ruder. (2016, 7 Nov). *An overview of gradient descent optimization algorithms*. Available: http://ruder.io/optimizing-gradient-descent/index.html

[16] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139-1147.

[17] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[18] F. Doukkali. (2017, 6 Nov). *Batch normalization in Neural Networks*. Available: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c