CIFAR-10: Image Classification Report

Overview

This report outlines the process and results of building a Convolutional Neural Network (CNN) to classify images from the CIFAR-10 dataset. We first created and trained our own model, before employing transfer learning on the pre-trained DenseNet-121 architecture to try enhance model performance.

For both models, we applied mostly the same data preprocessing steps. The report will delve deeper into the preprocessing workflow before detailing the approaches taken in creating our own model and the transfer learning. For each of the models, we will summarize the performance and finally reflect on the insights gained in this exercise.

Step 1: Data Preprocessing

The CIFAR-10 dataset consists of 60.000, 32 x 32 colour images, divided into 10 categories of 6.000 images each (ranging from different animals to vehicles). This means that missing data and data imbalance were not an issue. However, the images are very small, leading to a loss of detail which might have been helpful in classifying the images correctly. We thus decided on a data driven approach, focussing on improving the quality of the data, in hopes of being able to use a smaller and simpler model.

The following steps were applied to both models unless stated otherwise:

1. **Grayscale Conversion:** Removing the RGB colour channels increases contrast and reduces complexity, making it easier for the model to recognise patterns. It also reduces the amount of input data to only one colour channel, which can lead to improved training capability. In transfer training, the images were not converted to grayscale, as DenseNet-121 was trained on colour images and thus requires the same input shape.
2. **Normalization:** Image pixel values were scaled to a [0, 1] range to facilitate faster convergence and consistent data distribution.
3. **One-Hot Encoding:** Class labels were encoded into a categorical format to enable readability for the models.
4. **Augmented data:** This was done to avoid overfitting and increase accuracy on testing. For both models we applied rotation and flipping. For the transfer learning we additionally applied zooming and contrast adjustments for a more aggressive augmentation.
5. **Increasing image size:** This only had to be done for the transfer learning model. As DenseNet was trained on images of size 124 x 124, the images from CIFAR-10 had to be resized for use with the model.

Figure 1: Overview of preprocessing steps applied to our own model

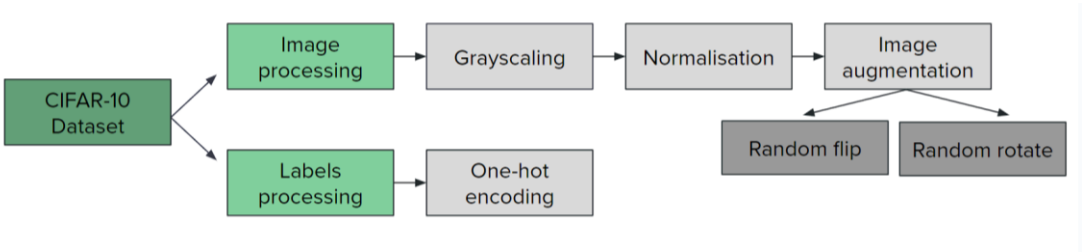
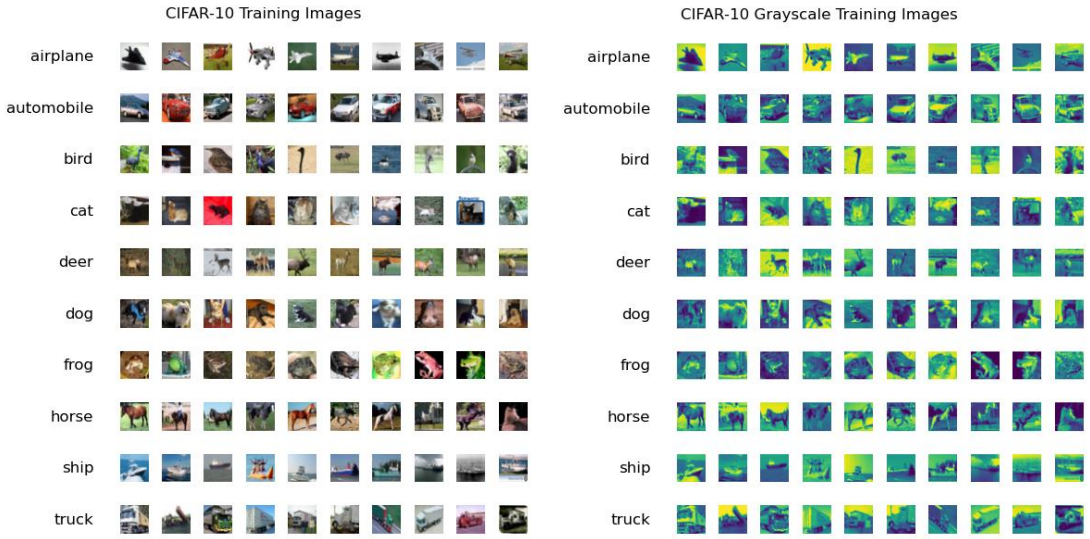


Figure 2: Visualization of images before and after grayscaling



Step 2: Model Development

CNN Architecture

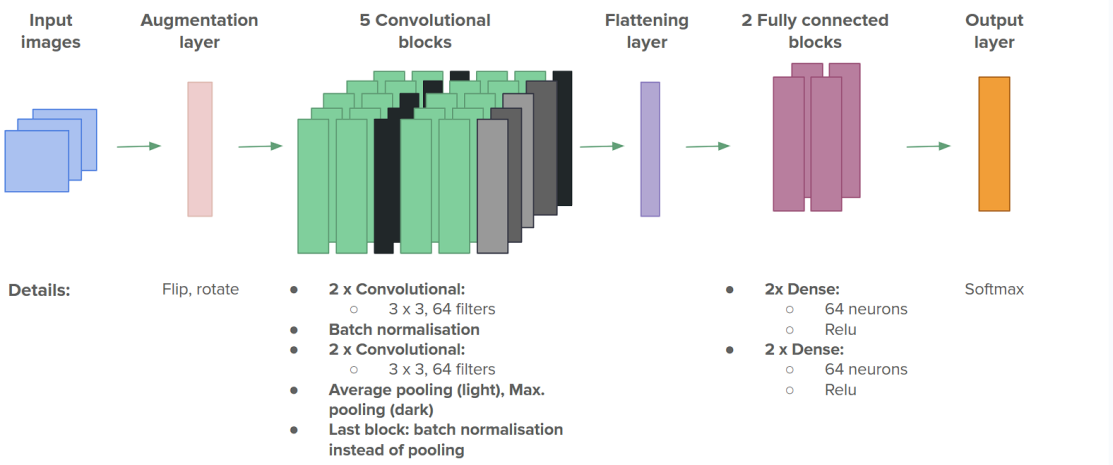
For our sequential CNN model, we decided to start out simple with only a few convolutional layers. However, after consistently not being able to achieve good results despite parameter tuning, we increased the model size. Eventually, we ended up with a much larger model than our starting prototype.

The final model version consists of:

- Input layer
- Augmentation layer
- 5 Convolutional (Conv.) blocks
    - 2 x Conv. layers (64 filters, 3 x 3, padding = same, activation = relu)
    - Batch normalisation layer
    - 2 x Conv. layers (64 filters, 3 x 3, padding = same, activation = relu)
    - 1 x Alternating AveragePooling and MaxPooling (we chose to use a Batch normalisation layer instead on the last block, to prevent the data from becoming too small)
- Flattening layer

- 2 Fully connected blocks
    - 2 x Dense layers (64 neurons, activation = relu)
- Output layer (activation = softmax)

Figure 3: Visualization of model architecture in the final model version



Training Process

We used the following parameter settings to train our model:

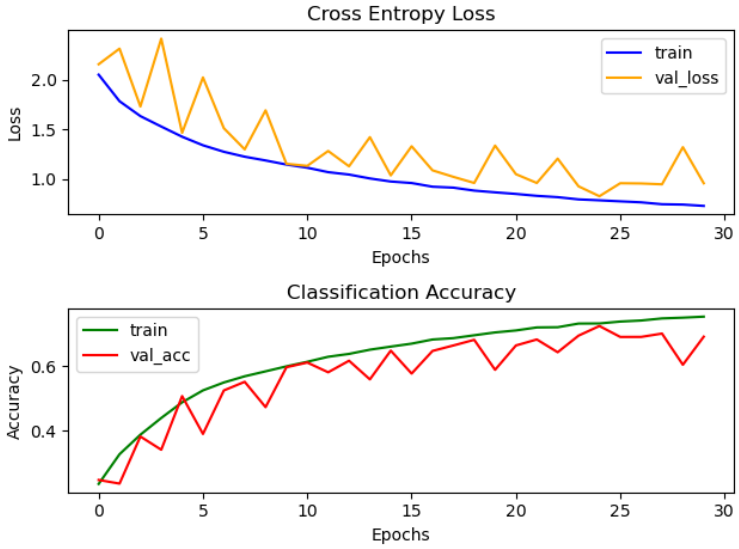
- **Optimizer:** Adam with a learning rate of 0.001. This worked better than the other optimizers (RMSprop, SDG). The custom learning rate of 0.001 also performed well.
- **Loss Function:** Categorical crossentropy, best suited for multi-class classification problems.
- **Batch Size:** 64, balancing training speed and model performance.
- **Epochs:** 30
- **Early stopping:** Implemented with a patience of 10 and monitoring validation loss, to prevent overfitting.
- **Validation set**: Training set was split with stratified shuffling to provide a balanced validation set for improved learning and monitoring of performance and overfitting

Results and Model Performance

- **Accuracy and Loss Trends:** The last version of the model achieved a training accuracy of 75.26% and validation accuracy of 69.06% after 30 epochs. Training loss was at 0.7284 with a validation loss of 0.9569. Even after many modifications from the original model, we did not manage to achieve a better result. A big problem for us was that the model would reach an early plateau at between 50-70% accuracy after 25-30 epochs. The validation accuracy also proved to be very inconsistent across the epochs, as can be seen by the zig-zag in the graph below. In previous attempts we also found the model to be underfitting, with an almost 50% lower validation accuracy when epochs were set to 100. We are unsure as to why there seems to be such a big difference between the validation and training

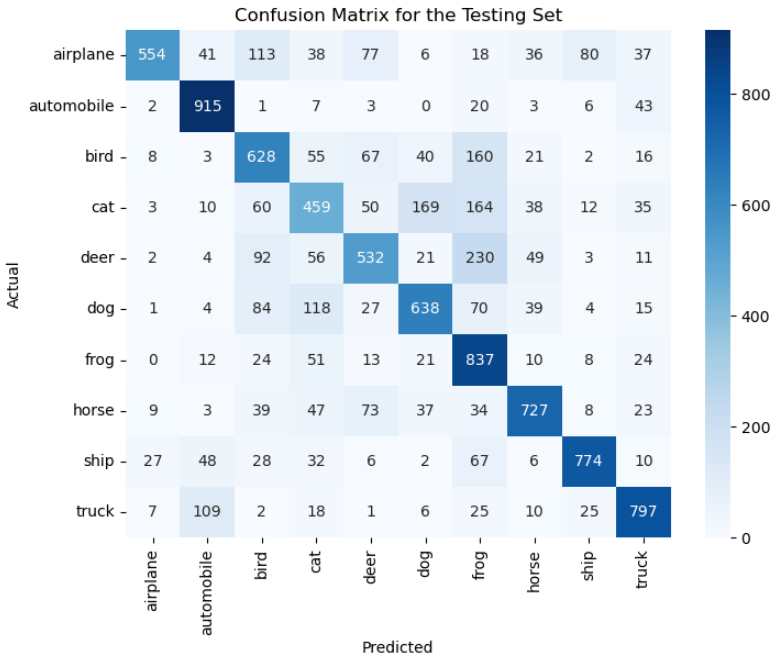
accuracy or why the models maxed out so early. Our improvement attempts are detailed further down.

Figure 4: Loss and accuracy curves of the final model



- **Confusion Matrix and testing performance:** The confusion matrix for the final testing, although not perfect, looked good overall and provided insights into class-specific performance. Specifically, it reveals higher misclassifications between certain classes like cats and dogs, as well as trucks and automobiles, which are visually similar. Interestingly, the model also seemed to confuse frogs and deer. The testing accuracy, precision, recall and F1 score also are within the range of the validation accuracy with:
    - Testing accuracy: 68.61%
    - Testing precision: 70.29%
    - Testing recall: 68.61%
    - Testing F1 score: 68.52%

Figure 5: Confusion matrix of the testing set



Steps taken to Improve the Results and Model Performance

We tried over 25 different variations of our model in total. Some of the techniques we tried to improve model performance are detailed below:

Technique	Observation
Starting with small/ simple model	Accuracy did not improve beyond 40-50%
Increase number of epochs	Allowed for better monitoring of performance over time. However, decreasing epochs from 100 to 30 had a significant increase of performance for our last model
Increase number of filters and neurons	No significant effect noted
Adding dropout layers	Dropout rate of 0.2 seemed ideal for the model, but these layers were taken out of the final model in an attempt to prevent underfitting
Different optimizers	Adam seemed to work best for the model
Different learning rates	Learning rate of 0.001 worked best
Adding regularization to convolutional layers	This was not leading to improved performance and was thus taken out
Different activation function	We tried using softmax on the second last dense layer. Removing this layer improved performance
Different batch size	Increasing batch size led to faster training, but smaller batch sizes increased accuracy. We settled on 64

Removing augmentation layer	This gave us our best performance of 82% training accuracy and 92.85% validation accuracy, but we decided to leave the augmentation layer in place to prevent overfitting
AveragePooling vs MaxPooling	Using AveragePooling increased our performance, but only with a smaller model. With a larger model, these lead to a loss of information and had to be combined with MaxPooling
Increasing model size	This improved our performance after tweaking of some parameters, but took longer to train and still not giving great performance. The larger size seemed to increase the underfitting problem

Step 3: Transfer Learning

CNN Architecture

After researching various CNN architectures available (mainly ResNet, VGG16, Inception), we decided on DenseNet-121. DenseNet is newer and larger than VGG16. It was also trained on CIFAR-10, CIFAR-100 and ImageNet. We thus expected the model to perform better on the CIFAR-10 dataset than comparable models, even though the reported accuracy on CIFAR-10 is around 95% which is not significantly better than smaller models like VGG16. Another advantage of DenseNet is the fact that the model uses connected layers where feature maps of previous convolutional layers are fed as input into the following layers. This aims to reduce vanishing gradient, further adding to the robustness of the model. Our choice of the smallest of the DenseNet models (121) was purely based on the size, as we were concerned about being able to run a larger model on our machines without using online GPUs like Paperspace. For further information on the DenseNet model, please refer to the 2018 report 'Densely Connected Convolutional Networks' by Huang et al.

Modifications for CIFAR-10:

- **Input Layer:** CIFAR-10 images are small (32x32), but DenseNet expects larger inputs (124 x 124), hence images were resized to match the expected input size of 3. We kept the images in RGB form as this is the input DenseNet was trained on CIFAR-10.
- **Base Layers:** We kept most of the original DenseNet layers, unfreezing the last 20 layers to connect to our additional layers. After training with 20 unfrozen layers, we increased this to 40 layers.
- **Additional Layers:** We added two layers before the output layer. Both are dense layers with 512 and 128 neurons respectively and an activation function of relu. The output layer was changed to fit our dataset with 10 neurons and a softmax activation function.

- **Data Argumentation:** In addition to the augmentation used for our own model, we introduced a more aggressive augmentation layer by adding Random Zoom & Random Contrast.

Training Process

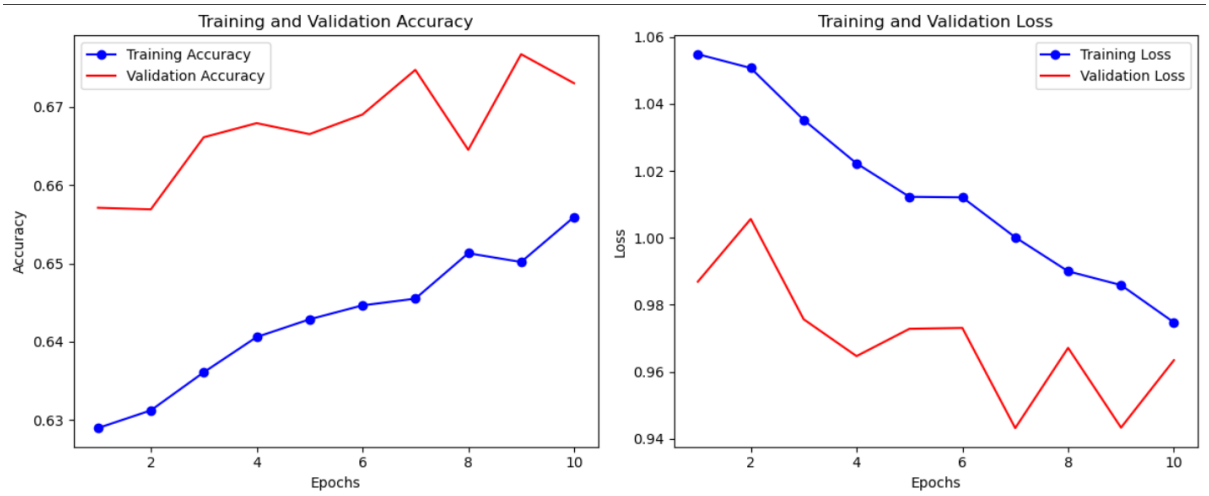
We used the following parameter settings to train our model:

- **Optimizer:** Adam with a learning rate of 0.01
- **Loss Function:** Categorical crossentropy
- **Batch Size:** 32 for better training outcomes
- **Epochs:** 20
- **Early stopping:** Implemented with a patience of 3 to prevent overfitting
- **Validation set**: Training set was split to provide a validation set for improved learning and monitoring of performance and overfitting

Results and Model Performance

- **Accuracy and Loss Trends:** After some improvements from the original design, the model achieved a training accuracy of over 65.45% and a validation accuracy of around 67.45% after 20 epochs. Training loss was at 0.9772 with a validation loss of 0.9246. Even after a few modifications from the original model, we did not manage to achieve a better result. We also retrained the model with 40 unfrozen layers, achieving a result of 65.92% training accuracy and 67.30% validation accuracy over 10 epochs. In this model, the validation accuracy also proved to be very inconsistent across the epochs, as can be seen by the zig-zag in the graph below. Over the training run, the training loss decreased steadily, indicating a good level of training, without overfitting.

Figure 6: Loss and accuracy curves of the final model



- **Confusion Matrix and testing performance:** Refer to section 1 Own Model. The results were similar on the DenseNet-121 transfer learning model.

Steps taken to Improve the Results and Model Performance

We had started the model with higher neurons in each of the dense layers (512 and 128 respectively), with a very low learning rate of 1e^-3. Our starting batch size was 64. However, after only getting a maximum performance of 53% training accuracy and 12% validation accuracy, we decided to decrease the number of neurons to converge more to the rest of the DenseNet architecture. Also, DenseNet was trained with a high training rate of 0.1, so we decided to increase our learning rate too. This led to better results overall, but in the final version we decided to increase the number of neurons to 512 and 128 again. We also trained the model with 40 unfrozen layers for 10 epochs, achieving a slightly better result than our model with 20 unfrozen layers.

Insights Gained

- **Importance of Data Preprocessing:** Properly normalized and encoded data significantly impacted model training effectiveness.
- **Effectiveness of Transfer Learning:** Using a pre-trained model reduced the need for an extensive dataset and lowered training times while improving model robustness.
- **Parameter Tuning:** Adjustments in learning rate and batch size showed noticeable effects on model performance, highlighting the importance of hyperparameter optimization in neural network training.

Conclusion

Our data-driven approach did not yield the desired results in either of the models. However, with experimentation we were able to improve the performance of our own developed model. Given more time, we would need to revisit the model architecture and parameters to try identify why the validation accuracy was so inconsistent and why the overall training accuracy reached an early plateau in many of the variants we tried.

The use of DenseNet for transfer learning proved

to be challenging for the CIFAR-10 image classification task. Our results started low and eventually plateaued, similarly to the own model.

However, irrespective of the model chosen, the most important thing we learnt is that experimentation is crucial and often, the best way to determine the optimal model is through empirical testing and tuning.

Future work could explore additional enhancements such as further hyperparameter tuning or experimenting with other advanced CNN architectures like Vision Transformers (ViT) or Xception for potentially even better performance.

This project demonstrated the power of CNNs in handling complex image classification tasks efficiently and effectively, particularly when combined with strategies like transfer learning.