Image Classification with CNNs

An IRONHACK project

This report describes the development and experimentation process of building a Convolutional Neural Network (CNN) designed for image classification using the CIFAR-10 dataset. The document details the chosen architecture, including modifications to enhance model performance, and provides a thorough explanation of the preprocessing steps applied to the data. It includes a deep dive into the training process, covering step by step all the tweaks and additions to the model, from learning rate, batch size or the optimizer to weight adjustment and loss functions at the end. Model performance is evaluated, with an emphasis on the best-performing model, supported by analysis and insights gained from various experiments. Visualizations and diagrams are included to aid in understanding the approach and results.

1. CNN's Architecture

The final architecture was designed with three convolutional blocks, each progressively increasing the number of filters (64, 128, and 256) to extract higher-level features. Each block applies two convolutional layers with batch normalization, ReLU activation, and max-pooling for down-sampling, followed by dropout to prevent overfitting. Data augmentation is applied after the Input to improve generalization. After feature extraction, a fully connected dense layer with L2 regularization and dropout is employed, leading to a final Softmax layer for classification across the 10 classes.

- **Input Layer**
- **Augmentation Layers**
  - Random Translation (V/H, 0.05)
  - Random Flip (Horizontal)
- **1st Convolutional Block**
  - Conv2d (64 filters, 3x3) + Batch Normalization + Activation (Relu)
  - Conv2d (64 filters, 3x3) + Batch Normalization + Activation (Relu)
  - MaxPooling2D (2, 2)
  - Dropout (0.1)
- **2nd Convolutional Block**
  - Conv2d (128 filters, 3x3) + Batch Normalization + Activation (Relu)
  - Conv2d (128 filters, 3x3) + Batch Normalization + Activation (Relu)
  - MaxPooling2D (2, 2)
  - Dropout (0.2)
- **3rd Convolutional Block**
  - Conv2d (256 filters, 3x3) + Batch Normalization + Activation (Relu)
  - Conv2d (256 filters, 3x3) + Batch Normalization + Activation (Relu)
  - MaxPooling2D (2, 2)
  - Dropout (0.5)
- **Dense Layers**
  - Flatten
  - Dense (128, L2 Regularizer (0.005)) + Batch Normalization + Activation (Relu)
  - Dropout (0.4)
- **Output**
  - Dense (10) + Activation (SoftMax)

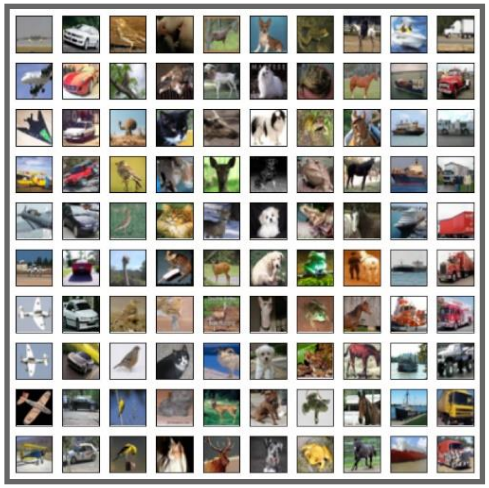
2. Preprocessing Steps

After selecting the CIFAR-10 dataset for its balance between simplicity, versatility, and inherent classification challenges, we began by loading the data and replacing the numerical class labels with more readable, human-friendly names.

```
custom_class_names = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

The next step was plotting a grid of 10 randomly sampled images from each class to be able to visualize the dataset.

Next, we visualized the dataset by plotting a grid of 10 randomly sampled images from each class to gain a better understanding of its contents. Afterward, we converted the class labels to categorical values and normalized the images to the **0-1** range while preserving the three-color channels.



We also experimented with global data augmentation and class-targeted data augmentation, particularly for the classes with the poorest performance. While these techniques helped reduce overfitting, they often resulted in a slight reduction in accuracy and significantly increased training time (up to 4 times longer).

Finally, we implemented data augmentation directly in the model's architecture. After testing several configurations, we found that applying horizontal flips and image translations provided the best results, reducing overfitting and improving test accuracy in the final model.

```
model.add(RandomTranslation(height_factor=0.05, width_factor=0.05))
model.add(RandomFlip("horizontal"))
```

3. Training Process

We decided to start by training a different model each to cover more terrain. We were of course documenting the steps taken and putting major advancements in common. For brevity's sake most of the individual tweaks will be omitted, focusing only on the important changes.

The starting models were slight variations of the ones from the Lab on CNNs:
2 Conv2D(32) -> MaxPooling2D -> Flatten -> Dense(output)

Early on we set the padding to 'same' to avoid the size dropping from filtering. And we introduced Batch Normalization before the ReLU activation layers, which improved generalization.

```
Conv2D(64, (3, 3), padding='same'), BatchNormalization(), Activation("relu"),
```

We changed from SGD to Adam optimizer increasing accuracy from 0.6 to 0.8. Probably because the model was getting more complex, and SGD doesn't perform well in deeper models.
Drop out layers were also introduced after the Convolutional (0.2) and Dense layers (0.6), getting Training and Validation Accuracy closer together.

Next, we introduced Early Stopping on 'max' for the Validation Accuracy and increased the patience to 5 to improve training efficiency and reduce overfitting in the last Epochs

```
early_stopping = EarlyStopping(monitor='val_accuracy', patience=5, mode='max', restore_best_weights=True)
```

As we struggled to improve above 0.8 Accuracy, we decided to add depth to the model to help with complex feature extraction. Step by step we added a 2nd and 3rd Convolutional Block and increased the number of filters to 64, 128 and 256 respectively. Next step was another fully connected layer with 128 neurons, and the last change was to reduce batch size from 512 to 128 to further improve generalization as the model is less likely to memorize the training data.
With the new architecture we reached 0.95 / 0.85 (train / val) Accuracy and decreased the Loss from 0.7 to around 0.4. (We later tried both adding more Conv2D / Dense layers and further increasing the number of Filters and Neurons; But the gains were negligible for the increased computational cost).

Our next step was to tackle the overfitting. We introduced L2 Regularization in the Dense layer, (ending up at 0.005) and skipping Conv2D layers because normalization and dropout were already helping with overfitting, and we didn't want to impact feature extraction in the Conv2D layers.

We implemented a Learning Rate Scheduler to reduce the learning rates when hitting plateaus and increased the epochs to 100 consequently (although it will rarely go over 60).

As the model was still struggling with high complexity features, we added class weights for the model to penalize mistakes and to try tackling the worse performing classes (cats and dogs).

```
class_weights = compute_class_weight(
    class_weight='balanced', classes=np.unique(y_train.argmax(axis=1)), y=y_train.argmax(axis=1))
class_weights_dict = dict(enumerate(class_weights))
class_weights_dict[3] *= 2; class_weights_dict[5] *= 2
```

At this point we were sitting at around 0.97 / 0.89 Accuracy, but our Loss was still rounding 0.4 so out last addition was to define a focal loss function aiming to emphasize the loss from classes that were proving harder to classify, putting more work on harder classes and less on the easier to identify ones. Train accuracy dropped slightly but Validation accuracy hit 0.9 for the first time; And more importantly we skimmed the Loss from around 0.3 all the way to the final 0.005 values.

```
def focal_loss(gamma=2.0, alpha=0.25):
    def focal_loss_fixed(y_true, y_pred):
        y_pred = tf.keras.backend.clip(y_pred, tf.keras.backend.epsilon(),1 - tf.keras.backend.epsilon())
        ce_loss = -y_true * tf.keras.backend.log(y_pred)
        loss = alpha * tf.keras.backend.pow(1 - y_pred, gamma) * ce_loss
        return tf.keras.backend.sum(loss, axis=1)
    return focal_loss_fixed
[. . .]
model.compile(optimizer=Adam(), loss=focal_loss(gamma=2.0, alpha=0.25), metrics=['accuracy'])
```

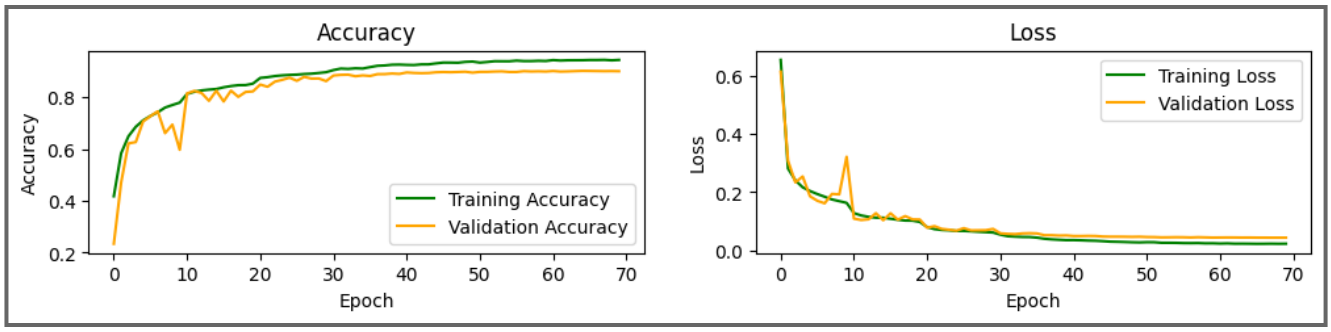
4. Performance Analysis

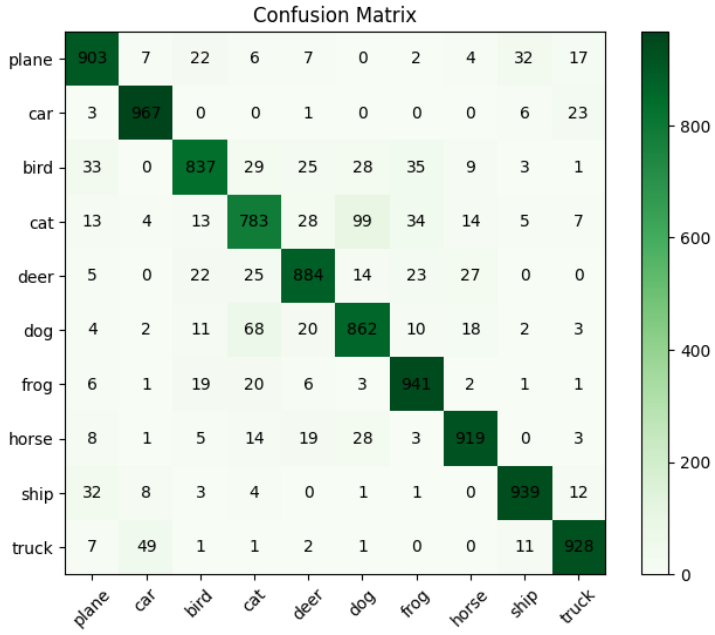
The model exhibits strong performance across training, validation, and test sets, though some overfitting is present. The training accuracy is notably high at 0.95 (having reached up to 0.98 during training) with a low loss of 0.02, while the validation accuracy is slightly lower at 0.91 and a higher loss of 0.05. The discrepancy suggests that the model's generalization to unseen data is slightly inferior; also supported by the test accuracy of 0.9 and loss of 0.05.

The precision, recall, and F1 Scores across individual classes are generally high, particularly for Cars and Ships, which have F1 Scores of 0.95 and 0.94, respectively. These results suggest that the model is stronger at distinguishing these classes with high accuracy. However, performance is lower for the Cats and Dogs, which have F1s of 0.80 and 0.85, indicating difficulties differentiating between them due to their similar features.

Overall, the weighted average precision, recall, and F1 Score are all 0.,9 reflecting a balanced model that performs consistently across most classes. However, the slight differences in per-class performance indicate the need for further fine-tuning or additional data augmentation to improve recognition on the harder classes.

Train Accuracy: 0.9482 **Loss:** 0.0241
Val Accuracy: 0.9059 **Loss:** 0.0460
Test Accuracy: 0.8963 **Loss:** 0.0489





	Precision	Recall	F1-Score	Support
Plane	0.89	0.90	0.90	1000
Car	0.93	0.97	0.95	1000
Bird	0.90	0.84	0.87	1000
Cat	0.82	0.78	0.80	1000
Deer	0.89	0.88	0.89	1000
Dog	0.83	0.86	0.85	1000
Frog	0.90	0.94	0.92	1000
Horse	0.93	0.92	0.92	1000
Ship	0.94	0.94	0.94	1000
Truck	0.93	0.93	0.93	1000
Accuracy			0.90	10000
Macro Avg	0.90	0.90	0.90	10000
Weighted Avg	0.90	0.90	0.90	10000

5. Transfer Learning Experiments

This section explains the different models we tested for Transfer Learning and why we ended up selecting our initial model instead. Models Tested: ResNet50, VGG16, and SqueezeNet.

a. ResNet50

Initially, ResNet50 struggled with performance, achieving training and validation accuracies around 0.5. Unfreezing the lower layers of the network did not lead to any improvements. However, when the upper layers were unfrozen, training accuracy significantly increased to 0.94, but this caused severe overfitting, with validation accuracy stagnating at 0.6. Attempts to mitigate overfitting through additional methods—such as unfreezing more layers, applying data augmentation, changing optimizers, adjusting learning rates, and adding dense layers—resulted in minimal improvements and failed to resolve the overfitting issue.

b. VGG16

VGG16 initially outperformed ResNet50, reaching an accuracy of approximately **0.7**. However, efforts to further improve the model by unfreezing layers, modifying the optimizer, fine-tuning the learning rate, applying data augmentation, and increasing the complexity of the dense layers led to negligible gains. The model consistently plateaued at a similar validation accuracy, suggesting that these modifications had limited impact on improving overall performance.

c. SqueezeNet

SqueezeNet emerged as the most promising model, despite early difficulties importing pre-trained weights. After implementing the architecture from scratch, the model achieved good initial results with a training accuracy above 0.8 and a validation accuracy of 0.77. Further fine-tuning—including reducing the learning rate, applying more aggressive data augmentation, and adjusting the learning rate reduction factor—helped reduce overfitting. However, despite improvements in training and validation curves, validation accuracy continued to plateau at around 0.7, suggesting challenges in pushing past this performance threshold.

d. Conclusion

Across all transfer learning models, unfreezing layers and adjusting hyperparameters had a significant impact on performance but highlighted the difficulty of achieving effective generalization, particularly with ResNet50. SqueezeNet demonstrated a more stable learning curve, but further improvements are needed to surpass the validation accuracy plateau. Ultimately, none of the transfer learning models were able to outperform our own custom model, which consistently achieved an accuracy of **0.9**. Thus, the custom model was selected as the best-performing solution.

6. Insights from Experimentation

We began this project with initial uncertainty regarding the most suitable models for CNN-based image classification. After conducting a thorough review of various architectures, we determined that the best approach was to develop our own models from scratch, beginning with simple, smaller architectures and progressively increasing complexity.

Throughout this process, we applied various optimization techniques, experimenting with layer adjustments, hyperparameter tuning, and other modifications. While some of those changes helped us move forward, we found an equal number of changes that left us without improvements or with going one (or more) steps back. Data augmentation was one of them, as we ended up testing different settings one by one, we could narrow it down to a couple of settings that were helping and discarded the rest. Which sometimes leads to terrible results.

The change in the optimizer proved to be crucial to our efforts, while the increase to more convoluted blocks or dense layers to our current model yielded no further improvements.
We achieved a quite robust model on our own and moving to transfer learning was more of a setback than a step forward. Initial transfer learning models did not provide an improvement and ResNet and VGG were clearly worse. Even after data augmentation and fine tuning, we could not find a way through. SqueezeNet seemed promising but could not find the fine-tuning adjustments needed to improve it.

Overall, our model seemed robust and although we don't reach over 0.95 like state-of-the-art models, reaching 0.9 on a relatively simple model was very satisfying. Comparing learning curves across models, confusion matrix and other metrics to see the improvements was quite a journey. A journey that brought us from a confused state in terms of knowing anything about CNN development to a state where we are comfortable looking at the code, playing with hyperparameters, optimizers etc. This also showed us there is plenty more to learn, but oh, how far we have come in so little time.