1. Project overview and objectives

This project aims to design and train a Convolutional Neural Network (CNN) to classify images from the CIFAR-10 dataset into 10 predefined categories.

2. Dataset and Preprocessing

2.1. The CIFAR-10 dataset

We had to make a choice and use either the CIFAR-10 dataset and another one that contained images with better resolution and consisted of 10 categories of animals.

The CIFAR-10 dataset was chosen over the other for the following reasons:

1. **Balanced Classes**: It contains an equal number of images per class, simplifying training.
2. **Small Image Size**: This makes it ideal for fast experimentation and evaluating CNN architectures, as well as for fast prototyping and feedback on model performance.
3. **Benchmark Status**: Widely used in research, allowing comparisons between different architectures.

It has some limitations, like its limited image resolution, which can affect the performance due to its limited feature details. It has generic classes which reduces its real world application, but as the purpose of this project is learning, we wanted to use a dataset that could speed up the processes and allow us to gather more learnings and insights.

2.2. Image resizing and normalization techniques

Before feeding the data into the CNN, we normalized the pixel values of the CIFAR-10 images, which originally range from **0 to 255**. High pixel values can cause **unstable gradients** and slow learning during training. To prevent this, we scaled the pixel values to a **[0, 1] range** by dividing them by **255**, which improves both the stability and speed of the model's convergence.

1. Converted the image data to float32 using .astype('float32').
2. Scaled pixel values by dividing by 255.0.

This normalization step enhances the model's ability to learn from the input images, leading to more efficient training.

2.3. One hot encoding

We also applied **one-hot encoding** to the class labels using to_categorical(). This converts the integer labels (0–9) into binary vectors, where each vector has a length of 10 with a single "1" representing the correct class and the rest being "0".

This normalization step enhances the model's ability to learn from the input images, leading to more efficient training.

2.4. Data augmentation strategies

For the CIFAR-10 dataset, we used **rotation_range=15**, **width_shift_range=0.1**, and **height_shift_range=0.1** to introduce variability while preserving important image features. Given the small size of CIFAR-10 images (32x32 pixels), more aggressive augmentations (like rotations of 20 degrees or shifts of 0.2) could distort the images and degrade performance.

By using these moderate augmentations, we maintained a balance between adding diversity to the dataset and preserving the integrity of the images for effective learning. This measure also prevents the model from overfit.

3. Model 1

3.1 Initial Research

We decided to use the Sequential model from Keras. It's a simple and intuitive way to stack layers in a **linear** manner, which makes sense for building CNNs like the one in our CIFAR-10 example. There are other approaches that might be more flexible, depending on the complexity of the architecture. If our model would have been more complex and needed **share layers** or would involve **multiple inputs or outputs** we could have used another model**.**

3.1.1. Architecture

With the goal of learning in mind, we first developed our own architecture. This initial design included three convolutional layers with **32 filters**, a **3x3 kernel size**, and **2x2 max pooling** after each convolutional layer. We also used two Dense layers.

After testing, we iterated and shifted to a model inspired by the **VGG16 architecture**, which has a proven track record for CIFAR-10 performance. This decision was further supported by Carlo´s recommendations.

3.1.2. Layers

Each convolutional layer focuses on a different feature of the image. Given their simplicity, we started with four convolutional layers to evaluate how performance evolves as more layers are added, allowing us to iteratively improve the model. Additionally, we incorporated **MaxPooling** to reduce the spatial dimensions of the image, helping to decrease computational complexity.

3.1.3. Metrics

- **Accuracy:** allows us to see how many images were classified correctly out of all and works best with balanced classification tasks like CIFAR-10.
- **Precision and Recall:** to check false positives and true negatives, like incorrectly classifying a dog as a cat or missing classifying a cat as such.
- **F1-Score:** balances both precision and recall, very useful for multi-class classification problems where class distributions may be imbalanced.

3.1.4. Activation functions

After comparing several activation functions and researching their effectiveness with the CIFAR-10 dataset, we decided to use the **ReLU** activation function. We also experimented with alternatives like **Leaky ReLU**, which was intended to provide better performance by addressing issues such as the dying ReLU problem. However, our experiments showed that **ReLU** consistently yielded better results for our model, making it the most suitable choice for our specific application.

3.1.5. Loss functions

After comparing **categorical cross-entropy** and **sparse categorical cross-entropy**, we chose categorical cross-entropy. It is specifically designed for multi-class classification and works seamlessly with **one-hot encoded labels**, aligning with our preprocessing approach.

3.1.6. Optimizers

After evaluating several optimization algorithms and researching their performance with the CIFAR-10 dataset, we chose to use the **Adam** optimizer. We also experimented with alternatives such as **Stochastic Gradient Descent (SGD)** and **RMSProp**, aiming to potentially achieve better convergence and performance. However, our experiments demonstrated that **Adam** provided superior results, leading to faster convergence and higher accuracy for our model.

3.2. Optimizations

3.2.1. Architecture

New layers: Initially, we experimented by adding up to two additional convolutional layers. However, this didn't improve performance, so we reverted to using three layers, though we later realized this was a mistake.

Filters: Increasing the number of filters across the three layers (32, 64, 128) caused the loss to spike, suggesting the model became too complex.

Batch normalization: Adding batch normalization after each convolutional layer led to a small but noticeable improvement in all metrics.

3.2.2. Hyperparameters

Learning rate: We manually tested values from 0.001 to 0.01, but eventually settled on using a learning rate scheduler, which showed the most improvement, though the gain was marginal.

Batch size: Reducing the batch size from 512 to 256 provided a small improvement in performance.

Epochs: After experimenting with increasing and decreasing the number of epochs, we decided to stick with 50 epochs, as further adjustments either had no effect or worsened the results.

3.2.3. Optimizers and regularization methods

Dropout: Adding dropout significantly improved performance. We initially tested with 50%, but eventually settled on 30%, which turned out to be the most effective optimization, bringing the greatest improvement in results.

Optimizers: We experimented with Leaky ReLU as an alternative activation function, but it did not improve performance. We decided to stick with the original ReLU activation, which yielded better results.

3.3.Results and performance for the first model (check presentation for graphs)

3.3.1. Metrics

Overfitting: Higher **train accuracy (82.45%)** vs. **test accuracy (74.15%)** suggests the model overfits on the training data.

Precision vs. Recall (Train): **High precision (90.41%)** but lower **recall (73.15%)** shows the model makes accurate predictions but misses many true positives.

Test Precision and Recall: **Test precision (82.98%)** is decent, but **recall (65.17%)** is lower, meaning the model misses more true positives in unseen data.

F1-Scores: The **F1-score (80.71% train, 72.81% test)** reflects a balance between precision and recall, but the drop indicates performance loss on new data.

3.3.2. Confusion matrix

Good Performance: Most predictions are correct, especially for classes "0" (810) and "1" (860).

Key Misclassifications: Class "2" is often confused with "3" (73 times), and class "9" with "1" (98 times).

Uneven Errors: Some classes (like "5" and "6") show fewer mistakes, while "2" and "9" have more.

Similar Classes: Misclassifications are common among visually similar classes (e.g., "2", "3", "4"), indicating areas for improvement.

4. Model 2

4.1.New Architecture

We began by replicating partially the **VGG16 architecture**, which involved using **12 convolutional layers**, a significant increase compared to the 3 layers we had used previously. The architecture became more complex as we progressively increased the number of filters in each batch of layers. For the sets with **128, 256, and 512 filters**, we added **three convolutional layers** instead of two. This modification resulted in noticeable improvements in **test accuracy**, **precision**, and **recall**.

4.2. Optimization

4.2.1. *Architecture*

Layers: We attempted to optimize the number of layers but found that our original approach yielded the best results.

Batch normalization: VGG16 model didn´t have Batch normalization and we tried removing it, which decreased substantially the performance.

4.2.2. Hyperparameters

Learning rate: We manually tested learning rates from 0.001 to 0.01, but ultimately stuck with the learning rate scheduler, which provided the best improvement, though it was not significant..

Batch Size: Increasing the batch size to **400** proved to be the optimal choice for this model, resulting in improved performance.

Epochs: Reducing the number of epochs from **50 to 35** was the adjustment that led to better results for this model.

4.2.3. Optimizers and regularization methods

Dropout: Adding **50% dropout** significantly improved performance, and this value turned out to be the optimal choice for the model.

4.3. Results a

nd performance of the second model (check presentation for graphs)

4.3.1. Metrics

High Training Accuracy: The model performs exceptionally well on the training set with 98% accuracy and a high F1-score of 0.98, indicating effective learning.

Overfitting Signs: Despite strong training metrics, the test performance drops significantly with 77.6% accuracy and a high test loss (1.26). This suggests overfitting, as the model struggles to generalize to unseen data.

Moderate Test Precision/Recall: Both test precision (0.78) and recall (0.77) show the model's ability to predict with a balance between true positives and false positives, but far less effectively than on training data.

4.3.2. Confusion matrix

Class 1 (Label 1) is predicted quite accurately with 899 correct classifications and only a few misclassifications, suggesting the model is good at identifying this class.

Classes 3 and 4 (Labels 3 & 4) show significant confusion between each other. For example, class 4 has a substantial number of its samples classified as class 3 and vice versa, indicating that the model struggles to differentiate between these two classes.

Class 2 (Label 2) has a relatively high number of incorrect predictions, with many of its samples being misclassified into classes 4, 3, and 6. This suggests that features for these classes might be similar or the model is not able to capture distinguishing features effectively.

Class 6 (Label 6) shows very good performance, with 856 correct predictions and relatively few misclassifications, indicating strong performance on this class.

5. Transfer learning

5.1. Resnet50

In our transfer learning experiments for the CIFAR-10 classification task, we initially utilized the **ResNet50** model due to its advanced architecture and proven effectiveness in image recognition. ResNet50's deep layers and residual connections enabled robust feature extraction, leading to high accuracy and reliable performance.

5.2 VGG16

Seeking to explore alternative architectures, we subsequently experimented with **VGG16**. However, our findings indicated that VGG16 did not achieve results as favorable as those obtained with ResNet50. Consequently, we prioritized ResNet50 for our final model, as it consistently delivered superior accuracy and better overall performance for our classification objectives.

6. Insights

6.1. Challenges faced during training

We encountered frequent kernel crashes in Paperspace, which made it impossible to properly analyze the performance of both transfer learning models, therefore limiting the optimization we could implement.

6.2. Learnings

For the first day and a half we only checked the test data, which lead us to miss signs of **overfitting** and also we couldn’t check **how the models were learning**. A huge learning: checkin both training and test data is **essential** to understand whether the model is learning well and generalizing effectively.

We forgot to add a **validation** set in our first models, therefore we couldn’t run it in the transfer learning, which is fundamental for building robust, generalizable models and ensuring effective transfer learning.

We faced a **TensorFlow version mismatch** during our transfer learning project due to the code being written for an older version, conflicting with the newer version in Paperspace. We were blocked here for a while, until with Isabella's help, we downgraded TensorFlow in Paperspace to match our code's version, resolving the compatibility issues. Another **learning**: This experience underscored the importance of ensuring software compatibility in machine learning projects and highlighted the value of collaboration in problem-solving.