Project Report: CIFAR-10 Image Classification

February 7, 2025

Contents

List of Tables

List of Figures

1 Introduction

The objective of this project is to classify images from the CIFAR-10 dataset utilizing neural networks. Data preprocessing, exploratory data analysis, model building, and performance evaluation are conducted.

2 Data Preparation

2.1 Data Acquisition

The CIFAR-10 dataset consists of 50,000 32x32 color images across 10 classes, encompassing objects such as airplanes, automobiles, birds, and more.

2.2 Preprocessing

Data normalization scales pixel values to a range between 0 and 1. Additionally, data augmentation techniques such as rotation, scaling, and flipping are applied to improve model generalization. `ImageDataDenerator` was used for augmentation to increase the diversity of data.

- **Implementation of normalization:**

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

- **Implementation of augmentation:**

```
from tensorflow.keras.preprocessing.image import
    ImageDataGenerator

datagen = ImageDataGenerator(
rotation_range=20,
width_shift_range=0.2,
height_shift_range=0.2,
horizontal_flip=True,
zoom_range=0.2,
shear_range=0.15

datagen.fit(x_train)
```

3 Exploratory Data Analysis (EDA)

3.1 Class Distribution Analysis

The distribution of images across classes is examined to ensure a balanced representation. The visualizations below confirm that the classes are evenly represented, i.e., each of the 10 classes has 5000 images:

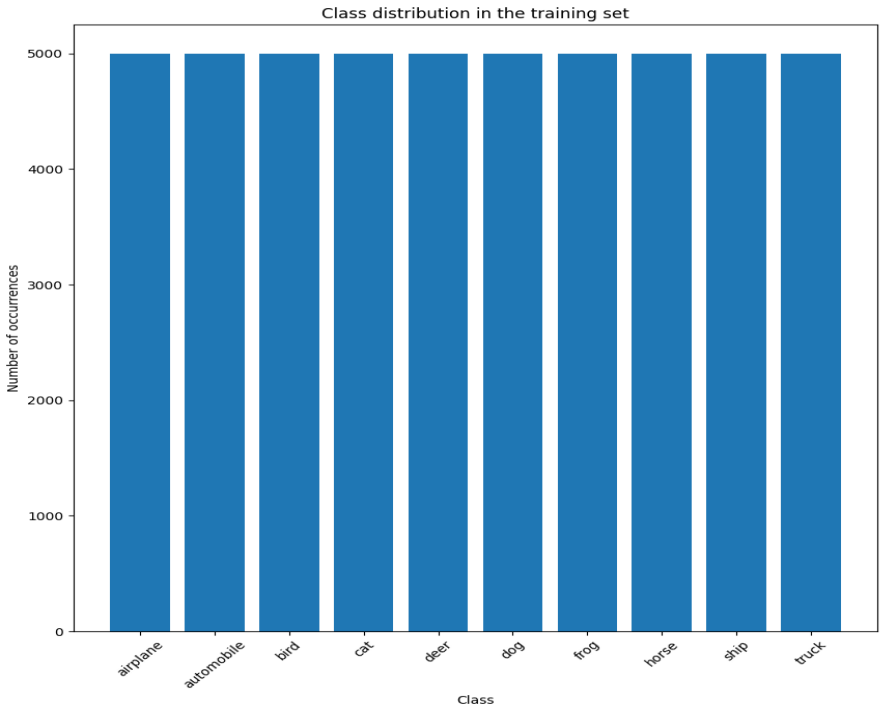

Figure 1: Class distribution in the training set.

3.2 Variety of images in each class

Displaying a few random images for half of the classes to represent how the images differ from each other:

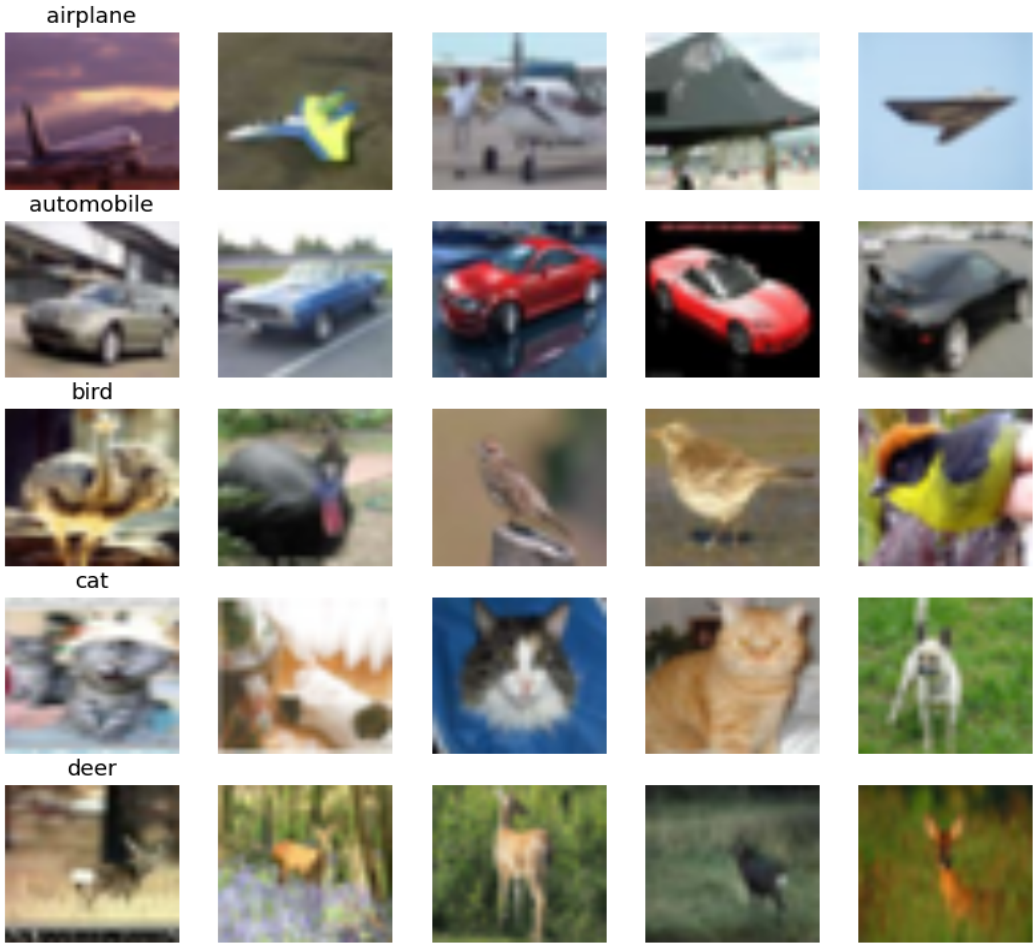

Figure 2: Variety of images in each classt.

3.3 Image Feature Analysis

Key features of the images, including geometry and color patterns, are analyzed to identify any potential data biases or anomalies. To this end, mean pixel values and standard deviations were calculated for the entire dataset as well as for each individual class. This analysis provides insights into the overall brightness and contrast variations among different classes, and highlights any significant disparities that might exist. Such metrics are crucial for ensuring that preprocessing techniques like normalization are effectively tailored to the specific characteristics of the dataset.
The table below shows the results:

Class [id]	Average pixel value	Standard deviation
airplane [0]	0.64153	0.25299
automobile [1]	0.54890	0.31109
bird [2]	0.49409	0.25678
cat [3]	0.46323	0.28865
deer [4]	0.42994	0.25072
dog [5]	0.42244	0.26808
frog [6]	0.42081	0.26383
horse [7]	0.54750	0.30047
ship [8]	0.64741	0.24978
truck [9]	0.69712	0.27628
Overall	[0.49140, 0.48216, 0.44653]	[0.24703, 0.24349, 0.26159]

Table 1: Average pixel values and standard deviation for each class and overall.

The analysis of the mean pixel values and standard deviations for the classes of the CIFAR-10 images reveals interesting differences in intensity and color variation across the classes. Below is the detailed interpretation:

1. **Mean Pixel Values:**
    - The highest mean pixel values are observed in Class 9 (0.697) and Class 8 (0.647), suggesting that images in these classes have a brighter tone.
    - The lowest mean values are found in Class 6 (0.421) and Class 5 (0.422), indicating darker tones in these images.
2. **Standard Deviation:**
    - The greatest color variation, measured by standard deviation, is present in Class 1 (0.311) and Class 7 (0.300), indicating a high variability in pixel intensity within each of these classes.
    - The smallest variation is found in Class 8 (0.250) and Class 4 (0.251), where pixel values are more uniform.

3. **Overall Means and Deviations:**
    - The overall mean pixel values for the entire dataset are [0.491, 0.482, 0.447], suggesting a prevalence of moderate tones and a slight dominance of darker shades in the dataset.
    - The overall standard deviations are [0.247, 0.243, 0.261], confirming a relatively even distribution of color intensity across different channels, with slightly higher variability in the blue channel.

Based on these results, each class in the CIFAR-10 dataset exhibits unique characteristics in terms of average brightness and color variability. Such analysis is particularly useful for understanding tonal range and color dynamics in image processing tasks, which can be utilized in the design and training of machine learning models.

4 Modeling

This section covers the creation of the model and preparation for training by tuning hyperparameters.

4.1 Model Architecture

The architecture of a `Convolutional Neural Network (CNN)` includes Convolutional, Pooling, Dropout, Flatten and Dense layers, optimized for image data processing. **Below is an example implementation of a CNN:**

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Flatten, Dense, Dropout

model = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='
       relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.2),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='
   sparse_categorical_crossentropy', metrics=['accuracy'
   ])
```

Explanation of the selection of individual parameters:

- `filters = 32` - the selection of 32 filters in the initial layers of a convolutional neural network (CNN) is a common starting point because it effectively captures fundamental features such as edges and colors while balancing computational complexity and performance.
- Activation Function: `ReLU (Rectified Linear Unit)` - was chosen as the activation function for Conv2D layers due to its computational efficiency, its ability to mitigate the vanishing gradient problem, and its promotion of sparsity, leading to faster training and more efficient model performance.
- Pooling Layer `MaxPooling2D` - used to accentuate the most expressive features, ignoring the less important ones and additionally reducing the risk of overfitting.
- Layer `Dropout` - used to increase model randomness and prevent overfitting.
- Layer `Flatten` - transforming the input data from a multidimensional matrix into a one-dimensional vector, which is required at the input of the `Dense` layer.
- Layers `Dense` - the first layer `Dense(128, activation='relu')` is used to process and generalize the patterns captured by the earlier layers of the network, and the last layer `Dense(10, activation='softmax')` transforms these patterns into the final classifications, giving probabilities for each of the 10 classes in `CIFAR-10`, summing them to 1 using `softmax`.

Then the model is compiled using:

- Optimizer `Adam (Adaptive Moment Estimation)` - an adaptive optimization method that uses momentum to accelerate gradient descent and achieve fast convergence.
- Loss Function `Sparse Categorical Crossentropy` - used for multiclass classification.
- Metric `Accuracy` - a metric for model evaluation, measuring what % of predictions are correct relative to the test or validation data labels.

4.2 Hyperparameter Tuning

The hyperparameter tuning process utilized the `Keras Tuner` library with `Random Search` to find the optimal model hyperparameters. The implementation is shown below:

```
from kerastuner import HyperParameters
from kerastuner.tuners import RandomSearch
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Flatten, Dense, Dropout

def build_model(hp):

    model = Sequential()
    for i in range(hp.Int('conv_layers', 1, 3)):
        model.add(Conv2D(
            filters=hp.Int('filters_' + str(i), min_value
               =32, max_value=128, step=32),
            kernel_size=(3, 3), activation='relu',
               input_shape=(32, 32, 3) if i == 0 else
               None))
        model.add(MaxPooling2D((2, 2)))
        model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(units=hp.Int('units', min_value=64,
       max_value=512, step=64),
                                    activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    model.compile(
            optimizer=hp.Choice('optimizer', ['adam', '
               rmsprop', 'sgd']),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

    return model

# hyperparameter tuning
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
```

```
    max_trials=3,
    executions_per_trial=2,
    directory='my_dir',
    project_name='my_project_enhanced' )

tuner.search(x_train, y_train, epochs=3, validation_split
    =0.1)
```

Experiments were conducted with various neural network configurations, including:

- different numbers of convolutional layers (ranging from 1 to 3).
- numbers of filters per convolutional layer (ranging from 32 to 128).
- units in the dense layer (ranging from 64 to 512).
- different optimizers were tested, including: `adam`, `rmsprop`, and `sgd`.

Each set of hyperparameters was evaluated based on validation accuracy using 10% of the data as a validation set. The optimized model was selected according to the `val_accuracy` metric. To reduce tuning time, the `max_trials` was limited to 3, with 2 `executions_per_trial`, and the total number of `epochs` was set to 3.

The result of hyperparameter tuning is shown below:

- `Best val_accuracy So Far: 0.6162000000476837`.
- `Total elapsed time: 00h 06m 24s`.

4.3 Finding and building the best model

After tuning the hyperparameters, the configurations with the most optimal hyperparameters were found and based on them a model was built ready for further training:

```
best_hps = tuner.get_best_hyperparameters(num_trials=1)
    [0]

best_conv_layers = best_hps.get('conv_layers')
print(f"Optimal number of convolutional layers: {
    best_conv_layers}")

for i in range(best_conv_layers):
```

```
        filters = best_hps.get(f'filters_{i}')
        print(f"Optimal number of filters for a layer {i}: {
            filters}")

print(f"Optimal number of units: {best_hps.get('units')}"
    )
print(f"The most effective optimizer: {best_hps.get('
    optimizer')}")
```

Hyperparameter configuration that was obtained based on empirical tests:

- Optimal number of convolutional layers: 2.
- Optimal number of filters for a layer 0: 64.
- Optimal number of filters for a layer 1: 32.
- Optimal number of units: 128.
- The most effective optimizer: rmsprop.

5 Model Training

In the next step, the model with the included hyperparameters was trained on the entire training set, taking 10% of the data as a validation set:

```
history = model.fit(x_train, y_train, epochs=20,
    batch_size=64, validation_split=0.1)
```

5.1 Training results

The training results are presented below:

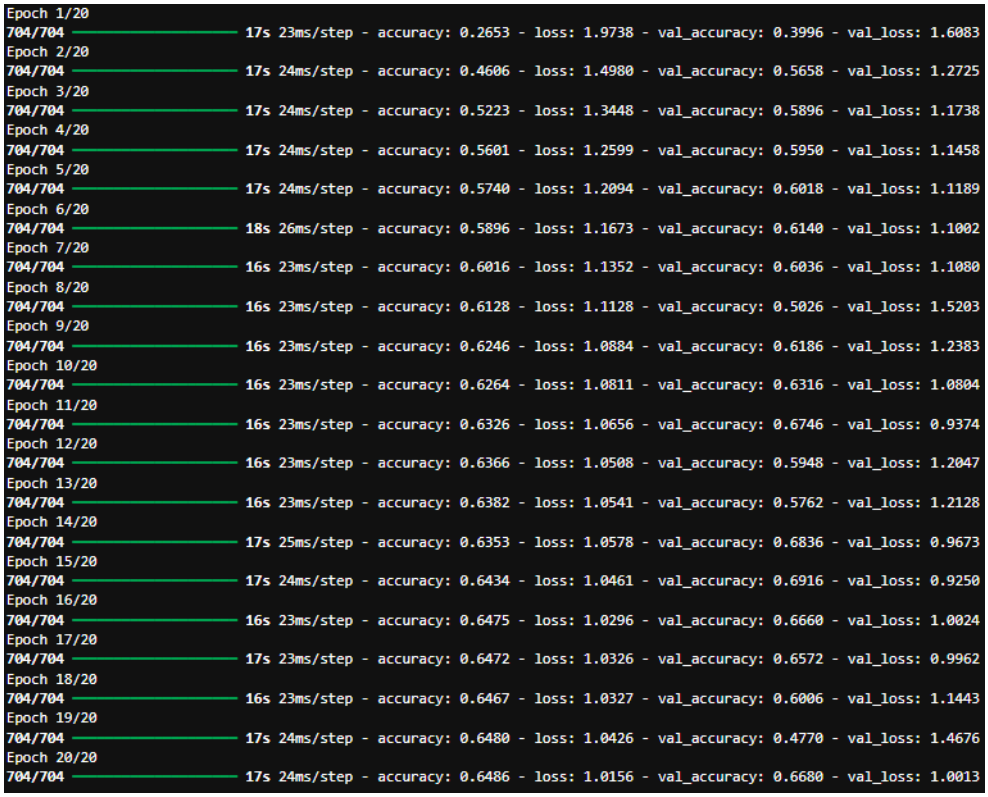

Figure 3: CNN model training results.

5.2 Conclusions and Recommendations:

1. **Convergence of Training Metrics:**

   As the epochs progressed, the training accuracy steadily increased while the training loss decreased. This indicates that the model is learning and improving its performance on the training data, which is a sign of proper convergence.

2. **Validation Performance:**

   The validation accuracy generally improved over the first few epochs, reaching higher values in later epochs, although there were fluctuations. The best validation accuracy was achieved around epochs 14 and 15, suggesting that the model was able to generalize reasonably well to the validation data.

3. **Fluctuations in Validation Loss:**

   There are some fluctuations observed in the validation loss, which could be due to overfitting or the validation set not perfectly capturing the model's performance. It's important to monitor this trend to see if additional regularization or early stopping might benefit the model's performance in future iterations.

In summary, the model seems to be training correctly, with signs of convergence as indicated by increasing accuracy and decreasing loss over time. However, monitoring for overfitting by examining validation metrics can further refine its performance.

6 Model Evaluation and Prediction

After training the model, it's time to evaluate its performance on the test set, consisting of data that the model has not encountered before.

6.1 Accuracy of the model during evaluation

Main `accuracy` metric tested on test set:

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy}")
```

and **the following result was obtained:**

- `Test accuracy: 0.6575999855995178`.

6.2 Other metrics achieved based on model prediction

To calculate other metrics it uses model prediction and functions from the `scikit learn library`:

```
from sklearn.metrics import precision_score, recall_score
    , f1_score, accuracy_score
import numpy as np

y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)

precision = precision_score(y_test, y_pred_classes,
    average='macro')
```

```
recall = recall_score(y_test, y_pred_classes, average='
    macro')
f1 = f1_score(y_test, y_pred_classes, average='macro')

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

The results of the individual metrics are presented below:

- `Precision: 0.6890180949955423`.
- `Recall: 0.6576000000000001`.
- `F1 Score: 0.6649735856295848`.

6.3 Final metrics results

The table below presents the final results of the metrics that describe the characteristics and effectiveness of the trained model (values rounded to 3 decimal places):

Metric	Value
Accuracy	0.657
Precision	0.689
Recall	0.658
F1 Score	0.665

Table 2: Summary of metrics results for the trained CNN model.

The following metrics evaluate the performance of the CNN (Convolutional Neural Network) model:

- **Accuracy:** 0.657
  This means that the model correctly predicted outcomes in approximately 65.7% of cases. This is not a high result, indicating room for improvement. However, note that accuracy can be misleading if the data is imbalanced (e.g., when one category is much more frequent).
- **Precision:** 0.689
  Precision indicates how many of the positive predictions were actually correct. Here, it is 68.9%. High precision implies that the model rarely makes mistakes when it claims something belongs to a given category.

- **Recall:** 0.658
  Recall measures how well the model captures all actual positive cases. Here, it is 65.8%. Recall is crucial when it is important to detect all relevant cases, even at the cost of more false alarms.

- **F1 Score:** 0.665
  The F1 score is a combination of precision and recall. A result of 66.5% suggests that the model tries to maintain a balance between these two metrics. This is useful when aiming to balance detecting as many instances as possible with ensuring prediction accuracy.

How to Improve These Metrics:

- **More data:** More training data often helps the model to learn better, improving all metrics.

- **Model adjustment:** Different network architectures can be tried, or the existing one can be deepened or modified.

- **Improving data quality:** Removing noise and ensuring cleaner data can enhance results.

- **Data balancing:** Ensure all categories are represented in similar proportions.

- **Parameter optimization:** Adjusting model parameters (e.g., learning rate) can affect results.

To find the best configurations, experiments should be conducted, and an iterative approach should be taken.

6.4 Confusion Matrix Insights

The analysis of the confusion matrix reveals misclassified classes, indicating areas for potential improvement in data handling or model architecture. The obtained confusion matrix is presented below, which **highlights two classes: `dog` and `deer`, for which the model has the greatest problems with correct prediction:**

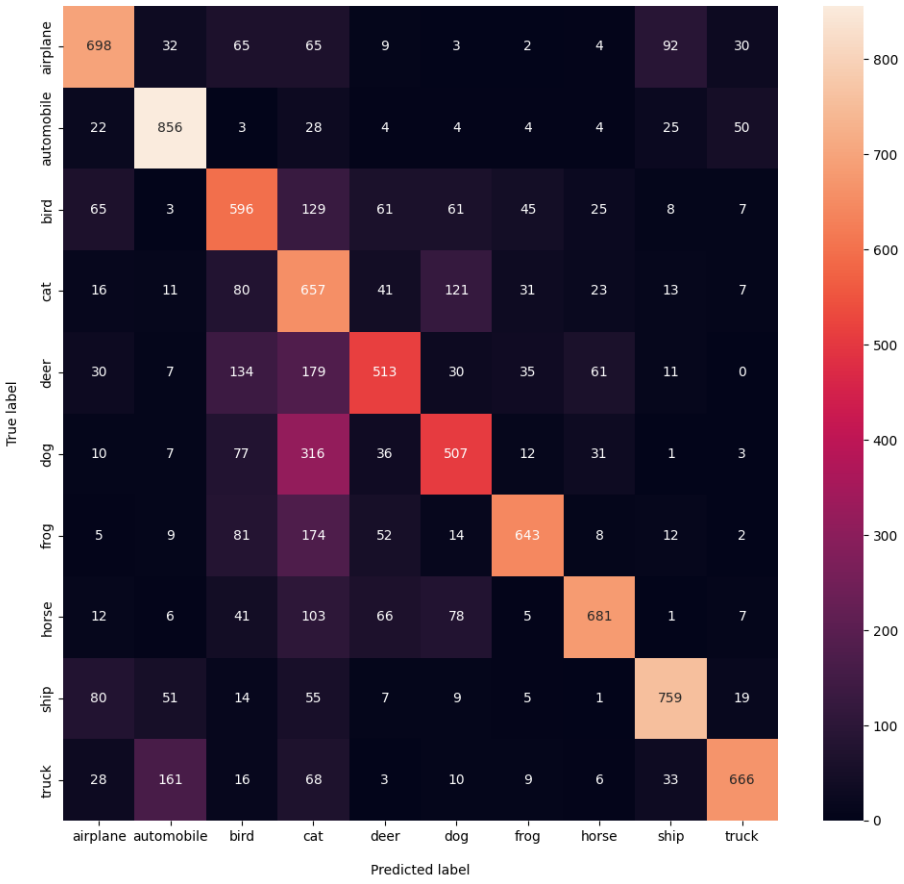

Figure 4: Obtained confusion matrix.

Interpretation of results on the example of a class: "Deer"

1. Confusion matrix results:
   - **Total Cases:** 1000
   - **Total Predictions:** 792
   - **Correct Predictions:** 513
2. Metrics results:
   - **Accuracy:** 65.76%
   - **Precision:** 64.77%

- **Recall:** 51.3%
- **F1 Score:** 57.3%

3. Recommendations for Improvement:

   - **Data Augmentation:** Apply techniques such as rotation, scaling, and flipping to increase the diversity of training samples, which might help the model generalize better.
   - **Class Rebalancing:** Consider oversampling the "deer" class or undersampling more prevalent classes to address any class imbalance.
   - **Model Architecture Tuning:** Experiment with different CNN architectures or deeper layers to better capture complex features of the "deer" class.
   - **Advanced Regularization:** Utilize techniques like dropout or batch normalization to prevent overfitting and improve model generalization.
   - **Transfer Learning:** Leverage pre-trained models on similar datasets and fine-tune them for `CIFAR-10` to improve performance.

Implementing these strategies may increase both the recall and precision, leading to a more balanced and effective model for detecting the "deer" class and the other classes.

7 K-Fold Cross Validation

`K-fold Cross Validation` was performed to assess model robustness and ensure performance is consistent across different data partitions. The number of 5 folds was chosen due to the large amount of data: 50000. Below are the cross-validation implementations:

```
from sklearn.model_selection import KFold
from sklearn.metrics import precision_score, recall_score
    , f1_score
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
import numpy as np

accuracy_scores = []
precision_scores = []
recall_scores = []
```

```
f1_scores = []

kfold = KFold(n_splits=5, shuffle=True)

for train_idx, val_idx in kfold.split(x_train):

    model = load_model('best_model_NEW.h5')

    optimizer = Adam()

    model.compile(optimizer=optimizer, loss='
       sparse_categorical_crossentropy', metrics=['
       accuracy'])

    x_train_fold, x_val_fold = x_train[train_idx],
       x_train[val_idx]
    y_train_fold, y_val_fold = y_train[train_idx],
       y_train[val_idx]

    history = model.fit(datagen.flow(x_train_fold,
       y_train_fold, batch_size=64),
                        epochs=5,
                        validation_data=(x_val_fold,
                           y_val_fold),
                        verbose=1)

    val_accuracy = history.history['val_accuracy'][-1]
    accuracy_scores.append(val_accuracy)

    y_val_pred = model.predict(x_val_fold)
    y_val_pred_classes = np.argmax(y_val_pred, axis=1)

    precision = precision_score(y_val_fold,
       y_val_pred_classes, average='macro')
    recall = recall_score(y_val_fold, y_val_pred_classes,
        average='macro')
    f1 = f1_score(y_val_fold, y_val_pred_classes, average
       ='macro')

    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)
```

```
average_accuracy = np.mean(accuracy_scores)
average_precision = np.mean(precision_scores)
average_recall = np.mean(recall_scores)
average_f1 = np.mean(f1_scores)
```

Then averaged metrics were calculated for all folds:

Metric	Value
Average Accuracy	0.656
Average Precision	0.681
Average Recall	0.656
Average F1 Score	0.644

Table 3: Summary of average metrics after K-Fold Cross Validation.

Comparing the results with table2, which contains the metrics after first the model evaluation, very small differences can be seen, which confirms the fact that the model is consistent and works stably on different data fragments.

8 Conclusions and Future Work

The project aimed at classifying images from the `CIFAR-10` dataset using convolutional neural networks (CNNs). The stages of data preprocessing, exploratory data analysis, model architecture design, and performance evaluation were detailed in this report. Key achievements and observations include:

- **Data Preparation and Analysis:** Effective data preprocessing techniques, such as normalization and augmentation, were implemented to enhance the diversity and quality of the dataset. Exploratory data analysis provided insights into class distributions and image feature statistics, ensuring well-understood and balanced inputs for model training.
- **Model Architecture and Training:** A CNN was constructed with layers specifically designed for image data, leading to a model that effectively learned features across different classes. Hyperparameter tuning was performed using the `Keras Tuner` library, resulting in an optimized model configuration achieving a test accuracy of 65.7%.
- **Performance Metrics:** The model's performance was evaluated using several metrics, including precision, recall, and F1 score. The results

indicate a moderate level of classification accuracy, with opportunities for further enhancement, particularly in improving class-specific predictions such as for the "dog" and "deer" categories.

- **Model Evaluation and Robustness:** K-fold cross-validation confirmed the consistency and stability of the model's performance across different data partitions, indicating reliable generalization capabilities on unseen data.

Future Directions: Several potential improvements have been identified for future work:

1. **Enhanced Model Architectures:** Exploring deeper or more complex CNN architectures, such as `ResNet` or `Inception`, may capture more intricate patterns and improve accuracy.

2. **Advanced Techniques:** Implementing transfer learning by leveraging pre-trained models could significantly boost performance metrics.

3. **Refining Data Strategies:** Further efforts in data augmentation and class rebalancing could help mitigate issues observed in class-specific predictions, particularly for underrepresented classes.

4. **Regularization Methods:** Incorporating advanced regularization techniques like batch normalization or additional dropout layers can assist in managing overfitting and enhancing model robustness.

The findings of this project offer a foundation for continued exploration and innovation in the domain of image classification, with a focus on optimizing CNN methodologies for improved precision and recall.