IMAGE CLASSIFICATION PROJECT

20/05/2024
CSY3025 – Artificial Intelligence Techniques
Sergiu Popa

Table of Figures..

Table of Contents ....

NOTES:

Kaggle Dataset Link
https://www.kaggle.com/datasets/jessicali9530/stanford-cars-dataset
Files anno_test, anno.csv, train.csv, names.csv must be present for the code to work.
Folder structure: data/train, data/test

1. Introduction

Image classification involves categorizing images into predefined classes, a fundamental task in computer vision with applications spanning from autonomous vehicles and medical diagnosis to facial recognition and automated surveillance. The ability to classify images accurately enhances decision-making processes and operational efficiency in various industries.

The primary objectives of this project are:

- Dataset Selection: Choose a suitable dataset for the image classification task.
- Data Preparation: Normalize the data, splitting the images in train, val and test sets
- Data Preprocessing: Resizing images and normalizing pixel values, augmenting data,
- Model Development: Design and train multiple models, including a basic CNN, MobileNetV2, and ResNet50, to classify images.
- Performance Evaluation: Compare the performance of the developed models and analyze their effectiveness.

2. Methodology

2.1. Dataset Selection

The dataset selected for this project is the Stanford Cars dataset, which contains 16,185 images of cars categorized into 196 classes. Each class represents a different car model, with a balanced distribution between training (8,144 images) and testing (8,041 images) sets. The images are of varying sizes, representing cars from multiple angles and under different lighting conditions, providing a rich and challenging dataset for image classification tasks.
Rationale for Dataset Choice:

The Stanford Cars dataset was chosen for several reasons:

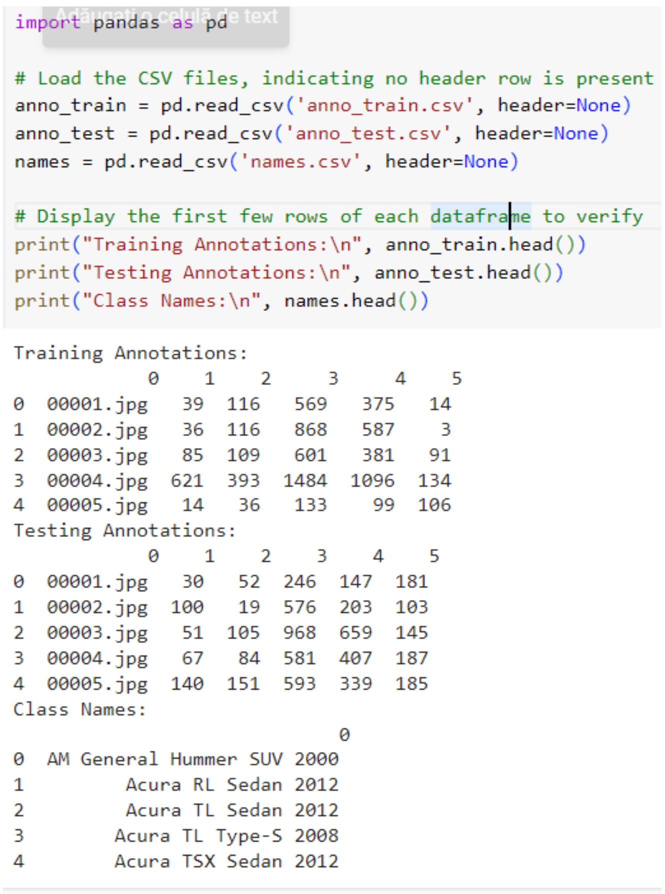
It’s relevance, the dataset's focus on vehicles aligns well with common real-world applications of image classification, such as automated vehicle identification and traffic monitoring.
Complexity of the data, including a consistent amount of picture plus the variety of car models and the high number of classes make it a suitable challenge for testing different neural network architectures. Detailed annotations including make, model, and year of each car enhance the dataset's utility for supervised learning tasks.
The dataset is a popular benchmark in academic research, providing a comprehensive introduction to neural networks and transfer learning.
The dataset's complexity ensures a rigorous evaluation of model performance, but it also presents challenges such as distinguishing between similar car models and dealing with varying image quality and backgrounds. Despite these challenges, the dataset's detailed annotations and wide acceptance in the research community make it an ideal choice for this project.

2.2. Data Preparation

To prepare the dataset for training, a series of preprocessing steps were undertaken to ensure the data's quality and compatibility with the neural network models. The preparation steps were chosen to ensure the dataset is well-organized, consistent, and ready for training. Renaming and updating the CSV files ensure there are no mismatches between image filenames and annotations. Merging images and annotations simplifies the training process by consolidating all data in a single location. Adding detailed annotations helps in better understanding and utilizing the dataset for training the models. These methods collectively contribute to improving the model's performance by providing clean, well-structured, and enriched data.

Loading and Handling CSV Files

The Stanford Cars dataset includes annotations and class names in CSV files that lack header information. and pandas is misinterpreting the first row of data as headers. This is a common issue when loading CSV files that lack a header row. This necessitated the explicit handling of headers during the loading process.



```
import pandas as pd

# Load the CSV files, indicating no header row is present
anno_train = pd.read_csv('anno_train.csv', header=None)
anno_test = pd.read_csv('anno_test.csv', header=None)
names = pd.read_csv('names.csv', header=None)

# Display the first few rows of each dataframe to verify
print("Training Annotations:\n", anno_train.head())
print("Testing Annotations:\n", anno_test.head())
print("Class Names:\n", names.head())
```

```
Training Annotations:
            0    1    2     3     4    5
0  00001.jpg   39  116   569   375   14
1  00002.jpg   36  116   868   587    3
2  00003.jpg   85  109   601   381   91
3  00004.jpg  621  393  1484  1096  134
4  00005.jpg   14   36   133    99  106
Testing Annotations:
            0    1    2    3    4    5
0  00001.jpg   30   52  246  147  181
1  00002.jpg  100   19  576  203  103
2  00003.jpg   51  105  968  659  145
3  00004.jpg   67   84  581  407  187
4  00005.jpg  140  151  593  339  185
Class Names:
                          0
0  AM General Hummer SUV 2000
1         Acura RL Sedan 2012
2         Acura TL Sedan 2012
3        Acura TL Type-S 2008
4        Acura TSX Sedan 2012
```

Figure 1 CSV Load

Renaming Images

To standardize the image filenames for easier management and to reflect changes in the annotations, the images in both the training and testing directories were renamed sequentially. This step ensures that there are no filename conflicts and that the images are consistently indexed.

Merging Images and CSV Files

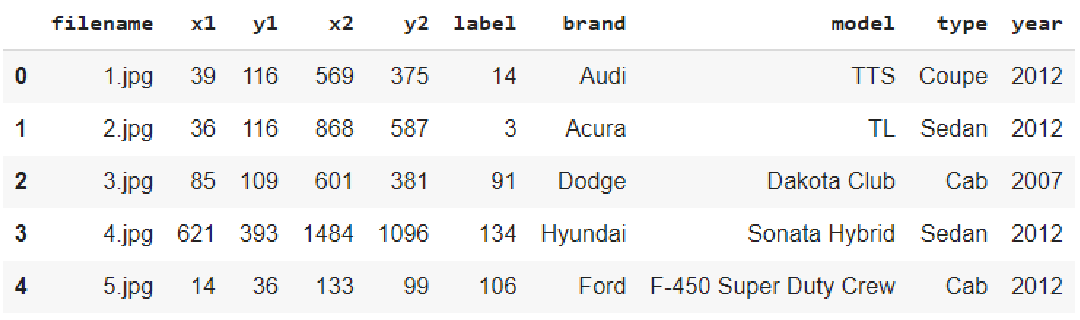
Since the data was split in 50/50 (train and test sets). For a better training process, I decided to regenerate the dataset in its original form and then split it into 70/20/10 training, validation, testing sets._To streamline the training process, the updated annotation files were concatenated into a single DataFrame, and the images from the testing directory were moved into the training directory.

Adjusting CSV Files

After renaming the images, the CSV files were updated to reflect the new filenames. This step ensures consistency between the filenames in the CSV files and the actual image files. Now instead of 2 files anno_train and anno_test, we have only 1 file final_full_data.

Adding Columns and Parsing Vehicle Descriptions

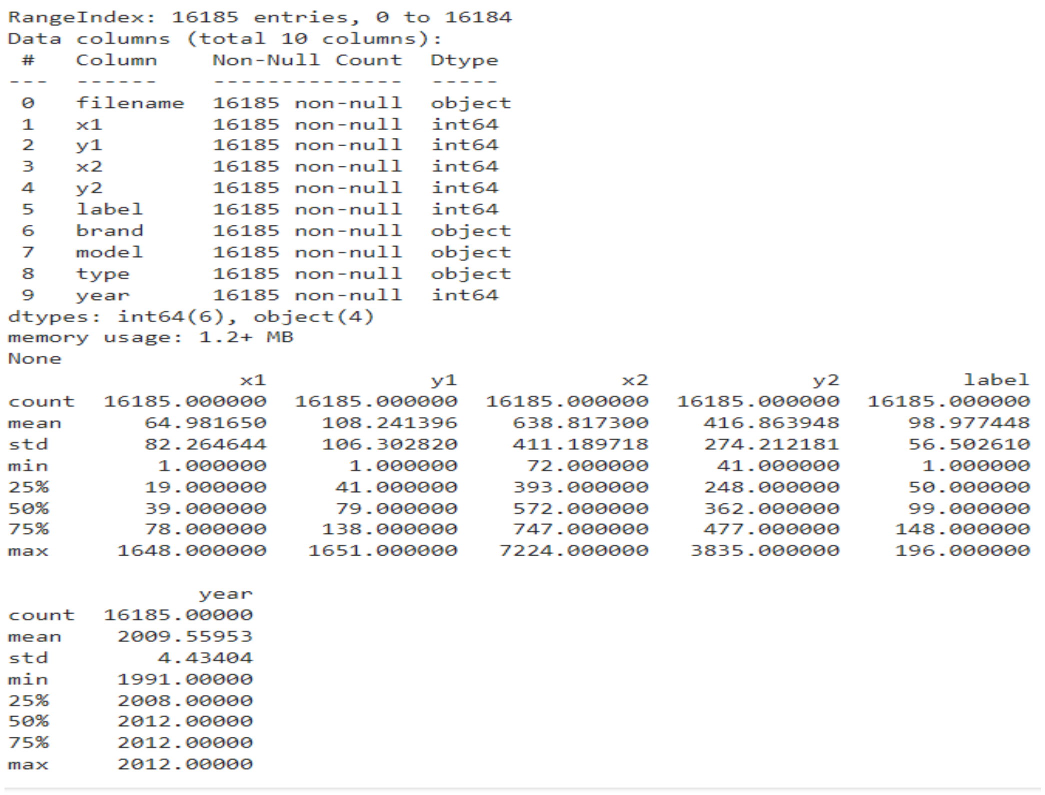
The dataset was further enriched by adding columns for brand, model, type, and year, extracted from the vehicle descriptions in the class names file. This provides more detailed annotations for each image.



	filename	x1	y1	x2	y2	label	brand	model	type	year
0	1.jpg	39	116	569	375	14	Audi	TTS	Coupe	2012
1	2.jpg	36	116	868	587	3	Acura	TL	Sedan	2012
2	3.jpg	85	109	601	381	91	Dodge	Dakota Club	Cab	2007
3	4.jpg	621	393	1484	1096	134	Hyundai	Sonata Hybrid	Sedan	2012
4	5.jpg	14	36	133	99	106	Ford	F-450 Super Duty Crew	Cab	2012

Figure 2 Modified column names and parsed car description

After modifying the CSV files and at the end of the preparation phase we run a check to see if there is any null data.



```
RangeIndex: 16185 entries, 0 to 16184
Data columns (total 10 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   filename  16185 non-null  object
 1   x1        16185 non-null  int64 
 2   y1        16185 non-null  int64 
 3   x2        16185 non-null  int64 
 4   y2        16185 non-null  int64 
 5   label     16185 non-null  int64 
 6   brand     16185 non-null  object
 7   model     16185 non-null  object
 8   type      16185 non-null  object
 9   year      16185 non-null  int64 
dtypes: int64(6), object(4)
memory usage: 1.2+ MB
None
                 x1            y1            x2            y2         label
count  16185.000000  16185.000000  16185.000000  16185.000000  16185.000000
mean      64.981650    108.241396    638.817300    416.863948     98.977448
std       82.264644    106.302820    411.189718    274.212181     56.502610
min        1.000000      1.000000     72.000000     41.000000      1.000000
25%       19.000000     41.000000    393.000000    248.000000     50.000000
50%       39.000000     79.000000    572.000000    362.000000     99.000000
75%       78.000000    138.000000    747.000000    477.000000    148.000000
max     1648.000000   1651.000000   7224.000000   3835.000000    196.000000

              year
count  16185.00000
mean    2009.55953
std        4.43404
min     1991.00000
25%     2008.00000
50%     2012.00000
75%     2012.00000
max     2012.00000
```

2.3. Data Visualization

To gain insights into the distribution and characteristics of the Stanford Cars dataset, several visualizations were created using Matplotlib and Seaborn libraries. These visualizations helped in understanding the dataset better and guided subsequent steps in model development and preprocessing.

Count of Entries by Brand - The first visualization plotted the count of car images for each brand in the dataset. This was achieved using a bar plot, where the brand column from the dataset was used to categorize the entries. The count of entries for each brand was displayed in descending order, providing a clear view of the most and least represented brands in the dataset.

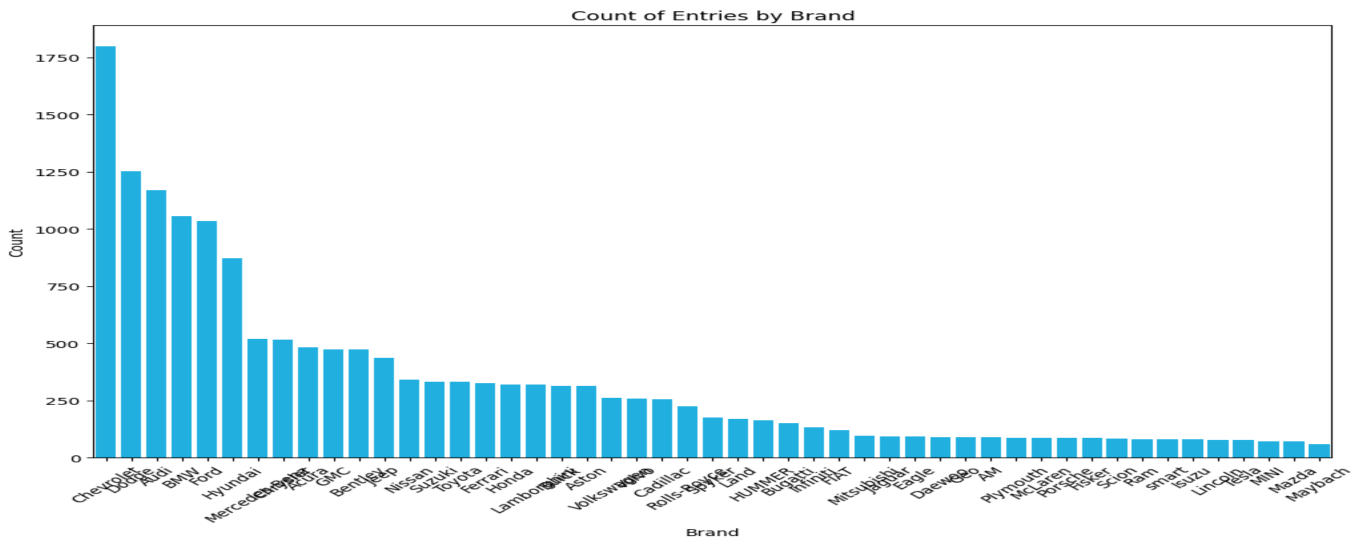


Figure 3 Brand Count

Count of Entries by Model - Given the large number of car models (196 classes), the visualization for models was limited to the top 20 most frequent car models to avoid overcrowding the plot. This bar plot used the model column to categorize the entries and provided insights into which car models had the most images.

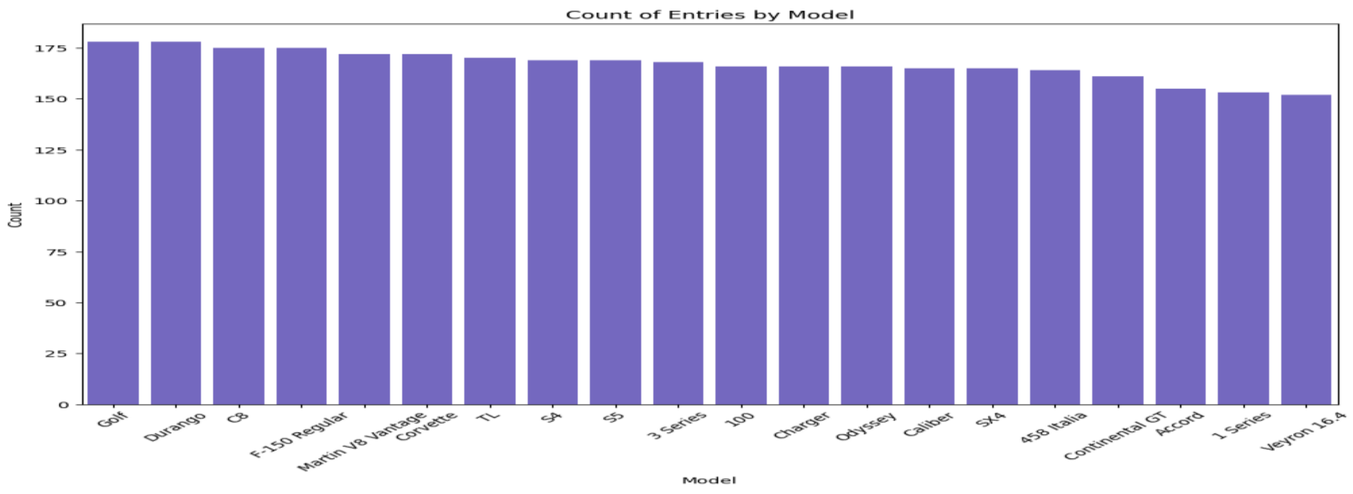


Figure 4 Model Count

Count of Entries by Type - A bar plot was also created to visualize the count of entries for each car type (e.g., SUV, sedan, convertible). The type column was used for categorization, and the entries were displayed in descending order of count.

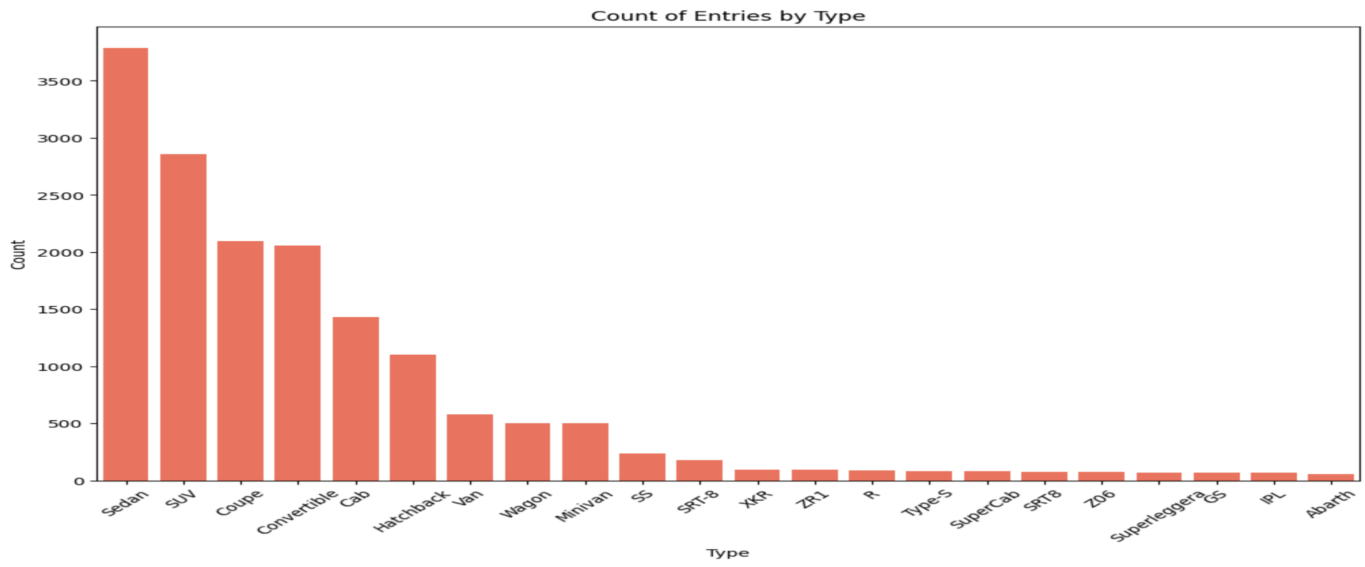


Figure 5 Vehicle Type Count

Count of Entries by Year - Finally, a bar plot was generated to show the distribution of car images by their year of manufacture. The year column was used to categorize the entries, providing a historical perspective on the dataset.

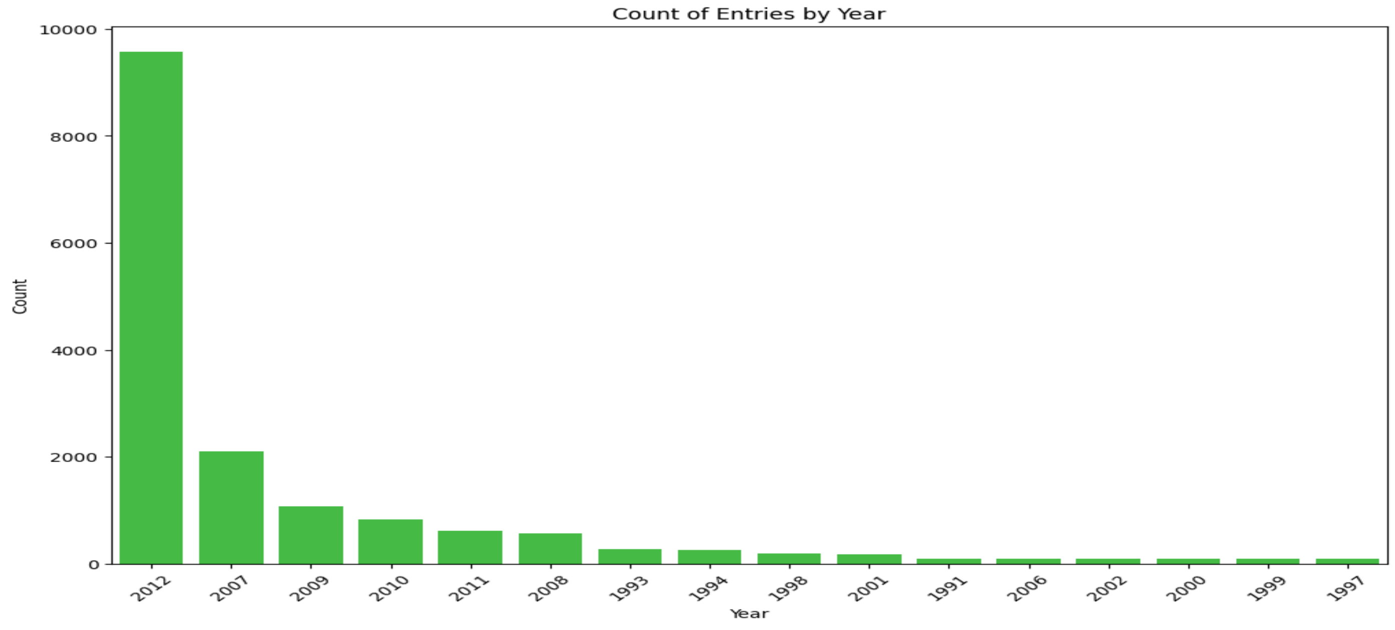


Figure 6 Year Count

These visualizations not only helped in understanding the dataset's composition but also highlighted any potential imbalances in the data distribution, which is crucial for ensuring fair and effective model training.

2.4. Data Preprocessing

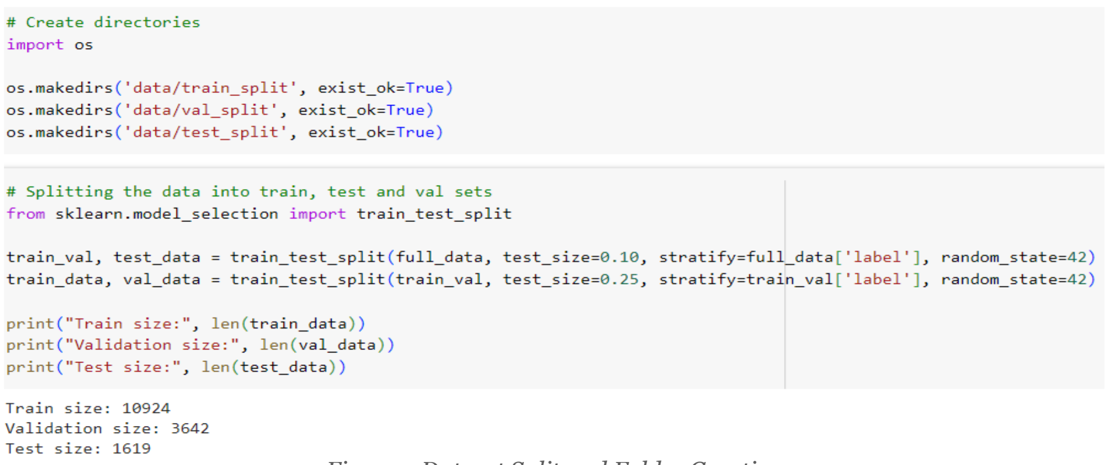
To prepare the dataset for model training, we implemented a series of preprocessing steps to ensure the images were appropriately formatted and the labels accurately corresponded to the images.

Data Splitting and Directory Organization

To facilitate the training, validation, and testing processes, the dataset was split into distinct subsets and organized into separate directories. This process involved several key steps to ensure that the data was evenly distributed and representative of all classes across the splits.

First, directories were created to store the training, validation, and testing datasets separately. This organization helps streamline the data management process during model training and evaluation.

The dataset was split into training, validation, and testing sets using the `train_test_split` function from `sklearn.model_selection`. A stratified split was performed to ensure that each subset contains a proportional representation of each class, which is critical for training a balanced and unbiased model. The data was first split into a training/validation set and a testing set, with 10% allocated to testing. The training/validation set was then further split into training and validation sets, with 25% allocated to validation.



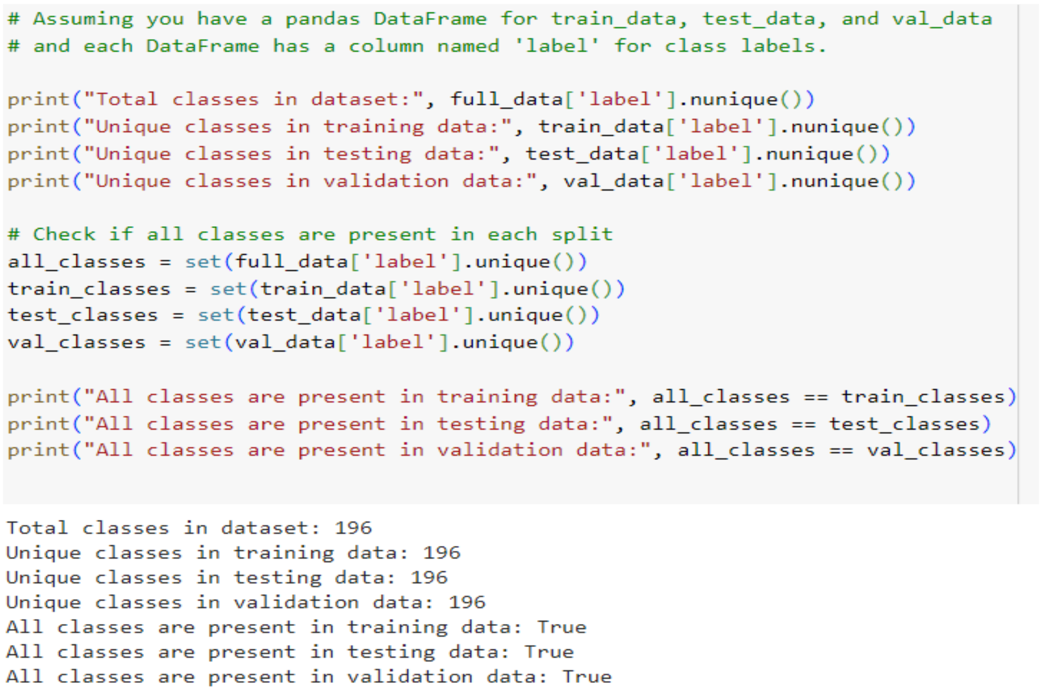
```python
# Create directories
import os

os.makedirs('data/train_split', exist_ok=True)
os.makedirs('data/val_split', exist_ok=True)
os.makedirs('data/test_split', exist_ok=True)
```

```python
# Splitting the data into train, test and val sets
from sklearn.model_selection import train_test_split

train_val, test_data = train_test_split(full_data, test_size=0.10, stratify=full_data['label'], random_state=42)
train_data, val_data = train_test_split(train_val, test_size=0.25, stratify=train_val['label'], random_state=42)

print("Train size:", len(train_data))
print("Validation size:", len(val_data))
print("Test size:", len(test_data))
```

```
Train size: 10924
Validation size: 3642
Test size: 1619
```

Figure 7 Dataset Split and Folder Creation

It is essential to ensure that all classes are represented in each data split. This was verified by checking the unique classes in the training, validation, and testing sets against the full dataset.



```python
# Assuming you have a pandas DataFrame for train_data, test_data, and val_data
# and each DataFrame has a column named 'label' for class labels.

print("Total classes in dataset:", full_data['label'].nunique())
print("Unique classes in training data:", train_data['label'].nunique())
print("Unique classes in testing data:", test_data['label'].nunique())
print("Unique classes in validation data:", val_data['label'].nunique())

# Check if all classes are present in each split
all_classes = set(full_data['label'].unique())
train_classes = set(train_data['label'].unique())
test_classes = set(test_data['label'].unique())
val_classes = set(val_data['label'].unique())

print("All classes are present in training data:", all_classes == train_classes)
print("All classes are present in testing data:", all_classes == test_classes)
print("All classes are present in validation data:", all_classes == val_classes)
```

```
Total classes in dataset: 196
Unique classes in training data: 196
Unique classes in testing data: 196
Unique classes in validation data: 196
All classes are present in training data: True
All classes are present in testing data: True
All classes are present in validation data: True
```

Figure 8 Unique Class Check

Finally, the images were moved to their corresponding directories based on the splits. This step involved iterating over the filenames in each split and relocating them from the source directory to the target directories created earlier.

By organizing the dataset in this manner, we ensured a structured and efficient workflow for the subsequent stages of model development and evaluation. This approach also facilitated the reproducibility of the experiments and the integrity of the data splits, critical for achieving reliable and consistent results.

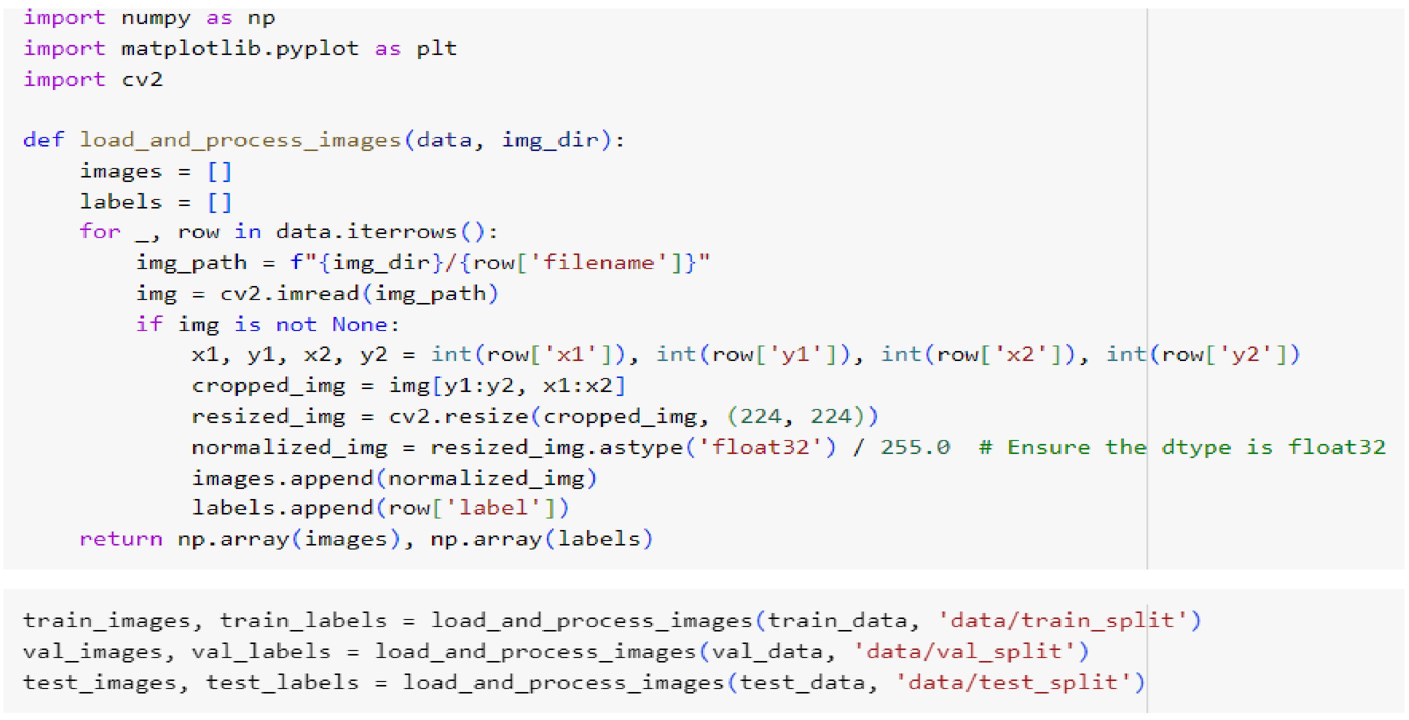
Firstly, the bounding box coordinates precisely delineate the region of interest in each image, which in our case, is the specific car in the image. By focusing on this region, we ensure that the model learns the most relevant features for classification without being distracted by background elements.

Cropping the images to these bounding boxes enhances the signal-to-noise ratio. Background clutter or irrelevant objects can introduce noise into the training data, potentially leading to incorrect classifications or a model that struggles to generalize. By using only the essential part of the image, the model can better focus on the distinguishing features of each car, such as the shape, color, and specific design elements.

Moreover, cropping images helps standardize the input data. When all images are centered on the subject and cropped to the same size, it ensures uniformity across the dataset. This uniformity is beneficial during the training process, as the model receives inputs of consistent dimensions and content, which can lead to more stable and efficient training.

Finally, this approach also aligns with the best practices in computer vision tasks. Many state-of-the-art models and datasets, such as those used in object detection and image classification challenges, rely on accurately cropped images to maximize model performance. Thus, by cropping images using the bounding box coordinates, we adopt a methodologically sound approach that is likely to yield better results.

For that reason, we developed a function to load and process the images based on their annotations. The function load_and_process_images reads each image, crops it according to the bounding box coordinates provided in the dataset, resizes it to a standard size of 224x224 pixels, and normalizes the pixel values to the range [0, 1].



```python
import numpy as np
import matplotlib.pyplot as plt
import cv2

def load_and_process_images(data, img_dir):
    images = []
    labels = []
    for _, row in data.iterrows():
        img_path = f"{img_dir}/{row['filename']}"
        img = cv2.imread(img_path)
        if img is not None:
            x1, y1, x2, y2 = int(row['x1']), int(row['y1']), int(row['x2']), int(row['y2'])
            cropped_img = img[y1:y2, x1:x2]
            resized_img = cv2.resize(cropped_img, (224, 224))
            normalized_img = resized_img.astype('float32') / 255.0  # Ensure the dtype is float32
            images.append(normalized_img)
            labels.append(row['label'])
    return np.array(images), np.array(labels)
```

```python
train_images, train_labels = load_and_process_images(train_data, 'data/train_split')
val_images, val_labels = load_and_process_images(val_data, 'data/val_split')
test_images, test_labels = load_and_process_images(test_data, 'data/test_split')
```

Figure 9 Loading and Processing Images

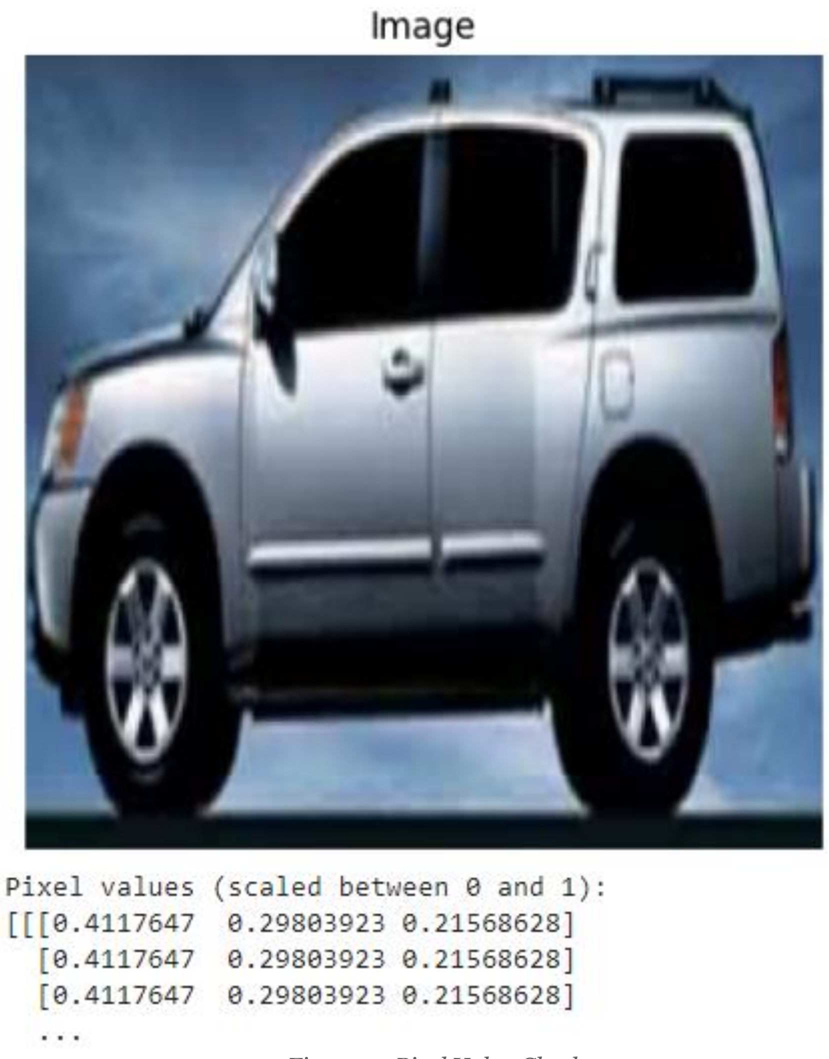
<u>Resizing Images</u>

Resizing standardizes the image dimensions, ensuring compatibility with the input requirements of pre-trained models like MobileNetV2 and ResNet50. Resizing images is crucial when working with Convolutional Neural Networks (CNNs). CNNs typically require a fixed input size for all images in the dataset. Resizing ensures that every image has the same dimensions, which is necessary for the architecture of the network that processes the input layer uniformly. Different image sizes can affect the training and inference speed of the model. Larger images can significantly increase the computational cost, while smaller images can reduce the amount of detail the model has to learn from. Many pre-trained models, like ResNet, or MobileNet, expect input images of a certain size, more specifically 224 x 224.

Using the expected image size helps leverage the full potential of these models and ensures that features learned during pre-training are applicable to your task. Also, an optimized image size can improve the performance of the model not just in terms of speed but also accuracy.

<u>Normalizing Pixel Values</u>

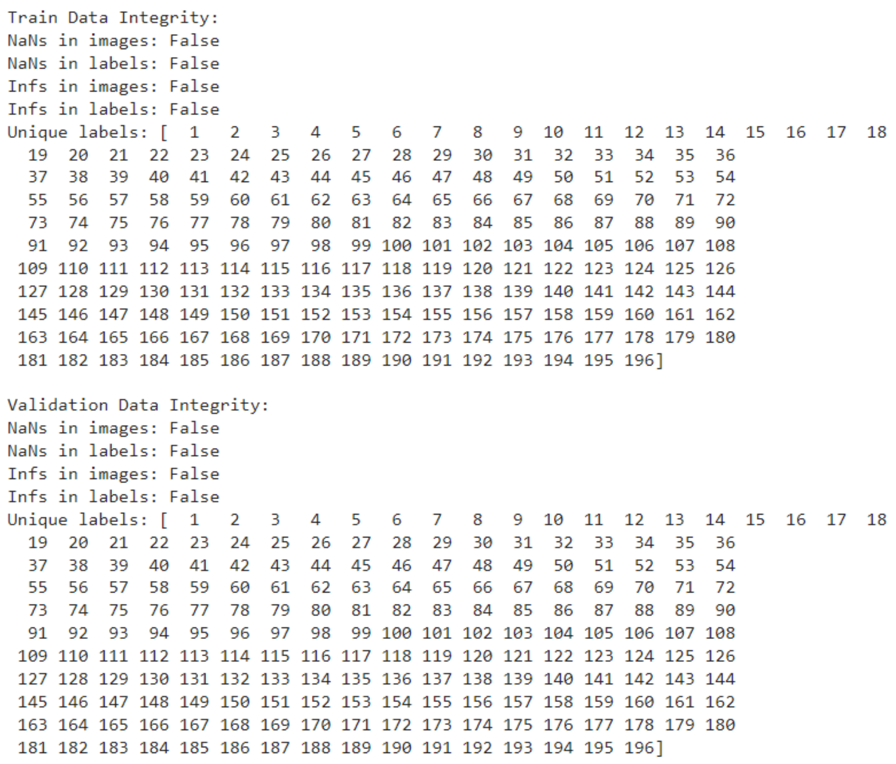
Normalizing pixel values is a fundamental preprocessing step that accelerates the training process and enhances model performance by ensuring that the inputs are on a similar scale. Normalization transforms the pixel values to a standard range, typically between 0 and 1. This consistency across input data helps the model learn more efficiently by stabilizing the gradients, leading to faster convergence and improved overall accuracy.



```
Pixel values (scaled between 0 and 1):
[[[0.4117647  0.29803923 0.21568628]
  [0.4117647  0.29803923 0.21568628]
  [0.4117647  0.29803923 0.21568628]
  ...
```

Figure 10 Pixel Value Check

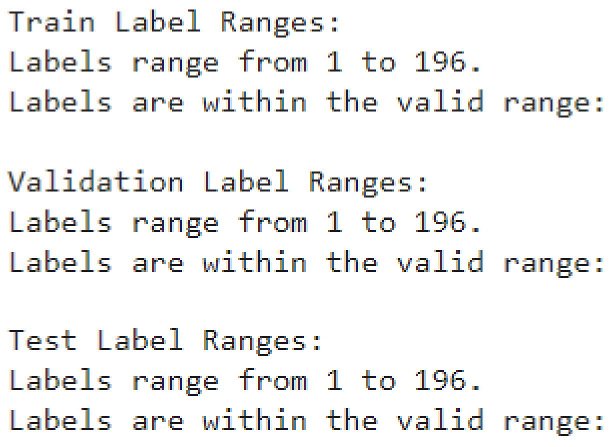
To further ensure the integrity of the data, we conducted several checks for the presence of NaN or infinite values in the images and labels. We also verified that the labels were within the expected range and that each subset of data contained all the unique labels from the dataset.



```
Train Data Integrity:
NaNs in images: False
NaNs in labels: False
Infs in images: False
Infs in labels: False
Unique labels: [  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196]

Validation Data Integrity:
NaNs in images: False
NaNs in labels: False
Infs in images: False
Infs in labels: False
Unique labels: [  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196]
```

Figure 11 NaN Values Check

We ensured that all labels were within the valid range by checking the minimum and maximum values of the labels against the total number of classes in the dataset.

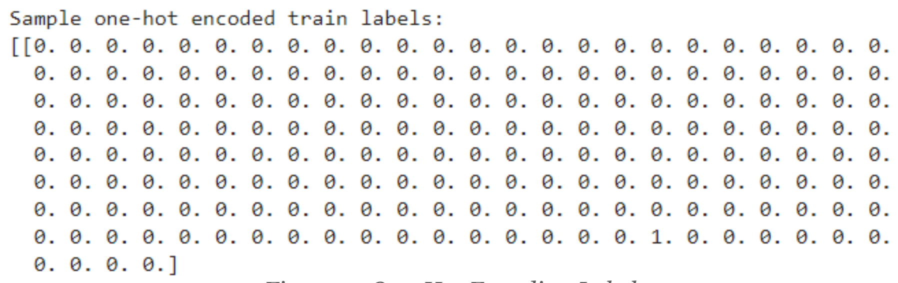


```
Train Label Ranges:
Labels range from 1 to 196.
Labels are within the valid range:

Validation Label Ranges:
Labels range from 1 to 196.
Labels are within the valid range:

Test Label Ranges:
Labels range from 1 to 196.
Labels are within the valid range:
```

Figure 12 Valid label range

Label Encoding

The labels were adjusted to start from zero and converted to one-hot encoding to be compatible with the model training process.



```
Sample one-hot encoded train labels:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0.]
```

Figure 13 One-Hot Encoding Labels

Finally, to ensure the integrity of the image-label pairs, we visualized a random sample of images along with their corresponding labels. This step helped verify that the images were correctly processed and that the labels matched the images.

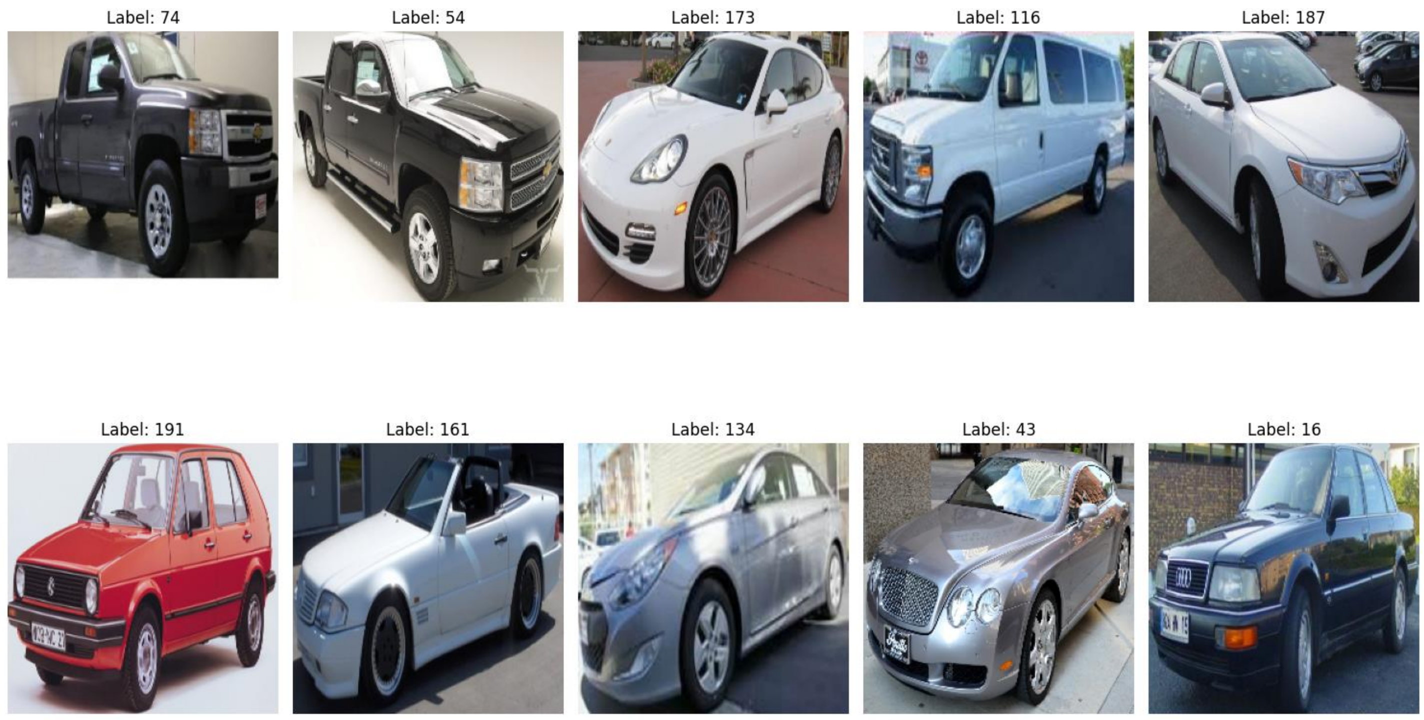
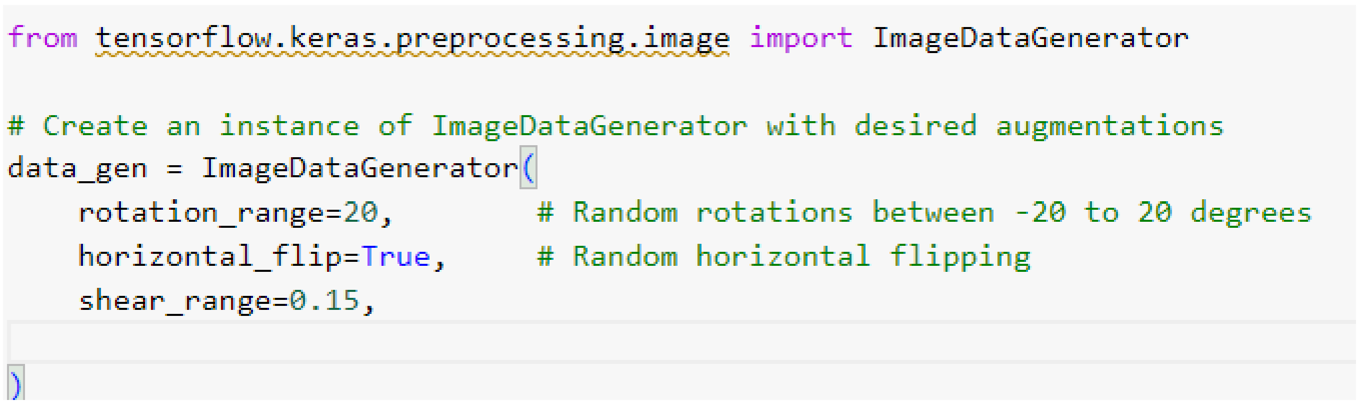


Figure 14 Check image and label match

<u>Data augmentation</u>

Data augmentation is a critical technique in machine learning, particularly in image classification tasks, where it involves generating additional training data from existing images. This is achieved through various transformations such as rotations, translations, and flips. Data augmentation helps to artificially expand the size of the training dataset, which can significantly enhance the performance and robustness of the model by preventing overfitting and improving generalization to new data.

In our project, we employed several augmentation techniques using the ImageDataGenerator class from TensorFlow's Keras API. The augmentations included random rotations between -20 to 20 degrees, horizontal and vertical shifts of up to 20%, shear transformations, random zooms, and horizontal flips. These transformations helped to simulate the variations and distortions that the model might encounter in real-world scenarios, thereby making it more resilient.



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

# Create an instance of ImageDataGenerator with desired augmentations
data_gen = ImageDataGenerator(
    rotation_range=20,         # Random rotations between -20 to 20 degrees
    horizontal_flip=True,      # Random horizontal flipping
    shear_range=0.15,

)
```

Figure 15 ImageDataGenerator and Augmentations

However, not all augmentations yielded positive results. Initially, we also experimented with brightness adjustments and channel shifts, but these caused excessive distortions, leading to a significant degradation in model performance. Images subjected to these transformations appeared unrealistic and impaired the model's ability to learn meaningful features. As a result, we decided to exclude brightness and channel shift augmentations and retained only those augmentations that contributed positively to the model's performance.

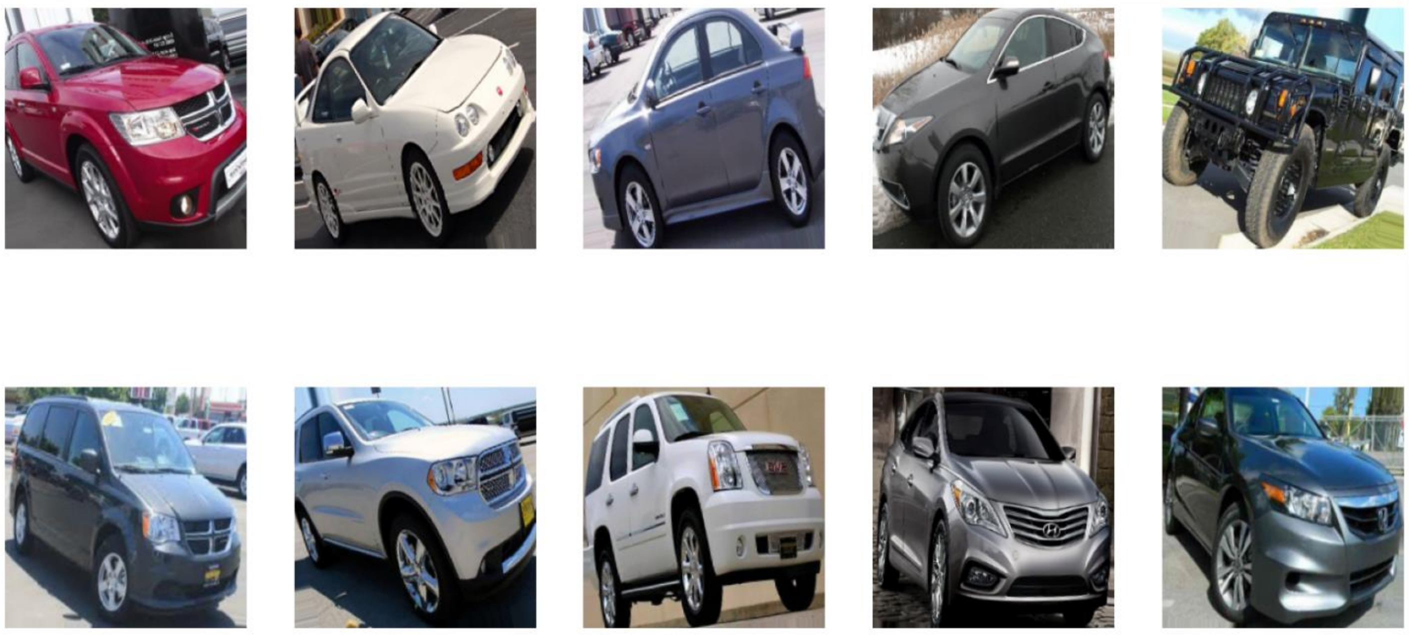


Figure 16 Augmented Images

3. Model Development

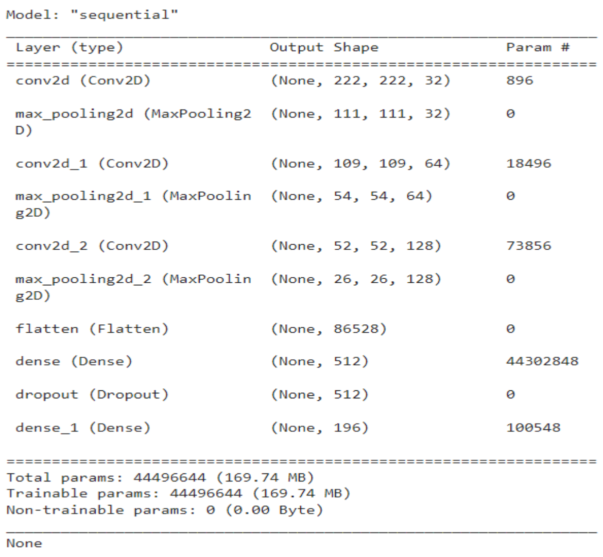
In this section, we focus on developing and training our image classification models. The objective is to build a robust model capable of accurately classifying images into their respective categories. Initially, we will create and train a basic Convolutional Neural Network (CNN) model from scratch. This custom CNN will serve as a baseline to understand the performance and limitations of a straightforward approach.

After establishing the baseline with our custom CNN, we will leverage transfer learning by using a pre-trained MobileNetV2 model. Transfer learning involves utilizing a model that has been previously trained on a large dataset, such as ImageNet, and fine-tuning it for our specific task. This method often leads to improved performance as the pre-trained model has already learned a wide range of features from its original training.

By comparing the performance of the basic CNN with the pre-trained MobileNetV2 model, we aim to highlight the benefits of transfer learning and determine the most effective approach for our image classification task. Below are the details of the models we developed and their respective training processes.

3.1. Basic Convolutional Neural Network (CNN) Model

The basic Convolutional Neural Network (CNN) model is designed to serve as a foundational approach for the image classification task. Here’s an overview of its architecture and components:



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 222, 222, 32)      896

 max_pooling2d (MaxPooling2  (None, 111, 111, 32)      0
 D)

 conv2d_1 (Conv2D)           (None, 109, 109, 64)      18496

 max_pooling2d_1 (MaxPoolin  (None, 54, 54, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 52, 52, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 26, 26, 128)       0
 g2D)

 flatten (Flatten)           (None, 86528)             0

 dense (Dense)               (None, 512)               44302848

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 196)               100548

=================================================================
Total params: 44496644 (169.74 MB)
Trainable params: 44496644 (169.74 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

Figure 17 Basic CNN Model Info

Convolutional Layers: These layers are responsible for feature extraction from the input images. Each convolutional layer applies a set of filters to the input image, producing feature maps that highlight various aspects of the input.

Conv2D(32, (3, 3), activation='relu'): This layer applies 32 filters, each of size 3x3, to the input image. The ReLU (Rectified Linear Unit) activation function introduces non-linearity to the model, allowing it to learn more complex features.
Conv2D(64, (3, 3), activation='relu'): This layer increases the number of filters to 64, further refining the feature extraction process.
Conv2D(128, (3, 3), activation='relu'): This layer applies 128 filters, continuing to extract higher-level features from the input image.

Pooling Layers: These layers downsample the feature maps, reducing their dimensions and the number of parameters in the model, which helps prevent overfitting.
MaxPooling2D((2, 2)): Each MaxPooling layer reduces the size of the feature maps by selecting the maximum value in each 2x2 region, effectively halving the spatial dimensions of the feature maps.

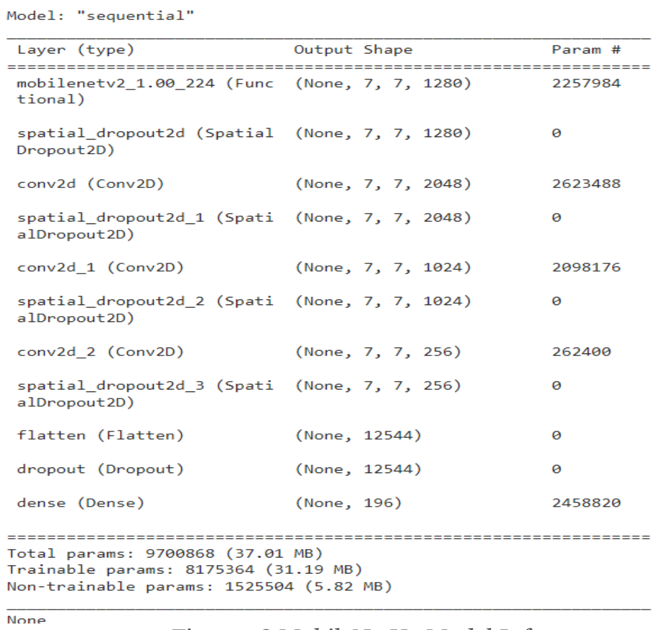
Fully Connected Layers: These layers, also known as dense layers, are used to perform the final classification based on the features extracted by the convolutional layers.

Flatten(): This layer converts the 2D feature maps into a 1D vector, preparing the data for the dense layers.
Dense(512, activation='relu'): This fully connected layer with 512 neurons further processes the flattened feature vector. The ReLU activation function is used to introduce non-linearity.

Dropout(0.5): This layer randomly drops 50% of its inputs during training, which helps prevent overfitting by ensuring that the model doesn’t rely too heavily on any single feature.
Dense(num_classes, activation='softmax'): The output layer uses the softmax activation function to produce a probability distribution over the 196 classes.

3.2. MobileNetV2 Pre-Trained Model

The MobileNetV2 model is a pre-trained convolutional neural network that has been trained on the ImageNet dataset. Transfer learning is applied here by fine-tuning this model for the specific image classification task.
Base Model (MobileNetV2): The MobileNetV2 architecture serves as the backbone for this model. It includes several layers optimized for efficient computation and high accuracy.
pretrained_model(weights='imagenet', include_top=False, input_shape=input_shape): This line loads the MobileNetV2 model with pre-trained weights from the ImageNet dataset, excluding the top classification layers. The include_top=False argument ensures that only the feature extraction layers are used.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)        2257984
 tional)

 spatial_dropout2d (Spatial  (None, 7, 7, 1280)        0
 Dropout2D)

 conv2d (Conv2D)             (None, 7, 7, 2048)        2623488

 spatial_dropout2d_1 (Spati  (None, 7, 7, 2048)        0
 alDropout2D)

 conv2d_1 (Conv2D)           (None, 7, 7, 1024)        2098176

 spatial_dropout2d_2 (Spati  (None, 7, 7, 1024)        0
 alDropout2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 256)         262400

 spatial_dropout2d_3 (Spati  (None, 7, 7, 256)         0
 alDropout2D)

 flatten (Flatten)           (None, 12544)             0

 dropout (Dropout)           (None, 12544)             0

 dense (Dense)               (None, 196)               2458820

=================================================================
Total params: 9700868 (37.01 MB)
Trainable params: 8175364 (31.19 MB)
Non-trainable params: 1525504 (5.82 MB)
_________________________________________________________________
None
```

Figure 18 MobileNetV2 Model Info

Custom Layers: These additional layers are added on top of the base MobileNetV2 model to adapt it for the specific classification task.

SpatialDropout2D(0.30): This layer applies dropout to the feature maps, randomly setting a fraction (30%) of the input units to zero at each update during training.
Conv2D(2048, (1, 1), padding='same'): This 1x1 convolutional layer with 2048 filters increases the depth of the feature maps, allowing the model to learn more complex patterns.
SpatialDropout2D(0.25): Another dropout layer to prevent overfitting.
Conv2D(1024, (1, 1), padding='same'): This layer reduces the depth of the feature maps to 1024.
Conv2D(256, (1, 1), padding='same'): Further reduces the depth to 256.
Flatten(): Converts the 2D feature maps into a 1D vector.
Dropout(0.40): Adds dropout to the fully connected layer.
Dense(196, activation='softmax'): The output layer for classifying the 196 categories.

Both models are compiled with the Adam optimizer, a categorical cross-entropy loss function, and categorical accuracy as the performance metric. Early stopping and learning rate scheduler callbacks are used to prevent overfitting and dynamically adjust the learning rate during training.

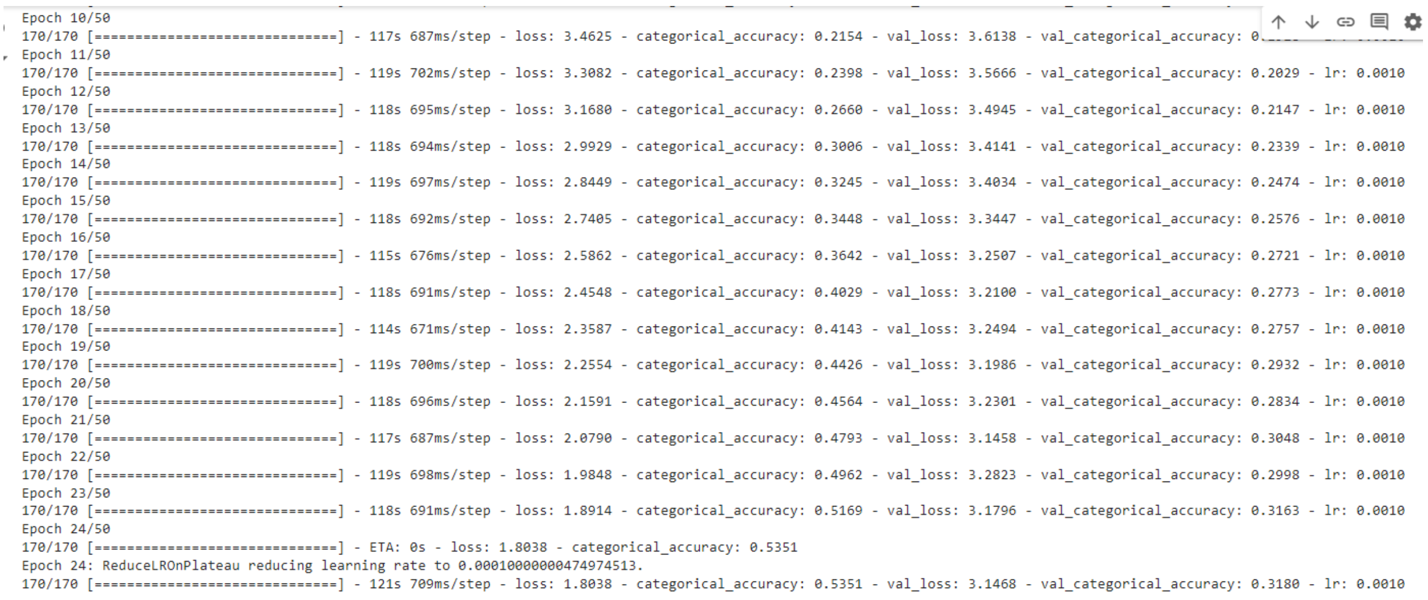
3.3. Training phase

Early Stopping and Learning Rate Scheduling

In the development of the image classification model, early stopping and learning rate scheduling were implemented to optimize the training process and improve the model's performance.

Early stopping is a regularization technique used to prevent overfitting during the training process. In this project, the `EarlyStopping` callback from the TensorFlow Keras library was used. The early stopping mechanism monitors the validation loss during training. The following parameters were set:

- **monitor='val_loss'**: The validation loss is monitored to determine the performance of the model on unseen data.
- **patience=5**: Training is stopped if the validation loss does not improve for 5 consecutive epochs.
- **restore_best_weights=True**: Ensures that the model weights are restored to the point where the validation loss was the lowest, providing the best possible model from the training process.



```
Epoch 10/50
170/170 [==============================] - 117s 687ms/step - loss: 3.4625 - categorical_accuracy: 0.2154 - val_loss: 3.6138 - val_categorical_accuracy: 0.
Epoch 11/50
170/170 [==============================] - 119s 702ms/step - loss: 3.3082 - categorical_accuracy: 0.2398 - val_loss: 3.5666 - val_categorical_accuracy: 0.2029 - lr: 0.0010
Epoch 12/50
170/170 [==============================] - 118s 695ms/step - loss: 3.1680 - categorical_accuracy: 0.2660 - val_loss: 3.4945 - val_categorical_accuracy: 0.2147 - lr: 0.0010
Epoch 13/50
170/170 [==============================] - 118s 694ms/step - loss: 2.9929 - categorical_accuracy: 0.3006 - val_loss: 3.4141 - val_categorical_accuracy: 0.2339 - lr: 0.0010
Epoch 14/50
170/170 [==============================] - 119s 697ms/step - loss: 2.8449 - categorical_accuracy: 0.3245 - val_loss: 3.4034 - val_categorical_accuracy: 0.2474 - lr: 0.0010
Epoch 15/50
170/170 [==============================] - 118s 692ms/step - loss: 2.7405 - categorical_accuracy: 0.3448 - val_loss: 3.3447 - val_categorical_accuracy: 0.2576 - lr: 0.0010
Epoch 16/50
170/170 [==============================] - 115s 676ms/step - loss: 2.5862 - categorical_accuracy: 0.3642 - val_loss: 3.2507 - val_categorical_accuracy: 0.2721 - lr: 0.0010
Epoch 17/50
170/170 [==============================] - 118s 691ms/step - loss: 2.4548 - categorical_accuracy: 0.4029 - val_loss: 3.2100 - val_categorical_accuracy: 0.2773 - lr: 0.0010
Epoch 18/50
170/170 [==============================] - 114s 671ms/step - loss: 2.3587 - categorical_accuracy: 0.4143 - val_loss: 3.2494 - val_categorical_accuracy: 0.2757 - lr: 0.0010
Epoch 19/50
170/170 [==============================] - 119s 700ms/step - loss: 2.2554 - categorical_accuracy: 0.4426 - val_loss: 3.1986 - val_categorical_accuracy: 0.2932 - lr: 0.0010
Epoch 20/50
170/170 [==============================] - 118s 696ms/step - loss: 2.1591 - categorical_accuracy: 0.4564 - val_loss: 3.2301 - val_categorical_accuracy: 0.2834 - lr: 0.0010
Epoch 21/50
170/170 [==============================] - 117s 687ms/step - loss: 2.0790 - categorical_accuracy: 0.4793 - val_loss: 3.1458 - val_categorical_accuracy: 0.3048 - lr: 0.0010
Epoch 22/50
170/170 [==============================] - 119s 698ms/step - loss: 1.9848 - categorical_accuracy: 0.4962 - val_loss: 3.2823 - val_categorical_accuracy: 0.2998 - lr: 0.0010
Epoch 23/50
170/170 [==============================] - 118s 691ms/step - loss: 1.8914 - categorical_accuracy: 0.5169 - val_loss: 3.1796 - val_categorical_accuracy: 0.3163 - lr: 0.0010
Epoch 24/50
170/170 [==============================] - ETA: 0s - loss: 1.8038 - categorical_accuracy: 0.5351
Epoch 24: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
170/170 [==============================] - 121s 709ms/step - loss: 1.8038 - categorical_accuracy: 0.5351 - val_loss: 3.1468 - val_categorical_accuracy: 0.3180 - lr: 0.0010
```

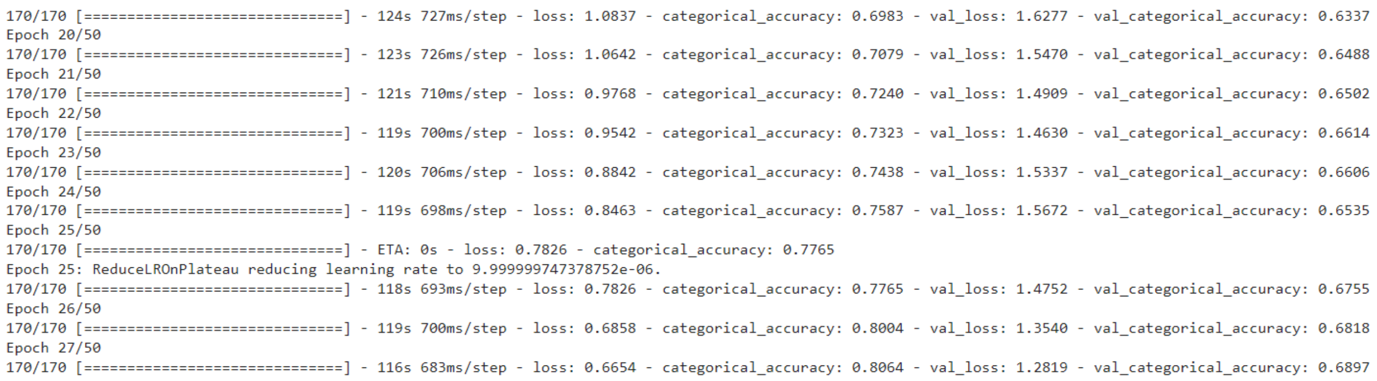
Figure 19 EarlyStopping in action

This approach helps in stopping the training process once the model stops improving, thus saving computational resources and preventing overfitting like it happened when training the basic CNN model in the figure above. The training process has stopped after 24 epochs because of the lack of improvement and implementation of EarlyStopping.

Learning rate scheduling dynamically adjusts the learning rate during training to enhance the model's convergence. The `ReduceLROnPlateau` callback was used to implement this. Key parameters include:

- **monitor='val_loss'**: The validation loss is monitored to adjust the learning rate.
- **factor=0.1**: The learning rate is reduced by a factor of 0.1 when a plateau in the validation loss is detected.
- **patience=3**: The learning rate is reduced if the validation loss does not improve for 3 consecutive epochs.

- **min_lr=1e-7**: A minimum learning rate is set to prevent the learning rate from becoming too small.
- **verbose=1**: A message is printed each time the learning rate is reduced, providing insights into the training process.



```
170/170 [==============================] - 124s 727ms/step - loss: 1.0837 - categorical_accuracy: 0.6983 - val_loss: 1.6277 - val_categorical_accuracy: 0.6337
Epoch 20/50
170/170 [==============================] - 123s 726ms/step - loss: 1.0642 - categorical_accuracy: 0.7079 - val_loss: 1.5470 - val_categorical_accuracy: 0.6488
Epoch 21/50
170/170 [==============================] - 121s 710ms/step - loss: 0.9768 - categorical_accuracy: 0.7240 - val_loss: 1.4909 - val_categorical_accuracy: 0.6502
Epoch 22/50
170/170 [==============================] - 119s 700ms/step - loss: 0.9542 - categorical_accuracy: 0.7323 - val_loss: 1.4630 - val_categorical_accuracy: 0.6614
Epoch 23/50
170/170 [==============================] - 120s 706ms/step - loss: 0.8842 - categorical_accuracy: 0.7438 - val_loss: 1.5337 - val_categorical_accuracy: 0.6606
Epoch 24/50
170/170 [==============================] - 119s 698ms/step - loss: 0.8463 - categorical_accuracy: 0.7587 - val_loss: 1.5672 - val_categorical_accuracy: 0.6535
Epoch 25/50
170/170 [==============================] - ETA: 0s - loss: 0.7826 - categorical_accuracy: 0.7765
Epoch 25: ReduceLROnPlateau reducing learning rate to 9.999999747378752e-06.
170/170 [==============================] - 118s 693ms/step - loss: 0.7826 - categorical_accuracy: 0.7765 - val_loss: 1.4752 - val_categorical_accuracy: 0.6755
Epoch 26/50
170/170 [==============================] - 119s 700ms/step - loss: 0.6858 - categorical_accuracy: 0.8004 - val_loss: 1.3540 - val_categorical_accuracy: 0.6818
Epoch 27/50
170/170 [==============================] - 116s 683ms/step - loss: 0.6654 - categorical_accuracy: 0.8064 - val_loss: 1.2819 - val_categorical_accuracy: 0.6897
```

Figure 20 Learning Rate Scheduler in action

The learning rate scheduler ensures that the model can continue to make progress even when the training stagnates, by allowing finer adjustments to the model's parameters as training progresses.

<u>Model Training</u>

The model was trained with these callbacks integrated into the `fit` method. The steps per epoch were calculated based on the number of training images and the batch size:

- **steps_size = len(train_images) // batch_size**: This defines the number of steps the model will run through the data in each epoch.

The `fit` method was called with the following parameters:

- **epochs=50**: The model is set to train for 50 epochs, although early stopping can halt training earlier if no improvement is observed.
- **steps_per_epoch=steps_size**: Defines the number of batch iterations per epoch.
- **validation_data=(val_images, val_labels_one_hot)**: Provides the validation data for evaluation during training.
- **callbacks=[early_stopping, learning_rate_scheduler, metrics callback]**: Integrates the early stopping and learning rate scheduler callbacks into the training process and also the metrics callback which elaborates on the performance metrics.

These strategies collectively enhance the model's training efficiency and effectiveness, ensuring better generalization to unseen data while maintaining computational efficiency.

3.4. Model Fine-Tuning

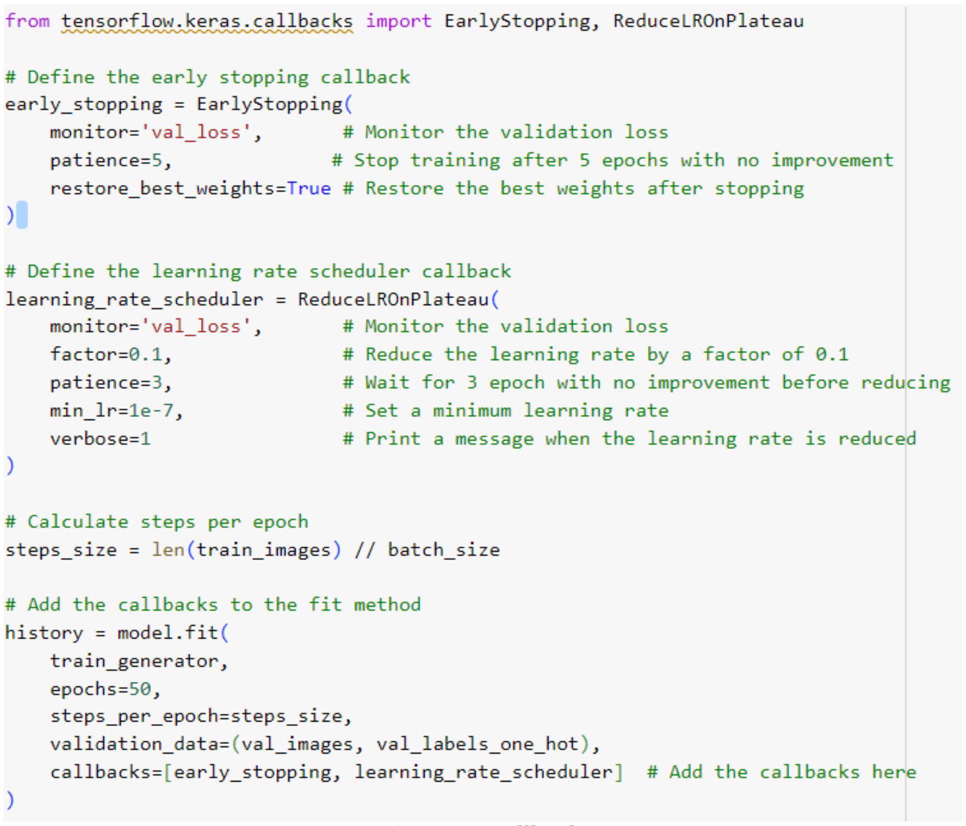
Fine-tuning the models was a crucial step to enhance the performance and generalization capability of the image classification models. The following techniques were employed to fine-tune the models effectively:

<u>Using Callbacks</u>

Callbacks are essential tools for controlling the training process and optimizing the model. The use of EarlyStopping and ReduceLROnPlateau callbacks played a significant role in preventing overfitting and ensuring the model converges appropriately.

- EarlyStopping: This callback monitors the validation loss and stops the training process if there is no improvement in the validation loss for a specified number of epochs (patience). This helps in avoiding overfitting and reduces unnecessary computation. The best weights are restored to the model after stopping, ensuring the model is in its best state.

- ReduceLROnPlateau: This callback reduces the learning rate when the validation loss has stopped improving. By reducing the learning rate, the model can make finer adjustments to its parameters, which is particularly useful towards the end of training.



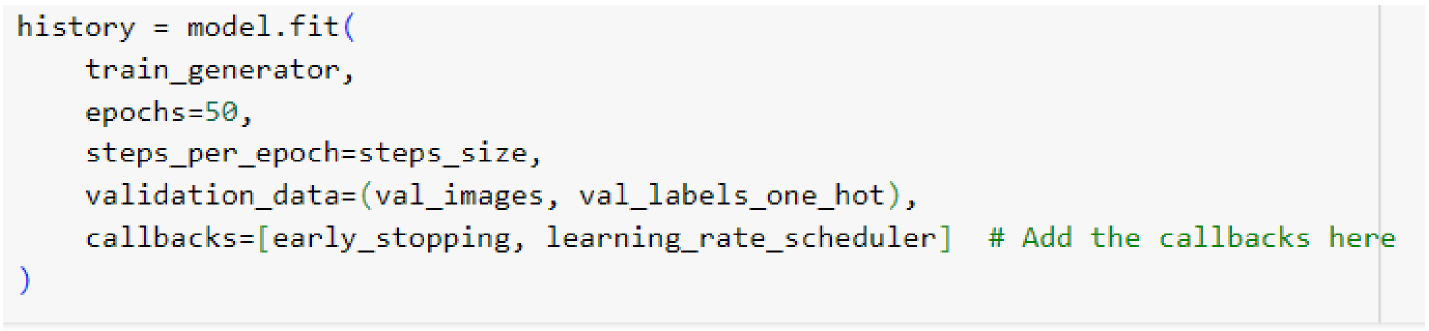
```python
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Define the early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',       # Monitor the validation loss
    patience=5,              # Stop training after 5 epochs with no improvement
    restore_best_weights=True # Restore the best weights after stopping
)

# Define the learning rate scheduler callback
learning_rate_scheduler = ReduceLROnPlateau(
    monitor='val_loss',       # Monitor the validation loss
    factor=0.1,               # Reduce the learning rate by a factor of 0.1
    patience=3,               # Wait for 3 epoch with no improvement before reducing
    min_lr=1e-7,              # Set a minimum learning rate
    verbose=1                 # Print a message when the learning rate is reduced
)

# Calculate steps per epoch
steps_size = len(train_images) // batch_size

# Add the callbacks to the fit method
history = model.fit(
    train_generator,
    epochs=50,
    steps_per_epoch=steps_size,
    validation_data=(val_images, val_labels_one_hot),
    callbacks=[early_stopping, learning_rate_scheduler]  # Add the callbacks here
)
```

Figure 21 Callbacks

Experimenting with Different Batch Sizes and Epoch Numbers

Batch size and the number of epochs are critical hyperparameters that significantly impact the model's performance and training time. Various batch sizes and epoch numbers were experimented with to find the optimal configuration:

Batch Size: Smaller batch sizes can provide a regularizing effect and improve the generalization of the model, but they can also result in noisier gradient estimates. Larger batch sizes, on the other hand, can provide more accurate gradient estimates but may lead to overfitting. Different batch sizes (e.g., 32, 64, 128) were tested to find a balance between these effects.



```python
history = model.fit(
    train_generator,
    epochs=50,
    steps_per_epoch=steps_size,
    validation_data=(val_images, val_labels_one_hot),
    callbacks=[early_stopping, learning_rate_scheduler]  # Add the callbacks here
)
```

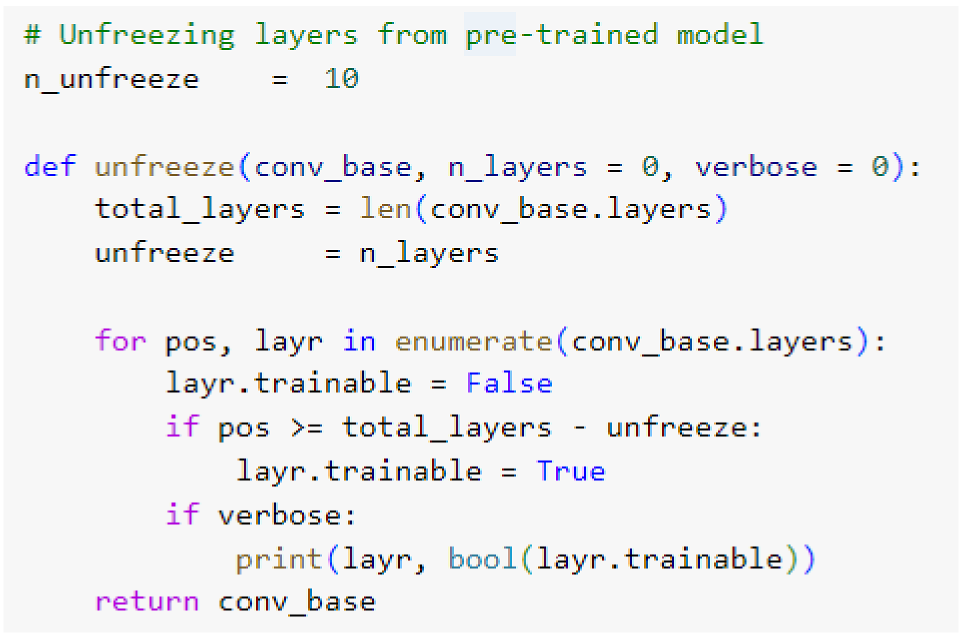
Figure 22 Training Settings

Epoch Numbers: The number of epochs determines how long the model trains on the dataset. Training for too few epochs may result in underfitting, while too many epochs can lead to overfitting. Different numbers of epochs were tested to ensure the model trains sufficiently without overfitting.

<u>Unfreezing Layers from the Pre-Trained Model</u>
Unfreezing certain layers of the pre-trained model allows the model to fine-tune the features learned by these layers based on the new dataset. This is particularly useful in transfer learning where the pre-trained model was initially trained on a different dataset.

Unfreezing Layers: By selectively unfreezing the top layers of the pre-trained model, the model can adjust its feature representations to better suit the new dataset. The number of layers to unfreeze was experimented with to find the optimal balance between retaining the pre-trained knowledge and adapting to the new data.

These strategies improved the model’s ability to learn from the dataset while preventing overfitting and ensuring efficient use of computational resources. The combination of callbacks, careful selection of batch sizes and epochs, and strategic unfreezing of layers from pre-trained models provided a robust framework for optimizing the model's performance.



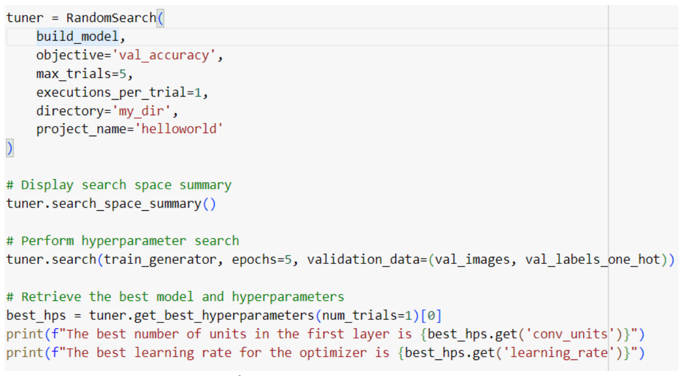
```python
# Unfreezing layers from pre-trained model
n_unfreeze    =  10

def unfreeze(conv_base, n_layers = 0, verbose = 0):
    total_layers = len(conv_base.layers)
    unfreeze     = n_layers

    for pos, layr in enumerate(conv_base.layers):
        layr.trainable = False
        if pos >= total_layers - unfreeze:
            layr.trainable = True
        if verbose:
            print(layr, bool(layr.trainable))
    return conv_base
```

Figure 23 Unfreeze Layers Functions

3.5. Hyper-parameter tuning

Hyperparameter Tuning with Keras Tuner

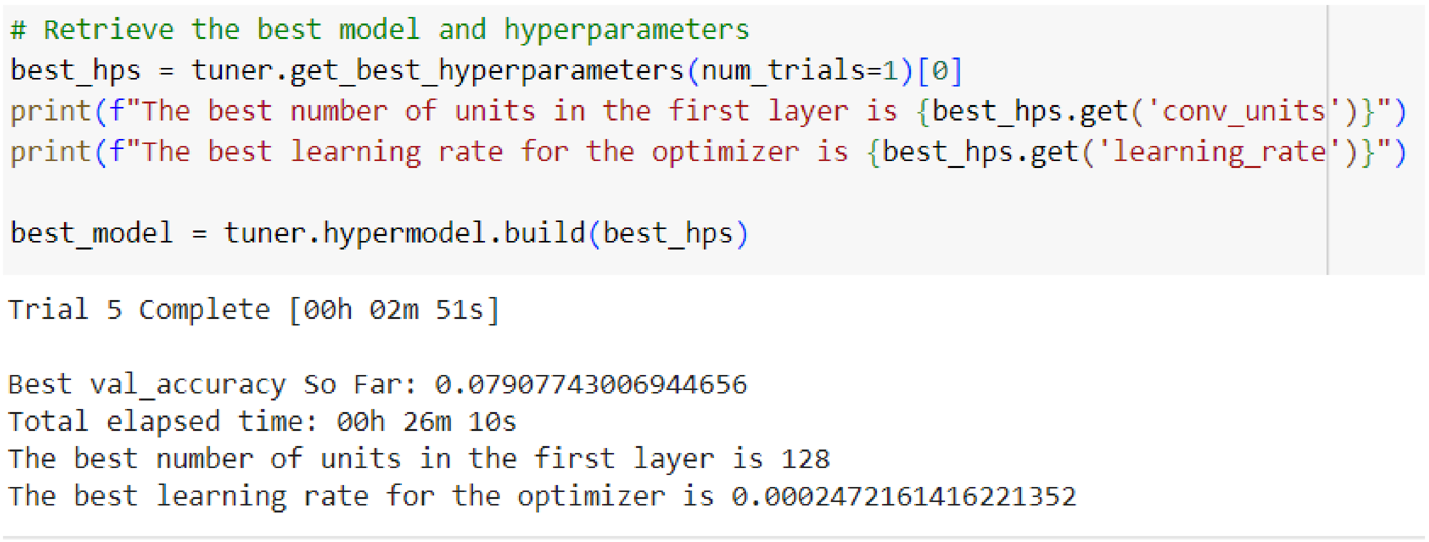
Hyperparameter tuning is a crucial step in optimizing the performance of a machine learning model. For this project, Keras Tuner was used to automate the process of searching for the best hyperparameters. Keras Tuner provides a systematic approach to identifying optimal hyperparameters by defining a search space and utilizing search algorithms like Random Search, Hyperband, or Bayesian Optimization.



```
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=1,
    directory='my_dir',
    project_name='helloworld'
)

# Display search space summary
tuner.search_space_summary()

# Perform hyperparameter search
tuner.search(train_generator, epochs=5, validation_data=(val_images, val_labels_one_hot))

# Retrieve the best model and hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"The best number of units in the first layer is {best_hps.get('conv_units')}")
print(f"The best learning rate for the optimizer is {best_hps.get('learning_rate')}")
```

Figure 24 Keras Tuner

The implementation began by installing Keras Tuner and defining a model-building function that incorporates hyperparameters for various model parameters such as the number of units in convolutional and dense layers, dropout rates, and learning rates. The RandomSearch class from Keras Tuner was then used to explore the hyperparameter space. By specifying the objective (validation accuracy), the maximum number of trials, and the executions per trial, the tuner effectively searched for the best combination of hyperparameters. This automated search saved significant time and effort compared to manual tuning and ensured a more thorough exploration of the hyperparameter space.



```
# Retrieve the best model and hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"The best number of units in the first layer is {best_hps.get('conv_units')}")
print(f"The best learning rate for the optimizer is {best_hps.get('learning_rate')}")

best_model = tuner.hypermodel.build(best_hps)
```

```
Trial 5 Complete [00h 02m 51s]

Best val_accuracy So Far: 0.07907743006944656
Total elapsed time: 00h 26m 10s
The best number of units in the first layer is 128
The best learning rate for the optimizer is 0.0002472161416221352
```

Figure 25 Hyperparameter Searching

The results of the hyperparameter tuning were then used to build the final model, which was trained and evaluated to verify the improvements. This method provided a structured and efficient way to optimize model performance, ensuring that the best possible hyperparameters were chosen for training the neural network.

4. Results and Discussion

4.1. Model Performance

To gain a comprehensive understanding of the model's performance, additional evaluation metrics such as F1-score, precision, and recall were included during training. These metrics provide deeper insights into the model's effectiveness, particularly in handling class imbalances.

The F1-score is a harmonic mean of precision and recall, offering a single metric that balances both false positives and false negatives. Precision measures the proportion of true positive predictions out of all positive predictions, indicating the accuracy of positive classifications. Recall, on the other hand, measures the proportion of true positive predictions out of all actual positives, reflecting the model's ability to capture all relevant instances.

By incorporating these metrics, we could evaluate the model beyond mere accuracy, allowing us to identify areas where the model might be misclassifying data. This comprehensive evaluation was essential for understanding the model's performance in real-world scenarios, where precision and recall might be more critical than overall accuracy.

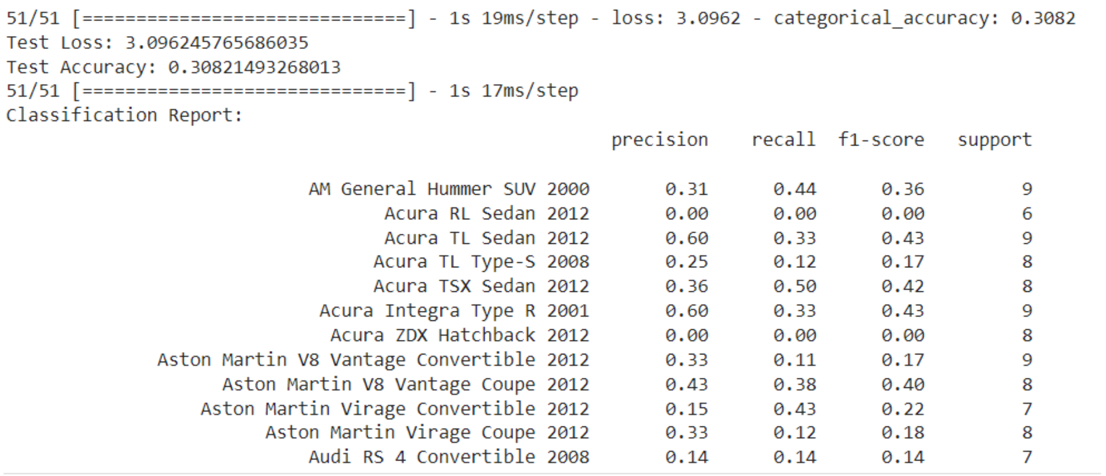
To add these metrics, custom functions were defined using TensorFlow's tf.keras.metrics API. During the model compilation, these custom metrics were included, ensuring that they were calculated and recorded at each epoch. The inclusion of F1-score, precision, and recall provided a more holistic view of the model's performance, guiding further refinements and optimizations.

Additionally, a detailed classification report is generated using the classification_report function from scikit-learn, which provides precision, recall, and F1-score for each class. This helps in understanding the model's performance across different categories.

Custom CNN Performance

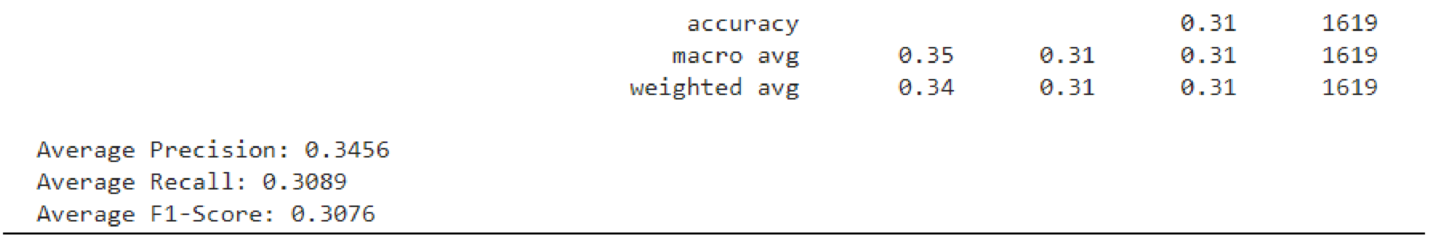
The performance of the basic CNN model on the Stanford Cars dataset was evaluated using several metrics, including precision, recall, F1-score, and accuracy. The classification report generated from the model's predictions on the test set revealed the following metrics:

- Average Precision: 0.3456
- Average Recall: 0.3089
- Average F1-Score: 0.3076
- Test Accuracy: 30.82%
- Test Loss: 3.09



```
51/51 [==============================] - 1s 19ms/step - loss: 3.0962 - categorical_accuracy: 0.3082
Test Loss: 3.096245765686035
Test Accuracy: 0.30821493268013
51/51 [==============================] - 1s 17ms/step
Classification Report:
                                                    precision    recall  f1-score   support

                         AM General Hummer SUV 2000       0.31      0.44      0.36         9
                                Acura RL Sedan 2012       0.00      0.00      0.00         6
                                Acura TL Sedan 2012       0.60      0.33      0.43         9
                               Acura TL Type-S 2008       0.25      0.12      0.17         8
                               Acura TSX Sedan 2012       0.36      0.50      0.42         8
                          Acura Integra Type R 2001       0.60      0.33      0.43         9
                          Acura ZDX Hatchback 2012       0.00      0.00      0.00         8
              Aston Martin V8 Vantage Convertible 2012       0.33      0.11      0.17         9
                    Aston Martin V8 Vantage Coupe 2012       0.43      0.38      0.40         8
                  Aston Martin Virage Convertible 2012       0.15      0.43      0.22         7
                        Aston Martin Virage Coupe 2012       0.33      0.12      0.18         8
                          Audi RS 4 Convertible 2008       0.14      0.14      0.14         7
```

Figure 26 Basic CNN Classification Report



```
                                          accuracy                            0.31      1619
                                         macro avg       0.35      0.31      0.31      1619
                                      weighted avg       0.34      0.31      0.31      1619

Average Precision: 0.3456
Average Recall: 0.3089
Average F1-Score: 0.3076
```

Figure 27 Basic CNN Performance Metrics

The plots for training and validation accuracy and loss further illustrate the model's performance:

- Training and Validation Accuracy: The accuracy plot shows a gradual increase in accuracy over the epochs, but it remained relatively low, indicating limited learning and generalization capabilities.
- Training and Validation Loss: The loss plot shows a steady decrease in loss over the epochs, but the high final loss value reflects the model's inadequate performance.

These findings emphasize the limitations of the basic CNN architecture in handling the complexity and diversity of the Stanford Cars dataset, which contains 196 different car classes with significant intra-class variability.

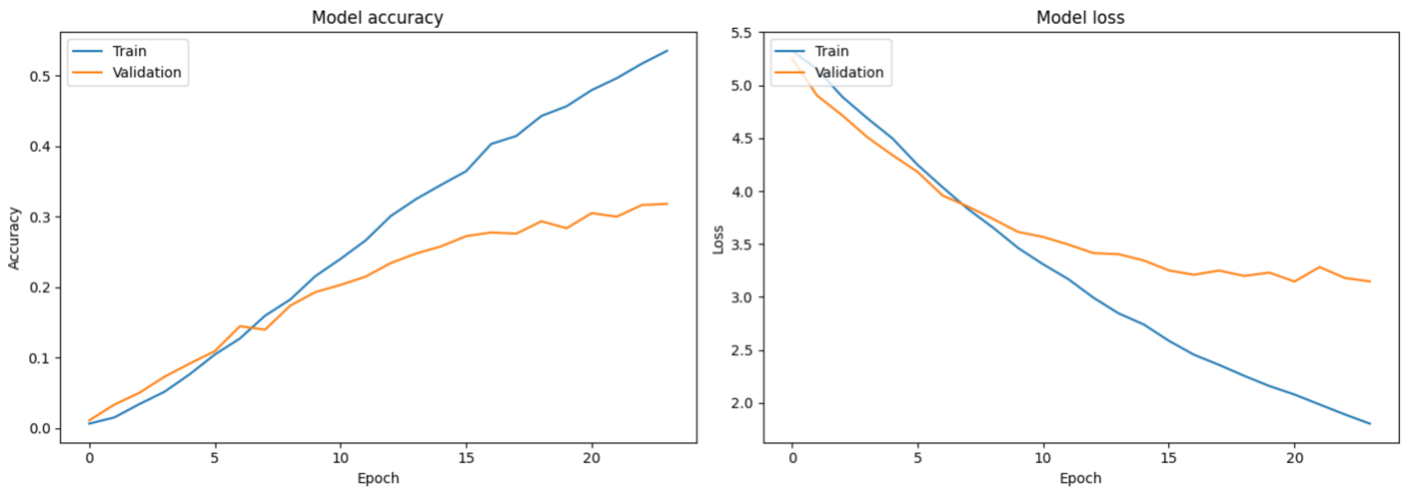
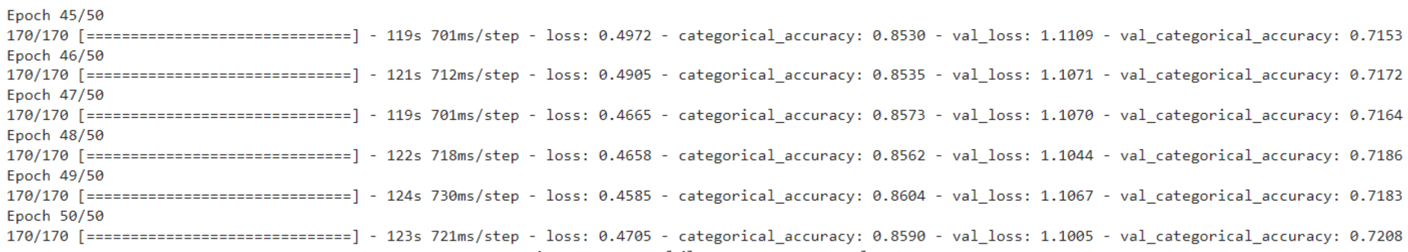


Figure 28 Basic CNN training and validation loss / accuracy

MobileNetV2 Performance



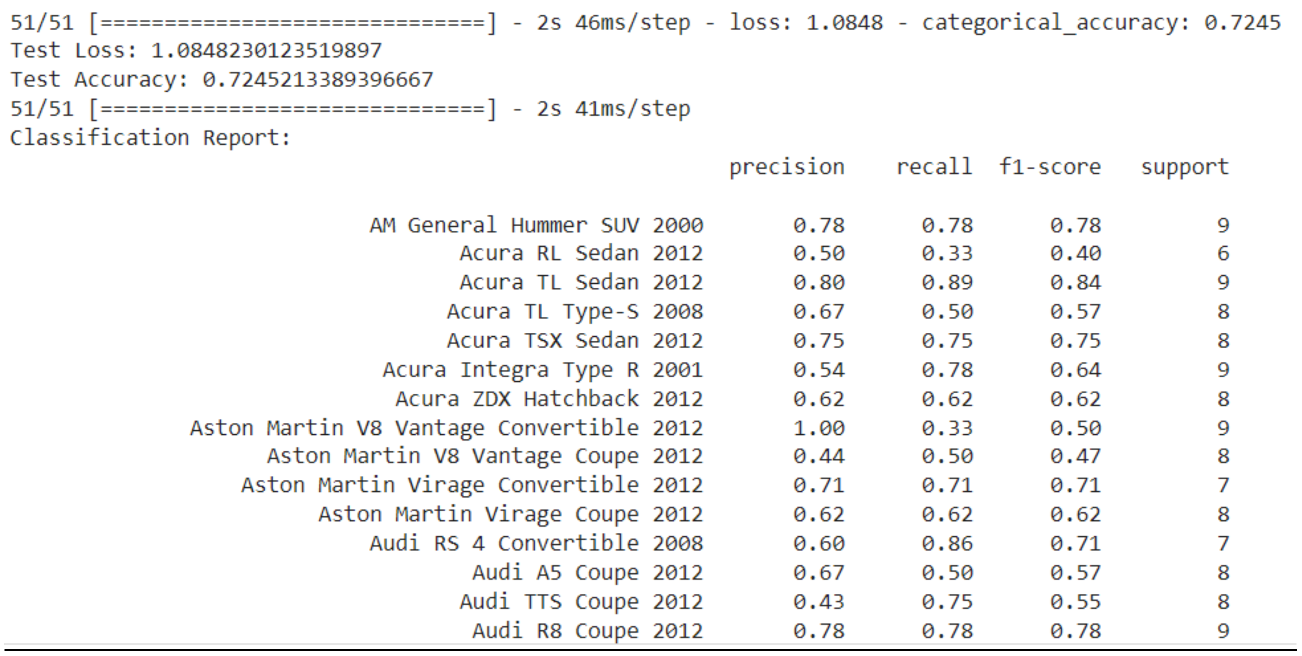
```
Epoch 45/50
170/170 [==============================] - 119s 701ms/step - loss: 0.4972 - categorical_accuracy: 0.8530 - val_loss: 1.1109 - val_categorical_accuracy: 0.7153
Epoch 46/50
170/170 [==============================] - 121s 712ms/step - loss: 0.4905 - categorical_accuracy: 0.8535 - val_loss: 1.1071 - val_categorical_accuracy: 0.7172
Epoch 47/50
170/170 [==============================] - 119s 701ms/step - loss: 0.4665 - categorical_accuracy: 0.8573 - val_loss: 1.1070 - val_categorical_accuracy: 0.7164
Epoch 48/50
170/170 [==============================] - 122s 718ms/step - loss: 0.4658 - categorical_accuracy: 0.8562 - val_loss: 1.1044 - val_categorical_accuracy: 0.7186
Epoch 49/50
170/170 [==============================] - 124s 730ms/step - loss: 0.4585 - categorical_accuracy: 0.8604 - val_loss: 1.1067 - val_categorical_accuracy: 0.7183
Epoch 50/50
170/170 [==============================] - 123s 721ms/step - loss: 0.4705 - categorical_accuracy: 0.8590 - val_loss: 1.1005 - val_categorical_accuracy: 0.7208
```

Figure 29 MobileNetV2 50 epochs

In contrast, the MobileNetV2 model, which leverages transfer learning, demonstrated significantly better performance on the same dataset. The classification report for MobileNetV2 provided the following metrics:

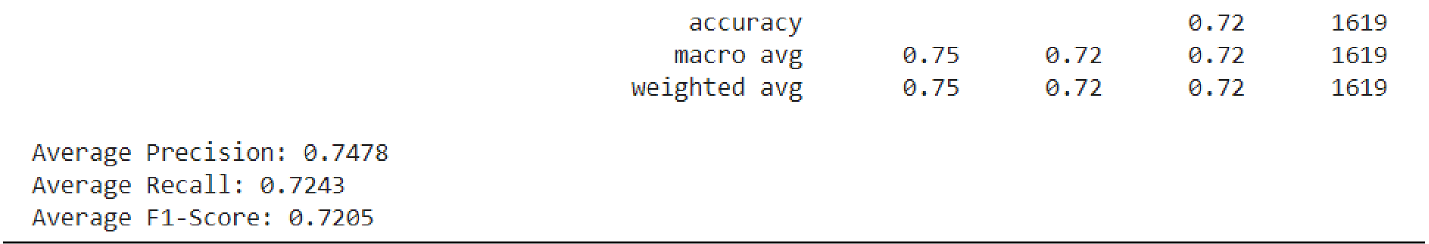
- Average Precision: 0.7478
- Average Recall: 0.7243

- Average F1-Score: 0.7205
- Test Accuracy: 72.45%
- Test Loss: 1.0848



```
51/51 [==============================] - 2s 46ms/step - loss: 1.0848 - categorical_accuracy: 0.7245
Test Loss: 1.0848230123519897
Test Accuracy: 0.7245213389396667
51/51 [==============================] - 2s 41ms/step
Classification Report:
                                                       precision    recall  f1-score   support

                    AM General Hummer SUV 2000       0.78      0.78      0.78         9
                           Acura RL Sedan 2012       0.50      0.33      0.40         6
                           Acura TL Sedan 2012       0.80      0.89      0.84         9
                          Acura TL Type-S 2008       0.67      0.50      0.57         8
                          Acura TSX Sedan 2012       0.75      0.75      0.75         8
                     Acura Integra Type R 2001       0.54      0.78      0.64         9
                     Acura ZDX Hatchback 2012        0.62      0.62      0.62         8
       Aston Martin V8 Vantage Convertible 2012       1.00      0.33      0.50         9
             Aston Martin V8 Vantage Coupe 2012       0.44      0.50      0.47         8
           Aston Martin Virage Convertible 2012       0.71      0.71      0.71         7
                 Aston Martin Virage Coupe 2012       0.62      0.62      0.62         8
                   Audi RS 4 Convertible 2008        0.60      0.86      0.71         7
                          Audi A5 Coupe 2012         0.67      0.50      0.57         8
                          Audi TTS Coupe 2012        0.43      0.75      0.55         8
                          Audi R8 Coupe 2012         0.78      0.78      0.78         9
```

Figure 30 MobileNetV2 Classification Report



```
                                accuracy                          0.72      1619
                               macro avg       0.75      0.72      0.72      1619
                            weighted avg       0.75      0.72      0.72      1619

Average Precision: 0.7478
Average Recall: 0.7243
Average F1-Score: 0.7205
```

Figure 31 MobileNetV2 Performance Metrics

The performance plots for MobileNetV2 highlight the model's efficiency:

- Training and Validation Accuracy: The accuracy plot indicates a rapid increase in accuracy, reaching high values early in the training process, and maintaining stability across epochs.
- Training and Validation Loss: The loss plot shows a significant reduction in loss, stabilizing at a much lower value compared to the basic CNN model, indicating better learning and generalization capabilities.

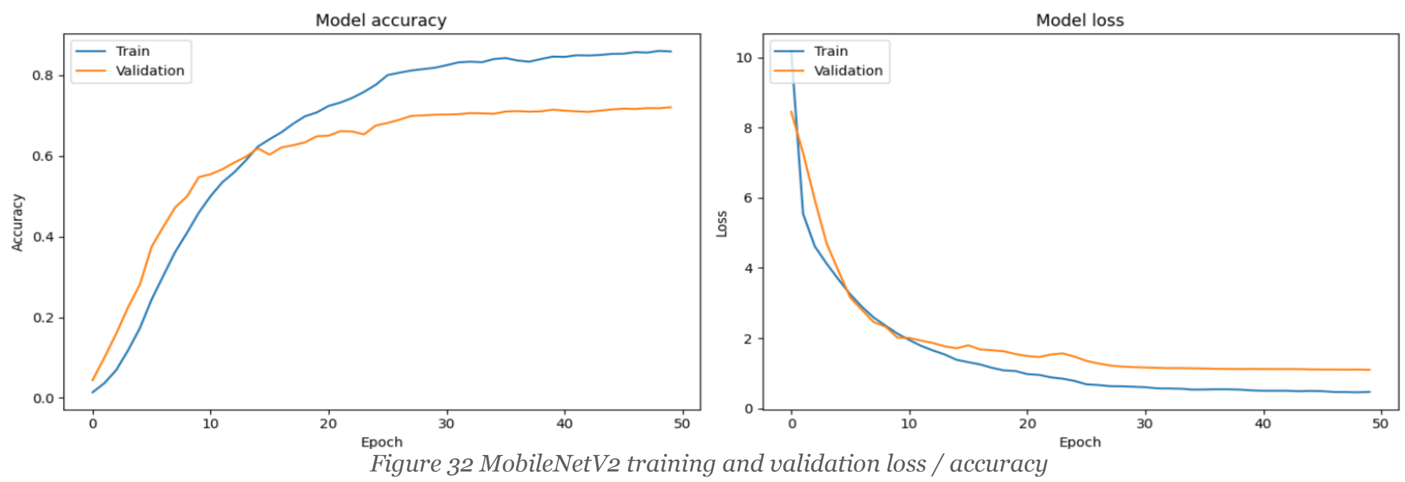
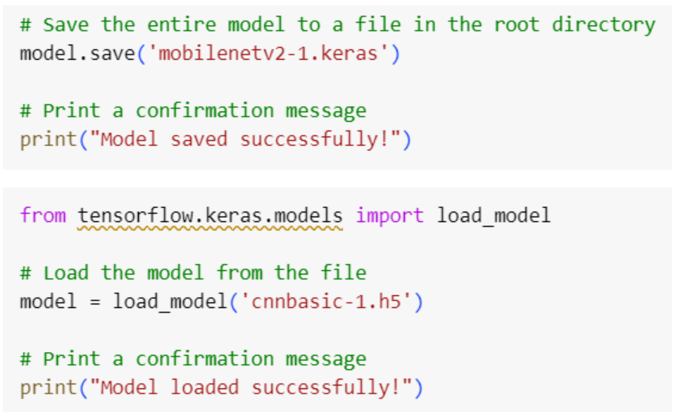


Figure 32 MobileNetV2 training and validation loss / accuracy

4.2. Testing

For the testing phase of the project, the implemented procedure ensures a thorough evaluation of the trained models. After training, each model is saved using the functionality provided in TensorFlow, enabling easy loading and re-evaluation at any time. This is particularly useful for testing and comparing different models without the need to retrain them from scratch.
The testing phase begins by loading the trained model using the load_model function from TensorFlow.

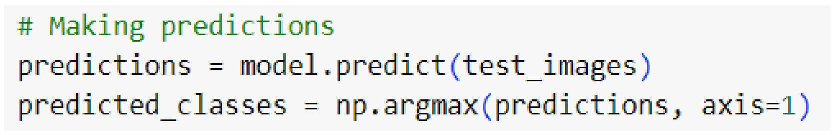


```python
# Save the entire model to a file in the root directory
model.save('mobilenetv2-1.keras')

# Print a confirmation message
print("Model saved successfully!")
```

```python
from tensorflow.keras.models import load_model

# Load the model from the file
model = load_model('cnnbasic-1.h5')

# Print a confirmation message
print("Model loaded successfully!")
```

Figure 33 Saving and Loading Model Functionality

Once the model is loaded, predictions are made on the test dataset using the predict method. The predicted class labels are obtained by taking the argmax of the prediction probabilities.



```python
# Making predictions
predictions = model.predict(test_images)
predicted_classes = np.argmax(predictions, axis=1)
```

Figure 34 Making predictions

To evaluate the performance of the models on individual test images, a function was implemented to display a set of randomly selected test images along with their actual and predicted labels. The function plot_images takes the test images, actual labels, predicted labels, and class names as input and plots the images in a grid format. The images are displayed with their actual and predicted class names, allowing for a visual assessment of the model's predictions.

Evaluation of Basic CNN vs. MobileNetV2

When tested on the test image set, the basic CNN model correctly predicted 3 out of 10 images, highlighting its struggle with accurate classification. In stark contrast, the MobileNetV2 model, leveraging transfer learning, correctly predicted 8 out of 10 images, demonstrating its superior capability in handling complex image classification tasks.
These results underline the significant performance gap between a basic CNN and a pre-trained model like MobileNetV2, showcasing the latter's effectiveness in achieving higher accuracy and reliability in predictions. The visual representation of predictions through the plotted images provides an intuitive understanding of how well the models perform in real-world scenarios.

MobileNetV2 Predictions

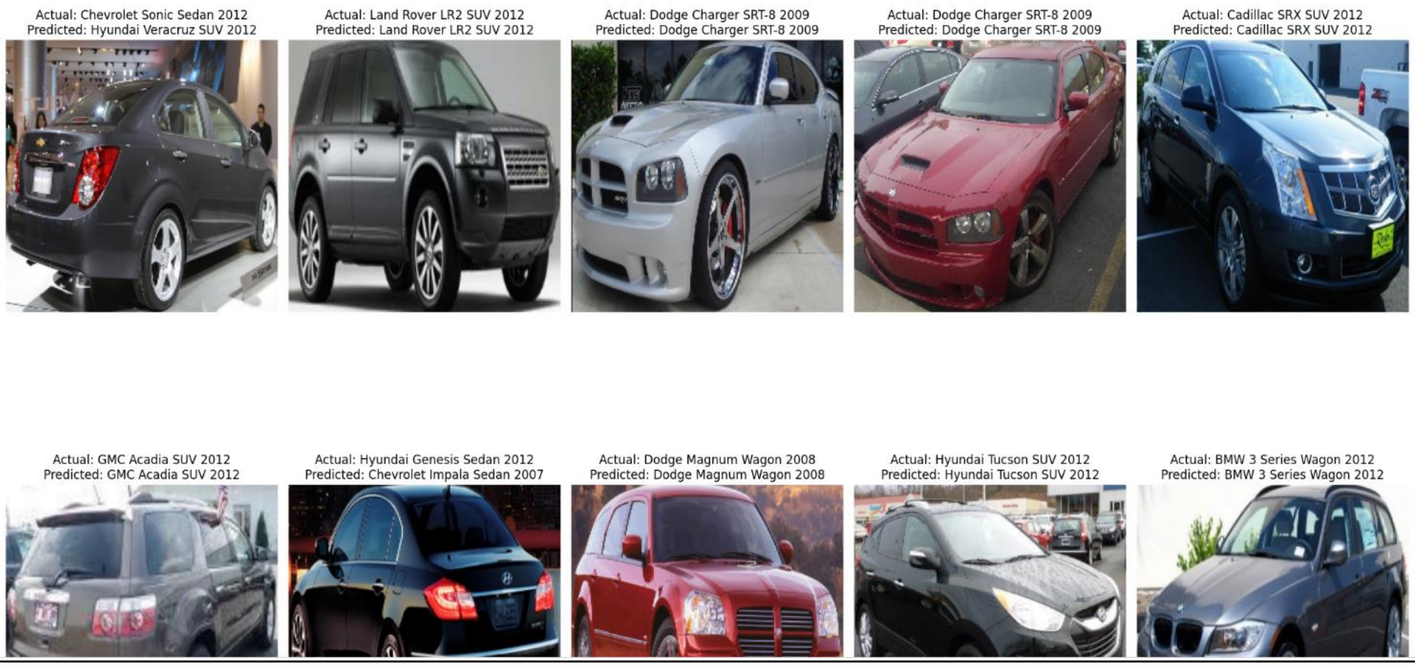

Actual: Chevrolet Sonic Sedan 2012
Predicted: Hyundai Veracruz SUV 2012
Actual: Land Rover LR2 SUV 2012
Predicted: Land Rover LR2 SUV 2012
Actual: Dodge Charger SRT-8 2009
Predicted: Dodge Charger SRT-8 2009
Actual: Dodge Charger SRT-8 2009
Predicted: Dodge Charger SRT-8 2009
Actual: Cadillac SRX SUV 2012
Predicted: Cadillac SRX SUV 2012
Actual: GMC Acadia SUV 2012
Predicted: GMC Acadia SUV 2012
Actual: Hyundai Genesis Sedan 2012
Predicted: Chevrolet Impala Sedan 2007
Actual: Dodge Magnum Wagon 2008
Predicted: Dodge Magnum Wagon 2008
Actual: Hyundai Tucson SUV 2012
Predicted: Hyundai Tucson SUV 2012
Actual: BMW 3 Series Wagon 2012
Predicted: BMW 3 Series Wagon 2012

Figure 35 MobileNetV2 Predictions

Basic CNN Predictions

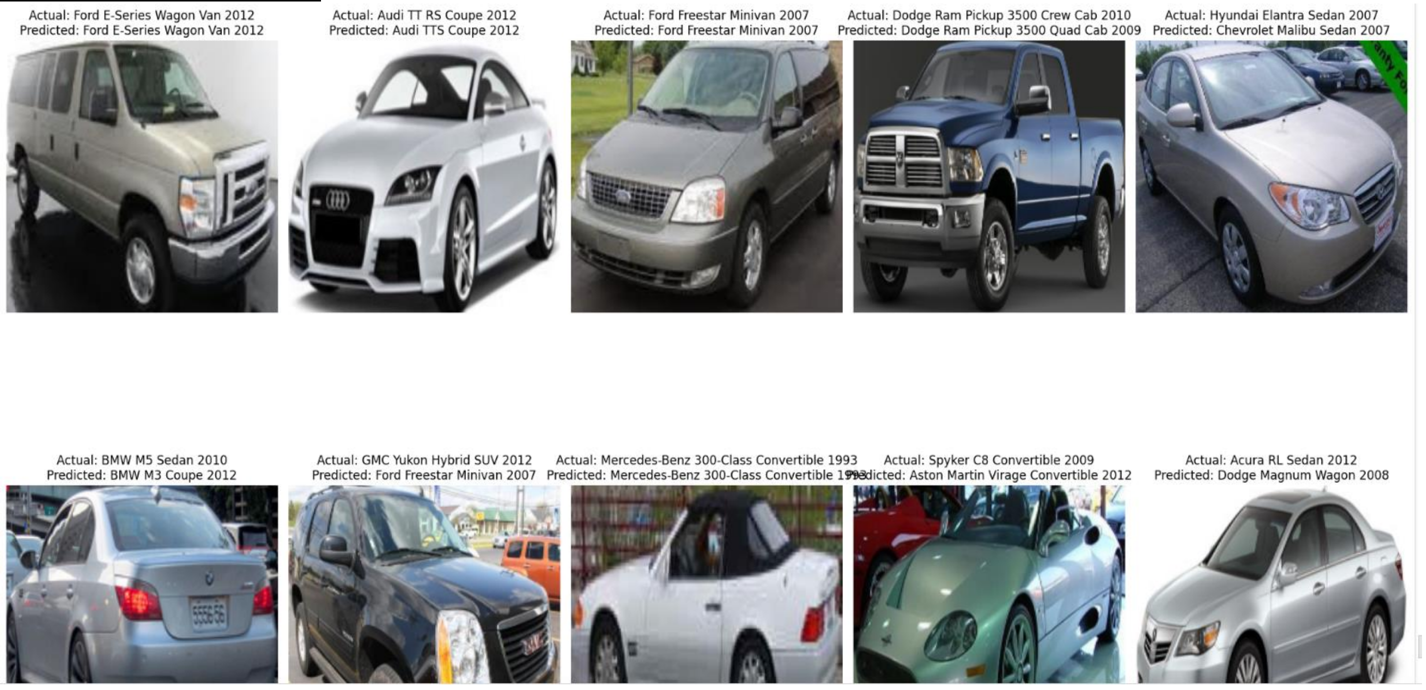

Actual: Ford E-Series Wagon Van 2012
Predicted: Ford E-Series Wagon Van 2012
Actual: Audi TT RS Coupe 2012
Predicted: Audi TTS Coupe 2012
Actual: Ford Freestar Minivan 2007
Predicted: Ford Freestar Minivan 2007
Actual: Dodge Ram Pickup 3500 Crew Cab 2010
Predicted: Dodge Ram Pickup 3500 Quad Cab 2009
Actual: Hyundai Elantra Sedan 2007
Predicted: Chevrolet Malibu Sedan 2007
Actual: BMW M5 Sedan 2010
Predicted: BMW M3 Coupe 2012
Actual: GMC Yukon Hybrid SUV 2012
Predicted: Ford Freestar Minivan 2007
Actual: Mercedes-Benz 300-Class Convertible 1993
Predicted: Mercedes-Benz 300-Class Convertible 1993
Actual: Spyker C8 Convertible 2009
Predicted: Aston Martin Virage Convertible 2012
Actual: Acura RL Sedan 2012
Predicted: Dodge Magnum Wagon 2008

Figure 36 Basic CNN Predictions

4.3. Analysis of results

Basic CNN Model Results

The basic CNN model achieved an average precision of 0.3456, an average recall of 0.3089, and an average F1-score of 0.3076. The overall accuracy of the model on the test set was 30.82%, with a test loss of 3.09. The categorical accuracy during training was also 30%, indicating that the model struggled to accurately classify a significant portion of the test images. This performance highlights the limitations of a simple CNN architecture in handling complex datasets like the Stanford Cars dataset, which contains a large number of classes (196) and significant intra-class variability.

Despite achieving a modest accuracy, the precision and recall values suggest that the model has difficulty in correctly identifying and recalling the correct car classes, resulting in a relatively low F1-score. The high test loss further underscores the model's inadequacy in learning from the data effectively. These results point to the need for more sophisticated techniques, such as transfer learning, to improve classification performance.

MobileNetV2 Model Results

In contrast, the MobileNetV2 model, which leverages transfer learning, significantly outperformed the basic CNN model. It achieved an average precision of 0.7478, an average recall of 0.7243, and an average F1-score of 0.7205. The overall test accuracy was 72.45%, with a test loss of 1.0848. The categorical accuracy during training was also 72.45%, indicating a substantial improvement over the basic CNN model.

The high precision and recall values suggest that MobileNetV2 is not only able to accurately identify the car classes but also recall a majority of the correct classes, resulting in a high F1-score. The lower test loss compared to the basic CNN model indicates that MobileNetV2 has a better learning capability and generalizes well to unseen data. The significant improvement in performance metrics highlights the efficacy of using pre-trained models and transfer learning in handling complex image classification tasks.

Comparative Analysis

The comparison between the basic CNN model and the MobileNetV2 model clearly demonstrates the advantages of using transfer learning with pre-trained models. The MobileNetV2 model's superior performance can be attributed to its more complex architecture and the pre-trained knowledge on a large and diverse dataset (ImageNet). This pre-trained knowledge provides a solid foundation for the model to learn and adapt to the specific task of car classification with greater accuracy and efficiency.

The results suggest that while a basic CNN model can serve as a starting point, more advanced techniques such as transfer learning are crucial for achieving higher performance in complex image classification tasks. The use of effective preprocessing, augmentation, and hyperparameter tuning further enhances the model's ability to generalize and perform well on new data.

4.4. Discussion

The project demonstrated that transfer learning with pre-trained models, such as MobileNetV2, significantly enhances performance in image classification tasks compared to a basic CNN model. The basic CNN model, although effective to some extent, lacked the sophistication and depth required to achieve high accuracy on a complex dataset like the Stanford Cars dataset. MobileNetV2, on the other hand, provided a robust framework built on pre-trained knowledge, which led to improved classification accuracy and overall performance.

Effective preprocessing, augmentation, and hyperparameter tuning proved to be crucial in optimizing model performance. Preprocessing steps, such as resizing images, normalizing pixel values, and applying data augmentation techniques, prepared the data effectively for training. Augmentation techniques like random rotations, width and height shifts, and horizontal flips helped increase the diversity of the training data, thereby enhancing the model's ability to generalize.

Hyperparameter tuning using Keras Tuner allowed for systematic exploration of different configurations, leading to the identification of the best model parameters. This process was instrumental in fine-tuning the model to achieve optimal performance. The combination of these strategies ensured that the model was well-prepared to handle the variability in the dataset and improved its ability to accurately classify images.

Transfer learning, in particular, showcased its strength by leveraging pre-trained knowledge from large datasets. This approach significantly reduced the training time and improved the model's performance by starting from a point of advanced learning rather than from scratch. The pre-trained MobileNetV2 model exhibited superior classification accuracy, highlighting the benefits of transfer learning in complex image classification tasks.

Future Work

Future work could involve experimenting with various other model architectures to further enhance performance. Exploring more advanced convolutional neural networks or novel architectures might yield even better results.

Investigating more sophisticated augmentation techniques could help improve the model's robustness and ability to generalize. Techniques such as random erasing, cutout, and mixup could be explored to see their impact on performance.

Developing strategies to handle class imbalance more effectively is another area for improvement. Techniques like oversampling, undersampling, or using synthetic data generation methods such as SMOTE (Synthetic Minority Over-sampling Technique) could be beneficial.

Additional Research

Exploring Transfer Learning with Other Pre-trained Models: Future research could explore the impact of using different pre-trained models for transfer learning. Models like ResNet, Inception, and EfficientNet might offer additional performance gains and should be investigated.

Studying the effects of various hyperparameter configurations on model performance could provide deeper insights. Systematic tuning and evaluation of learning rates, batch sizes, and other parameters could help identify the optimal settings for different datasets and tasks.

By addressing these areas, the project can be further refined, leading to even more performant and accurate image classification models. The ongoing research and experimentation in these domains will continue to push the boundaries of what is achievable in image classification tasks.

5. Conclusion

In conclusion, this project successfully demonstrated the effectiveness of transfer learning in image classification tasks using the Stanford Cars dataset. By comparing a basic Convolutional Neural Network (CNN) model with a pre-trained MobileNetV2 model, we highlighted the significant performance improvements achievable through transfer learning. The basic CNN model achieved a modest accuracy of 30.82%, while the MobileNetV2 model achieved a substantially higher accuracy of 72.45%. The comprehensive data preparation, including renaming images, augmenting data, and ensuring data integrity, combined with effective preprocessing techniques like resizing and normalizing images, were crucial in optimizing the model's performance. The use of advanced augmentation techniques, careful hyperparameter tuning, and leveraging pre-trained models significantly enhanced the classification accuracy and robustness of the models. This project underscores the importance of using sophisticated models and techniques in handling complex and diverse datasets, paving the way for future enhancements through experimenting with various architectures and augmentation strategies.