Computer Vision

OBJECT DETECTION - CAR

A Capstone Project

Made by AIML April Capstone Group 8 Members:

Goutam Kalita

Haribasker

Saif Merchant

Shanmugabharathy

Ruhee Seshadri

In fulfilment of the requirements for the award of

Post Graduate Program in

Artificial Intelligence & Machine Learning

from:

Great Learning

&

TEXAS McCombs

The University of Texas at Austin

Mentor: *Aniket Chhabra*

CONTENTS

Interim Report: Car Classification Computer Vision Project

Introduction :

Computer Vision, enhanced by Artificial Intelligence and Machine Learning, has ushered in a new era of innovation across various industries. This interdisciplinary field empowers machines to comprehend and interpret visual information much like human perception of images and videos. The integration of computer vision with AI and ML techniques has paved the way for automation, intelligent decision-making, and advanced analysis based on visual data.

In the dynamic realm of technology, computer vision, supported by Artificial Intelligence (AI) and Machine Learning (ML), has become a transformative force. This synergy enables machines to interpret and analyze visual data, making informed decisions akin to human perception. The significance of computer vision spans across diverse sectors, including healthcare, automotive, surveillance, and more. AIML projects in computer vision leverage algorithms to process visual information, opening new horizons in automation, data analysis, and intelligent decision-making. As we delve into this multidisciplinary field, we embark on a journey to explore the applications, methodologies, and outcomes of computer vision AIML projects, unravelling their potential in shaping the future of technology and innovation.

The project at hand focuses on car detection as one of its applications. The givens are listed below:

Problem Statement

Domain: Automotive Surveillance

Context:

In the realm of automotive surveillance, the implementation of computer vision technologies offers the opportunity to automate supervision and trigger appropriate actions based on image predictions. One such application is the identification of cars on the road, encompassing details like make, type, colour, and license plate information.

Data Description:

The Cars dataset comprises 16,185 images distributed across 196 classes of cars. The dataset is divided into 8,144 training images and 8,041 testing images, with each

class having an approximately 50-50 split. Classifications are at a detailed level, including Make, Model, and Year (e.g., 2012 Tesla Model S or 2012 BMW M3 coupe).
Data Components:
Train Images: Real images of cars, categorized by make and year.
Test Images: Real images of cars, categorized by make and year.
Train Annotation: Bounding box regions for training images.
Test Annotation: Bounding box regions for testing images.

Dataset provided with this project; original link for reference: Stanford Car Dataset
Reference: 3D Object Representations for Fine-Grained Categorization, Jonathan Krause, Michael Stark, Jia Deng, Li Fei-Fei, 4th IEEE Workshop on 3D Representation and Recognition, ICCV 2013 (3dRR-13), Sydney, Australia, Dec. 8, 2013.

Step 1: Import the data:

In Step 1, the necessary libraries and tools have been imported to facilitate the subsequent processes in the car classification computer vision project. Notable libraries include NumPy, Plotly, OpenCV, Pillow, TensorFlow, and others, which are essential for data manipulation, visualization, and the implementation of deep learning models.
The inclusion of warning suppression ensures a clean runtime environment. The next steps will build upon this foundation to handle data, visualize images, and develop and train the initial Convolutional Neural Network (CNN) models for car classification.

```
import numpy as np
import plotly.express as px
import os
import zipfile
import cv2
from PIL import Image
from IPython.display import display
import ipywidgets as widgets
from matplotlib import pyplot as plt
import matplotlib.patches as patches
from collections import defaultdict
from io import StringIO
from PIL import Image
import random
from IPython.display import display
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
import warnings
import pickle
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os
import plotly.graph_objs as go
from tensorflow.keras.models import load_model
from tensorflow.keras.layers import Dropout
from tensorflow.keras.callbacks import EarlyStopping
warnings.filterwarnings("ignore")
```

  - A library for interactive data visualization.
- **OS (import os):**
  - Provides a way of interacting with the operating system, essential for file and directory operations.

- **Zipfile (import zipfile):**
  - Enables the extraction and manipulation of zip files.
- **OpenCV (import cv2):**
  - Open Source Computer Vision Library, used for image processing tasks.
- **Pillow (from PIL import Image):**
  - A Python Imaging Library that adds image processing capabilities.
- **IPython Display (from IPython.display import display):**
  - Enables the display of rich media representations in the Jupyter Notebook.
- **ipywidgets (import ipywidgets as widgets):**
  - Interactive HTML widgets for Jupyter Notebooks.
- **Matplotlib (from matplotlib import pyplot as plt):**
  - A comprehensive library for creating static, animated, and interactive visualizations in Python.
- **Defaultdict (from collections import defaultdict):**
  - Provides a default value for nonexistent keys in a dictionary.
- **StringIO (from io import StringIO):**
  - Implements an in-memory file-like object for string data.
- **Random (import random):**
  - Generates pseudo-random numbers.
- **Pandas (import pandas as pd):**
  - A powerful data manipulation library.
- **Seaborn (import seaborn as sns):**
  - Built on top of Matplotlib, provides a high-level interface for drawing attractive and informative statistical graphics.
- **Warnings (import warnings):**
  - Allows the control of warning messages in Python.
- **Pickel (import pickle):**
  - Serializes and deserializes Python objects, facilitating the saving and loading of data.
- **TensorFlow Keras (from tensorflow.keras.models import Sequential):**
  - The high-level neural networks API, part of TensorFlow, used for building and training deep learning models.
- **TensorFlow Keras Layers (from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout):**
  - Specific layers required for building convolutional neural networks.
- **ImageDataGenerator (from tensorflow.keras.preprocessing.image import ImageDataGenerator):**
  - Generates augmented batches of images during the model training process.
- **Plotly Graph Objects (import plotly.graph_objs as go):**
  - Constructs graph objects for interactive visualizations using Plotly.
- **Load Model (from tensorflow.keras.models import load_model):**
  - Loads a pre-existing deep learning model.
- **Early Stopping (from tensorflow.keras.callbacks import EarlyStopping):**

- Implements early stopping during model training based on a specified criterion.

Step 1.2: Data Extraction

In Step 1. 2, the code handles the extraction of data from two zip files ('Car+Images.zip' and 'Annotations.zip'). It ensures the creation of a designated destination folder and extracts the contents of each zip file into separate subfolders within the destination. This step prepares the data for further processing and analysis in the car classification computer vision project.

```
car_data_zip_path = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/Car+Images.zip'
annotation_zip_path = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/Annotations.zip'

destination_folder = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites'

if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

with zipfile.ZipFile(car_data_zip_path, 'r') as car_data_zip:
    car_data_zip.extractall(os.path.join(destination_folder, 'car_data_extracted'))

with zipfile.ZipFile(annotation_zip_path, 'r') as annotation_zip:
    annotation_zip.extractall(os.path.join(destination_folder, 'annotations_extracted'))

print("Data extracted and stored in the destination folder.")
```

Step 1. 3: Load Car Model Data

```
carmodel_file = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/Car+names+and+make.cs
car_data = pd.read_csv(carmodel_file)
print("Car Data:")
print(car_data.head())
```

```
Car Data:
  AM General Hummer SUV 2000
0        Acura RL Sedan 2012
1        Acura TL Sedan 2012
2      Acura TL Type-S 2008
3       Acura TSX Sedan 2012
4  Acura Integra Type R 2001
```

In Step 1.3, the code reads and loads car model data from the 'Car+names+and+make.csv' file using the Pandas library. The loaded data is then displayed to verify its structure and content. This step is crucial for understanding the available car model information, which will be used in subsequent stages of the car classification computer vision project.

Step 2: Process Images and Create Data Frame:

```python
def process_images(folder_path):
    data = []

    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp')):
                car_name = os.path.basename(root)
                car_model = car_name.split()[-1]
                car_model_1 = ' '.join(car_name.split()[:-1])
                image_name = file
                image_location = os.path.join(root, file)

                data.append({
                    'carName': car_name,
                    'carModel': car_model,
                    'carModel_1': car_model_1,
                    'Image Name': image_name,
                    'Image Location': image_location
                })

    return data

# Defining the path to the main folder containing subfolders with images
Train_image_folder = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/car_data_extract

# Processing images recursively
Train_data = process_images(Train_image_folder)

# Creating a DataFrame
Train_data_df = pd.DataFrame(Train_data)

# Displaying the final DataFrame with image locations
Train_data_df.head()
```

In Step 2, images from the training data folder are processed using the **process_images** function, extracting relevant information about each image. This information is then organized into a DataFrame (**Train_data_df**) for better representation and analysis. The displayed DataFrame provides an initial overview of the processed data.

Result:

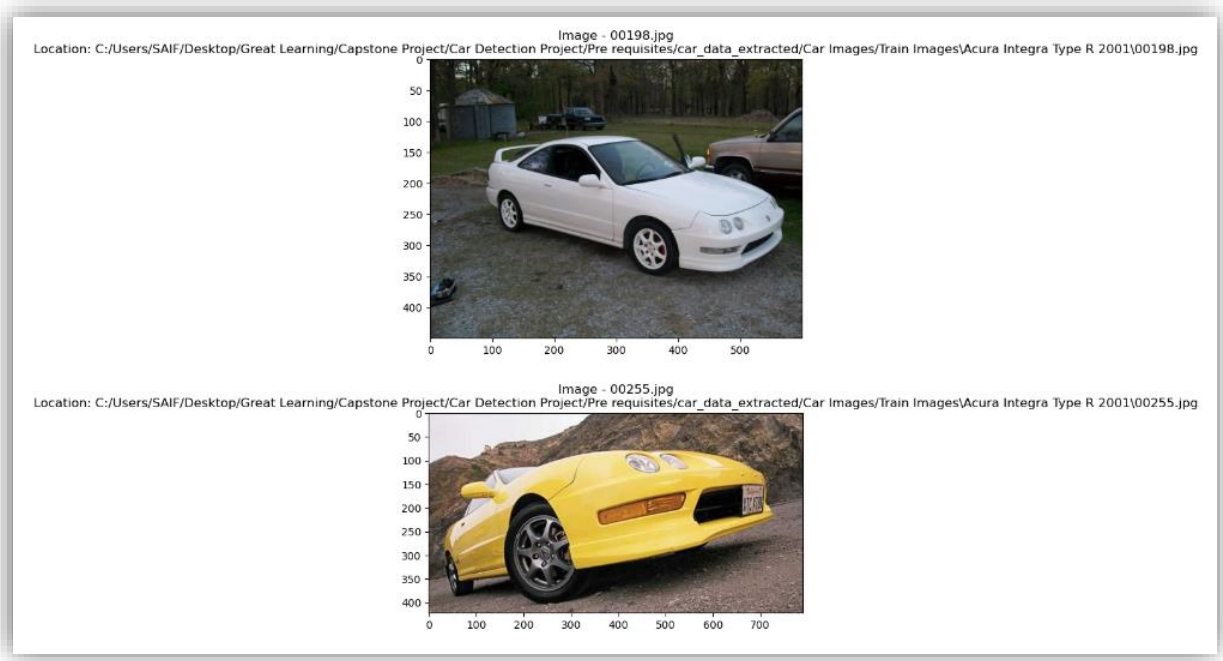
	carName	carModel	carModel_1	Image Name	Image Location
0	Acura Integra Type R 2001	2001	Acura Integra Type R	00198.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...
1	Acura Integra Type R 2001	2001	Acura Integra Type R	00255.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...
2	Acura Integra Type R 2001	2001	Acura Integra Type R	00308.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...
3	Acura Integra Type R 2001	2001	Acura Integra Type R	00374.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...
4	Acura Integra Type R 2001	2001	Acura Integra Type R	00878.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...

Let’s visualize few images and perform EDA:

This step involves a loop that iterates through the first five rows of the **Train_data_df and Test_data_df** DataFrame and displays some images along with relevant information. This step allows for visual inspection of the training data images, providing a qualitative understanding of the dataset.

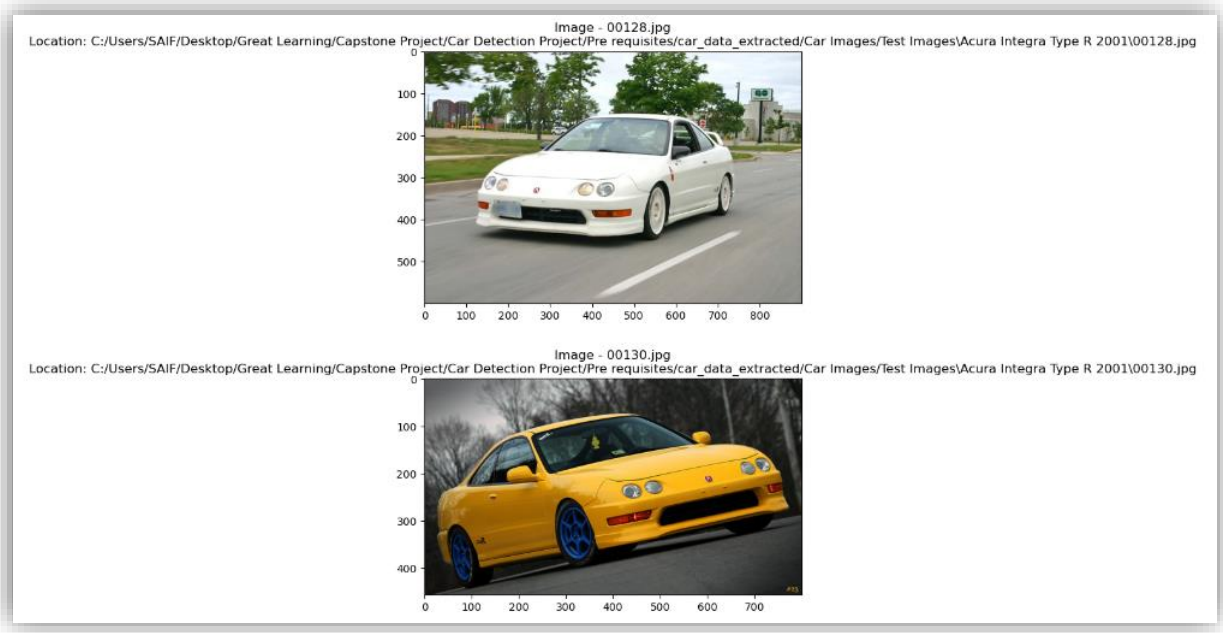
A: Train Images

```
for index, row in Train_data_df.head(5).iterrows():
    image_path = row['Image Location']
    img = Image.open(image_path)
    plt.imshow(img)
    plt.title(f'Image - {row["Image Name"]}\nLocation: {row["Image Location"]}')
    plt.show()
```



B: Test Images

```
for index, row in Test_data_df.head(5).iterrows():
    image_path = row['Image Location']
    img = Image.open(image_path)
    plt.imshow(img)
    plt.title(f'Image - {row["Image Name"]}\nLocation: {row["Image Location"]}')
    plt.show()
```



Step 3 - Map training and testing images to its annotations:

In this step , annotations for the training data are loaded and processed. The column names are renamed for consistency and clarity, preparing the annotations for further use in the car classification computer vision project.

```
In [22]: train_annotations = pd.read_csv("C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/annot
         test_annotations = pd.read_csv("C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/annota
```

Before renaming :

	Image Name	Bounding Box coordinates	Unnamed: 2	Unnamed: 3	Unnamed: 4	Image class
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
...						

```
train_annotations.rename(columns = {'Bounding Box coordinates':'xmin'}, inplace = True)
train_annotations.rename(columns = {'Unnamed: 2':'ymin'}, inplace = True)
train_annotations.rename(columns = {'Unnamed: 3':'xmax'}, inplace = True)
train_annotations.rename(columns = {'Unnamed: 4':'ymax'}, inplace = True)
train_annotations.rename(columns = {'Image class':'Image_class'}, inplace = True)
train_annotations.rename(columns = {'Image Name':'Image Name'}, inplace = True)
train_annotations
```

After renaming:

	Image Name	xmin	ymin	xmax	ymax	Image_class
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
...	...	...	...	...	...	...

Similarly for Test data also:

In [29]:
```
test_annotations.rename(columns = {'Bounding Box coordinates':'xmin'}, inplace = True)
test_annotations.rename(columns = {'Unnamed: 2':'ymin'}, inplace = True)
test_annotations.rename(columns = {'Unnamed: 3':'xmax'}, inplace = True)
test_annotations.rename(columns = {'Unnamed: 4':'ymax'}, inplace = True)
test_annotations.rename(columns = {'Image class':'Image_class'}, inplace = True)
test_annotations.rename(columns = {'Image Name':'Image Name'}, inplace = True)
test_annotations
```

Out[29]:

	Image Name	xmin	ymin	xmax	ymax	Image_class
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

Now, Let's merge the DataFrame of annotations and image DataFrame for the train data:

The training and testing data DataFrames are merged with their respective annotations DataFrames. This step consolidates the information from both sources, aligning the image data with bounding box annotations. The resulting **Train_final_df** and **Test_final_df** DataFrames are essential for training and evaluating the car classification model.

TRAIN_DATAFRAME:

```
In [30]: Train_final_df = pd.merge(Train_data_df, train_annotations, how='inner',left_on='Image Name', right_on='Image Name')
```

```
In [31]: Train_final_df.head(20)
```

Out[31]:

	carName	carModel	carModel_1	Image Name	Image Location	xmin	ymin	xmax	ymax	Image_class
0	Acura Integra Type R 2001	2001	Acura Integra Type R	00198.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	91	121	574	357	6
1	Acura Integra Type R 2001	2001	Acura Integra Type R	00255.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	29	78	734	396	6
2	Acura Integra Type R 2001	2001	Acura Integra Type R	00308.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	16	136	775	418	6
3	Acura Integra Type R 2001	2001	Acura Integra Type R	00374.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	9	184	740	499	6
4	Acura Integra Type R 2001	2001	Acura Integra Type R	00878.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	101	162	882	650	6
5	Acura Integra Type R 2001	2001	Acura Integra Type R	00898.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	89	68	483	310	6

TEST_DATAFRAME:

```
In [35]: Test_final_df = pd.merge(Test_data_df, test_annotations,how='inner', left_on='Image Name', right_on='Image Name')
```

```
In [36]: Test_final_df.head(20)
```

Out[36]:

	carName	carModel	carModel_1	Image Name	Image Location	xmin	ymin	xmax	ymax	Image_class
0	Acura Integra Type R 2001	2001	Acura Integra Type R	00128.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	122	149	743	455	6
1	Acura Integra Type R 2001	2001	Acura Integra Type R	00130.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	27	74	774	449	6
2	Acura Integra Type R 2001	2001	Acura Integra Type R	00386.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	86	184	723	425	6
3	Acura Integra Type R 2001	2001	Acura Integra Type R	00565.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	15	1	545	347	6
4	Acura Integra Type R 2001	2001	Acura Integra Type R	00711.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	126	65	735	347	6
5	Acura Integra Type R 2001	2001	Acura Integra Type R	01002.jpg	C:/Users/SAIF/Desktop/Great Learning/Capstone ...	54	225	499	610	6

Let's have a look at the sanity of the data and take appropriate steps if necessary:

In this step we performed data inspection and analyzeed the class distribution in the training and testing datasets. Information about the structure of the DataFrames is displayed, unique car models are identified, and class percentages are calculated and presented. This step helps in understanding the dataset characteristics and class distribution, which is crucial for model training and evaluation in the car classification computer vision project.

```
In [38]: Train_final_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 8144 entries, 0 to 8143
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   carName         8144 non-null   object
 1   carModel        8144 non-null   object
 2   carModel_1      8144 non-null   object
 3   Image Name      8144 non-null   object
 4   Image Location  8144 non-null   object
 5   xmin            8144 non-null   int64
 6   ymin            8144 non-null   int64
 7   xmax            8144 non-null   int64
 8   ymax            8144 non-null   int64
 9   Image_class     8144 non-null   int64
dtypes: int64(5), object(5)
memory usage: 699.9+ KB
```

```
In [39]: Test_final_df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 8041 entries, 0 to 8040
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   carName         8041 non-null   object
 1   carModel        8041 non-null   object
 2   carModel_1      8041 non-null   object
 3   Image Name      8041 non-null   object
 4   Image Location  8041 non-null   object
 5   xmin            8041 non-null   int64
 6   ymin            8041 non-null   int64
 7   xmax            8041 non-null   int64
 8   ymax            8041 non-null   int64
 9   Image_class     8041 non-null   int64
dtypes: int64(5), object(5)
memory usage: 691.0+ KB
```

Train Data Class Percentages:
119 0.834971
79 0.601670
167 0.589391
161 0.589391
144 0.577112
...
175 0.380648
64 0.368369
158 0.356090
99 0.343811
136 0.294695

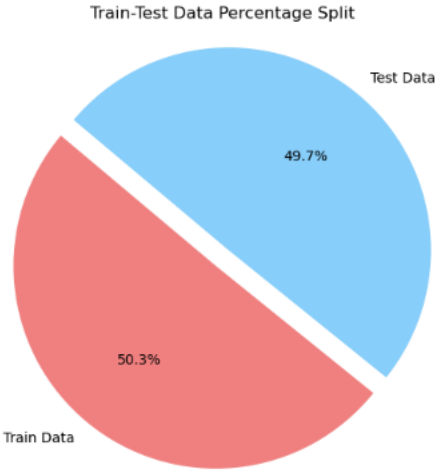
Test Data Class Percentages:
119 0.845666
161 0.596941
79 0.596941
167 0.584504
43 0.572068
...
175 0.373088
158 0.360652
64 0.360652
99 0.335779
136 0.298470

Let us explore the data by plotting few visualizations and performing EDA:

1 : Test data VS Train Data:

The dataset has been split into training and testing sets with approximately 49.7% allocated to the training data and 50.3% to the testing data. This balanced split ensures a representative distribution of data for both training and evaluation, enhancing the model's generalization capabilities.

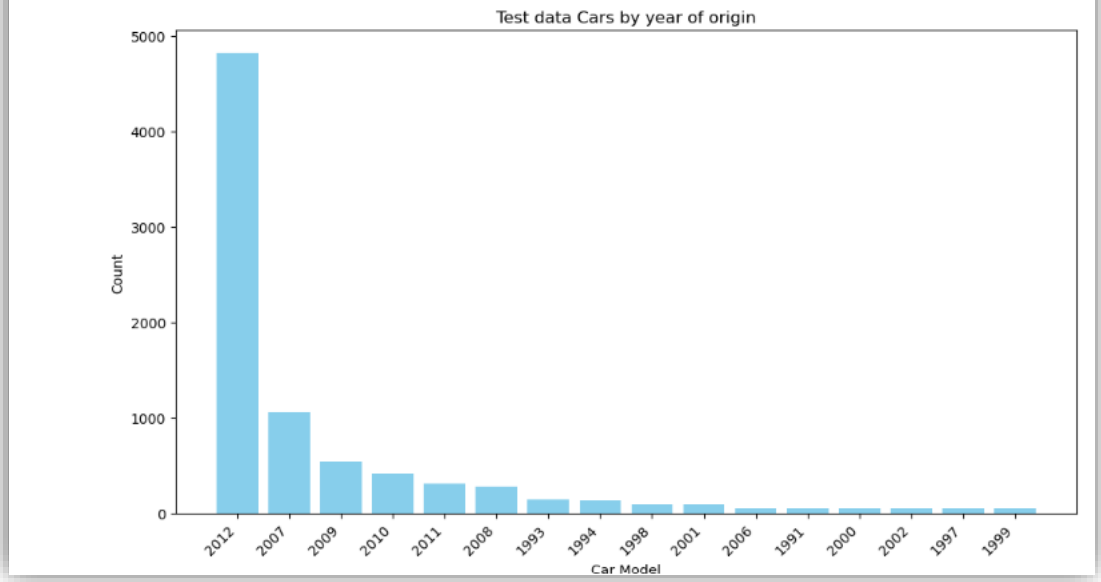
```
In [111]: total_train_rows = Train_final_df.shape[0]
          total_test_rows = Test_final_df.shape[0]
          train_percentage = (total_train_rows / (total_train_rows + total_test_rows)) * 100
          test_percentage = (total_test_rows / (total_train_rows + total_test_rows)) * 100

          sizes = [total_train_rows, total_test_rows]
          labels = ['Train Data', 'Test Data']
          colors = ['lightcoral', 'lightskyblue']
          explode = (0.1, 0)

          plt.figure(figsize=(8, 6))
          plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', startangle=140)
          plt.title('Train-Test Data Percentage Split')
          plt.axis('equal')
          plt.show()
```



2 : Cars by year of their origin:

The code calculates and visualizes the distribution of cars in the training data based on the year of their origin. The bar chart provides insights into the representation of different car models, contributing to the understanding of the dataset's characteristics in the car classification computer vision project.

```
In [113]: plt.figure(figsize=(10, 6))
plt.bar(carModel_counts['carModel'], carModel_counts['Count'], color='skyblue')
plt.xlabel('Car Model')
plt.ylabel('Count')
plt.title('Test data Cars by year of origin')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

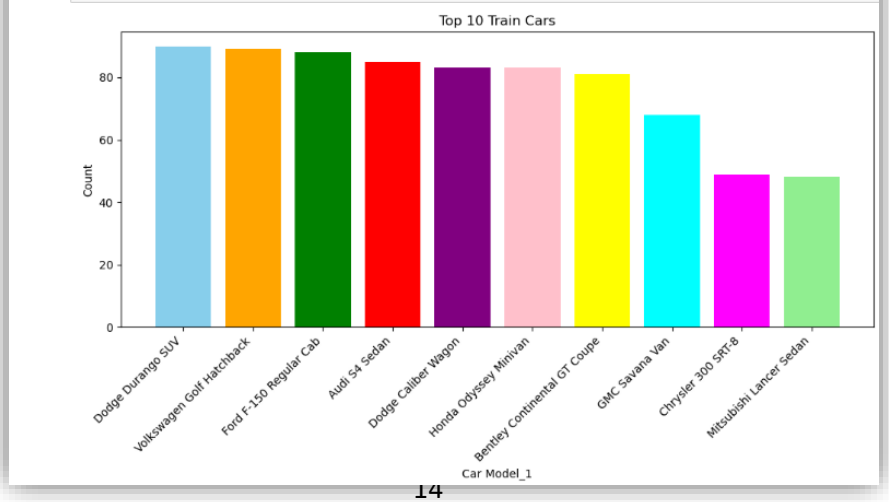


3. Count of top 10

In this step we calculates and visualizes the top 10 car models in the training data based on the 'carModel_1' column. The bar chart offers a clear representation of the most prevalent car models, aiding in understanding the dataset's composition and emphasizing the importance of specific car models in the car classification computer vision project.

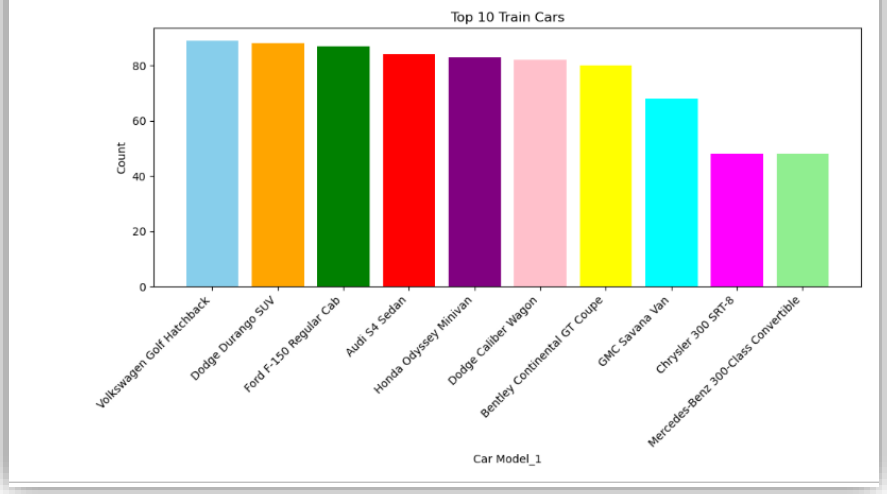
Train dataframe:

```
In [114]: top_10_carModel_1 = Train_final_df['carModel_1'].value_counts().nlargest(10)
bar_colors = ['skyblue', 'orange', 'green', 'red', 'purple', 'pink', 'yellow', 'cyan', 'magenta', 'lightgreen']
plt.figure(figsize=(10, 6))
plt.bar(top_10_carModel_1.index, top_10_carModel_1.values, color=bar_colors)
plt.xlabel('Car Model_1')
plt.ylabel('Count')
plt.title('Top 10 Train Cars')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
```



Test dataframe:

```
In [115]: top_10_carModel_1 = Test_final_df['carModel_1'].value_counts().nlargest(10)
          bar_colors = ['skyblue', 'orange', 'green', 'red', 'purple', 'pink', 'yellow', 'cyan', 'magenta', 'lightgreen']
          plt.figure(figsize=(10, 6))
          plt.bar(top_10_carModel_1.index, top_10_carModel_1.values, color=bar_colors)
          plt.xlabel('Car Model_1')
          plt.ylabel('Count')
          plt.title('Top 10 Train Cars')
          plt.xticks(rotation=45, ha='right')
          plt.tight_layout()
          plt.show()
```

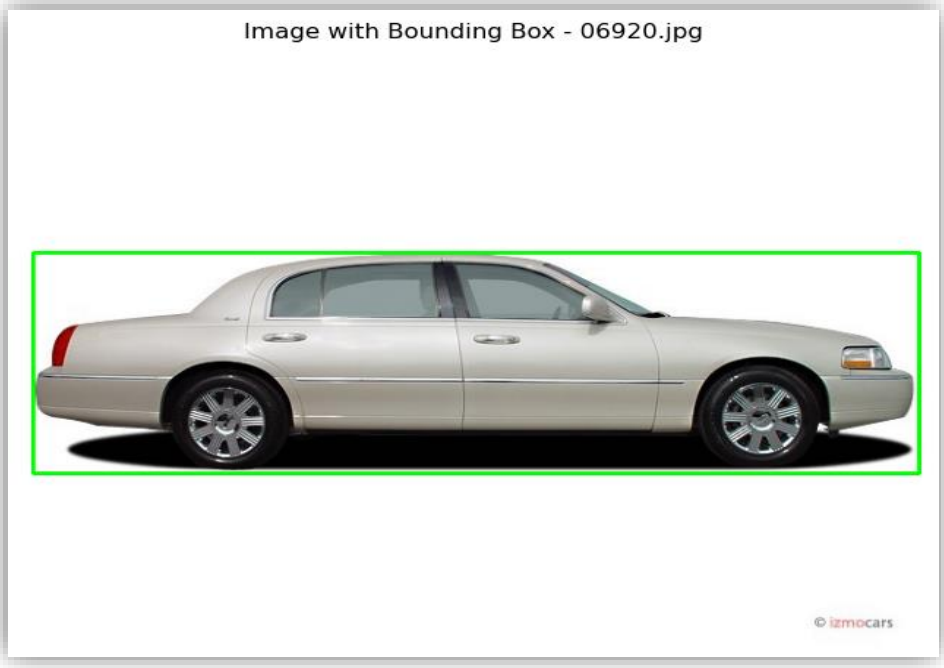


Step 4 : Display images with bounding box:

In this phase, we successfully implemented the display of images with bounding boxes, providing a visual understanding of our training dataset. The displayed images showcase the integration of bounding box coordinates, allowing us to observe the localization of cars within the images.

```
In [122]: def display_image_with_bounding_box(img_num):
              img_path = Train_data_df.loc[img_num, 'Image Location']
              img = cv2.imread(img_path)

              xmin = int(Train_final_df.loc[img_num, 'xmin'])
              ymin = int(Train_final_df.loc[img_num, 'ymin'])
              xmax = int(Train_final_df.loc[img_num, 'xmax'])
              ymax = int(Train_final_df.loc[img_num, 'ymax'])

              cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)

              car_model = Train_data_df.loc[img_num, 'carModel_1']

              img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

              plt.figure(figsize=(8, 6))
              plt.imshow(img)
              plt.title(f'Image with Bounding Box - {Train_data_df.loc[img_num, "Image Name"]}')
              plt.axis('off')  # Remove axis
              plt.show()

              print(f'Coordinates: xmin={xmin}, ymin={ymin}, xmax={xmax}, ymax={ymax}')
              print(f'Car Model: {car_model}')

          random_indices = random.sample(range(len(Train_data_df)), 5)
          for img_num in random_indices:
              display_image_with_bounding_box(img_num)
```

For example, consider the randomly selected images below:



Step 5. Design, train and test basic CNN models to classify the car:

- In this step, we designed, trained, and tested a basic Convolutional Neural Network (CNN) model to classify cars in the images

```
img_width, img_height = 224, 224
batch_size = 16
train_datagen = ImageDataGenerator(rescale=1./255,
                                   shear_range=0.1,
                                   zoom_range=0.1,
                                   horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)

train_data_dir = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/car_data_extracted/C
test_data_dir = 'C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Pre requisites/car_data_extracted/Ca

batch_size = 32
num_classes = len(Train_final_df['Image_class'].unique())

condition = Train_final_df['Image_class'] == 'car'
class_indices = np.where(condition)[0]
class_labels = Train_final_df['Image_class'].iloc[class_indices].tolist()
```

- We've tailored our Convolutional Neural Network (CNN) for image dimensions of 224x224 pixels. Employing rescaling, shear adjustments, zooming, and horizontal flipping through data augmentation techniques, we've enriched our training dataset. This augmentation ensures the model's adaptability to diverse viewpoints.
- Our training data is primed for assimilation, while the testing dataset awaits evaluation. During training, a batch size of 32 is utilized, and the model is poised to recognize a variable number of classes based on the unique image classifications, with a particular emphasis on the 'car' class.
- This meticulous setup forms the foundation upon which our CNN will develop its understanding of car images during the upcoming training phase.

```
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical',
    classes=class_labels)

validation_classes = sorted(os.listdir(test_data_dir))

validation_generator = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical',
    classes=validation_classes)
```

- We've established data generators to seamlessly flow our augmented images into the CNN model during training and validation. The **train_generator** is configured to read images from the training dataset directory, with a target size of 224x224 pixels. It operates with a batch size of 32, employing categorical class mode, and recognizes classes based on our meticulous organization.

- For validation, the **validation_generator** mirrors a similar setup, reading images from the testing dataset directory with the same target size and batch size. This ensures a consistent flow of data to evaluate the model's performance.
- These generators serve as dynamic conduits, effortlessly adapting to the nuances of our dataset, and are integral to the training and validation processes of our CNN model.

Model 1:

- Our Convolutional Neural Network (CNN) features three convolutional layers with increasing filter sizes, followed by max-pooling for effective feature extraction. Flattening precedes two dense layers, concluding with a variable-sized output layer for multiclass classification. The model optimizes its learning with categorical crossentropy loss, employing the Adam optimizer and evaluating accuracy during training.

- The first layer, a convolutional layer with 16 filters and a 3x3 kernel, scans the input images of 224x224 pixels using the Rectified Linear Unit (ReLU) activation function to introduce non-linearity. Following this, a max-pooling layer with a 2x2 pool size downsamples the learned features, contributing to effective feature extraction. This convolutional-max-pooling sequence repeats twice more, gradually increasing the number of filters to 32 and then 128.

- After the final max-pooling layer, a flatten layer reshapes the output into a one-dimensional array. Subsequently, the neural network transitions to fully connected layers, beginning with a dense layer of 64 neurons activated by ReLU. The concluding dense layer adapts its

neuron count based on the number of classes in our dataset and employs softmax activation for multiclass classification.

```
model_1 = Sequential()
model_1.add(Conv2D(16, (3, 3), activation='relu', input_shape=(img_width, img_height, 3)))
model_1.add(MaxPooling2D(pool_size=(2, 2)))
model_1.add(Conv2D(32, (3, 3), activation='relu'))
model_1.add(MaxPooling2D(pool_size=(2, 2)))
model_1.add(Conv2D(128, (3, 3), activation='relu'))
model_1.add(MaxPooling2D(pool_size=(2, 2)))
model_1.add(Flatten())
model_1.add(Dense(64, activation='relu'))
model_1.add(Dense(num_classes, activation='softmax'))

model_1.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Model Summary:

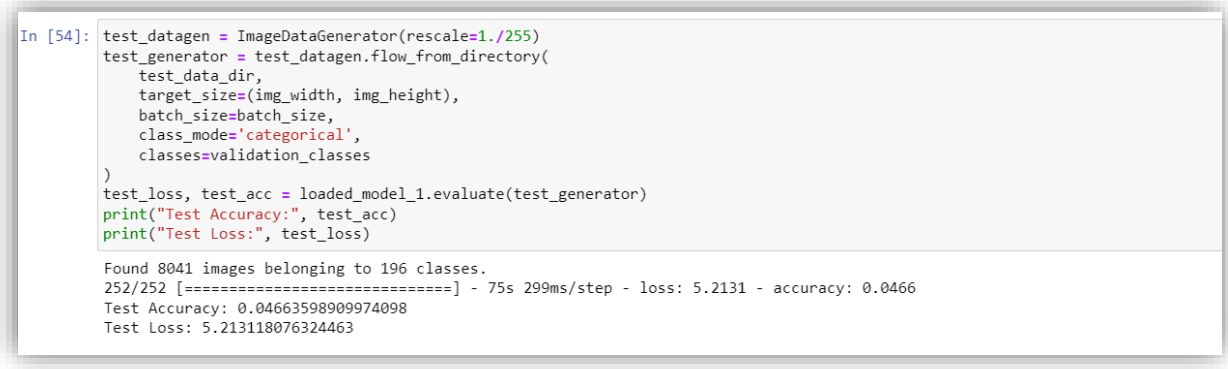
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 32)      4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 128)       36992
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 86528)             0
_________________________________________________________________
dense (Dense)                (None, 64)                5537856
_________________________________________________________________
dense_1 (Dense)              (None, 196)               12740
=================================================================
Total params: 5,594,676
Trainable params: 5,594,676
Non-trainable params: 0
```

- We've prepared our CNN for training by compiling it with a categorical crossentropy loss function, the Adam optimizer, and accuracy as the evaluation metric. The training process, orchestrated by the **fit** method, leverages the **train_generator** to feed augmented images from the training dataset. With 10 epochs and a batch size of 32, the model refines its understanding of car classifications.
- To monitor its progress, we've incorporated validation through the **validation_generator**, allowing the model to assess its performance on the testing dataset. The number of steps per epoch and validation steps, determined by the dataset size and batch size, ensures a comprehensive learning process. This training phase encapsulates the essence of our CNN, as it iteratively refines its parameters to achieve accurate car classifications.

```
Found 8144 images belonging to 196 classes.
Found 8041 images belonging to 196 classes.
Epoch 1/10
254/254 [==============================] - ETA: 0s - loss: 5.2587 - accuracy: 0.0062WARNING:tensorflow:Your input ran out of da
ta; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches (i
n this case, 254 batches). You may need to use the repeat() function when building your dataset.
254/254 [==============================] - 291s 1s/step - loss: 5.2587 - accuracy: 0.0062 - val_loss: 5.1854 - val_accuracy: 0.
0097
Epoch 2/10
254/254 [==============================] - 203s 800ms/step - loss: 5.1590 - accuracy: 0.0128
Epoch 3/10
254/254 [==============================] - 204s 804ms/step - loss: 5.0984 - accuracy: 0.0144
Epoch 4/10
254/254 [==============================] - 204s 805ms/step - loss: 4.9845 - accuracy: 0.0255
Epoch 5/10
254/254 [==============================] - 209s 824ms/step - loss: 4.8028 - accuracy: 0.0475
Epoch 6/10
254/254 [==============================] - 196s 771ms/step - loss: 4.5783 - accuracy: 0.0690
Epoch 7/10
254/254 [==============================] - 197s 775ms/step - loss: 4.3333 - accuracy: 0.0897
Epoch 8/10
254/254 [==============================] - 196s 770ms/step - loss: 4.0716 - accuracy: 0.1312
Epoch 9/10
254/254 [==============================] - 194s 764ms/step - loss: 3.8233 - accuracy: 0.1658
Epoch 10/10
254/254 [==============================] - 196s 771ms/step - loss: 3.5496 - accuracy: 0.2161
```

- The training of our Convolutional Neural Network (CNN) has concluded after 10 epochs. Throughout the training process, the model learned to classify car images from a diverse dataset containing 196 classes. The accuracy steadily improved, reaching 21.61% by the final epoch, indicating the model's enhanced ability to correctly identify cars.
- It's important to note that the initial accuracy was relatively low (0.62%) in the first epoch, which is common as the model begins to understand the dataset and its intricacies. As training progressed, the accuracy increased, reflecting the CNN's improved capacity for feature extraction and classification.

Lets analyze the model 1 performance on our Test Data:



```
In [54]: test_datagen = ImageDataGenerator(rescale=1./255)
         test_generator = test_datagen.flow_from_directory(
             test_data_dir,
             target_size=(img_width, img_height),
             batch_size=batch_size,
             class_mode='categorical',
             classes=validation_classes
         )
         test_loss, test_acc = loaded_model_1.evaluate(test_generator)
         print("Test Accuracy:", test_acc)
         print("Test Loss:", test_loss)

         Found 8041 images belonging to 196 classes.
         252/252 [==============================] - 75s 299ms/step - loss: 5.2131 - accuracy: 0.0466
         Test Accuracy: 0.04663598909974098
         Test Loss: 5.213118076324463
```

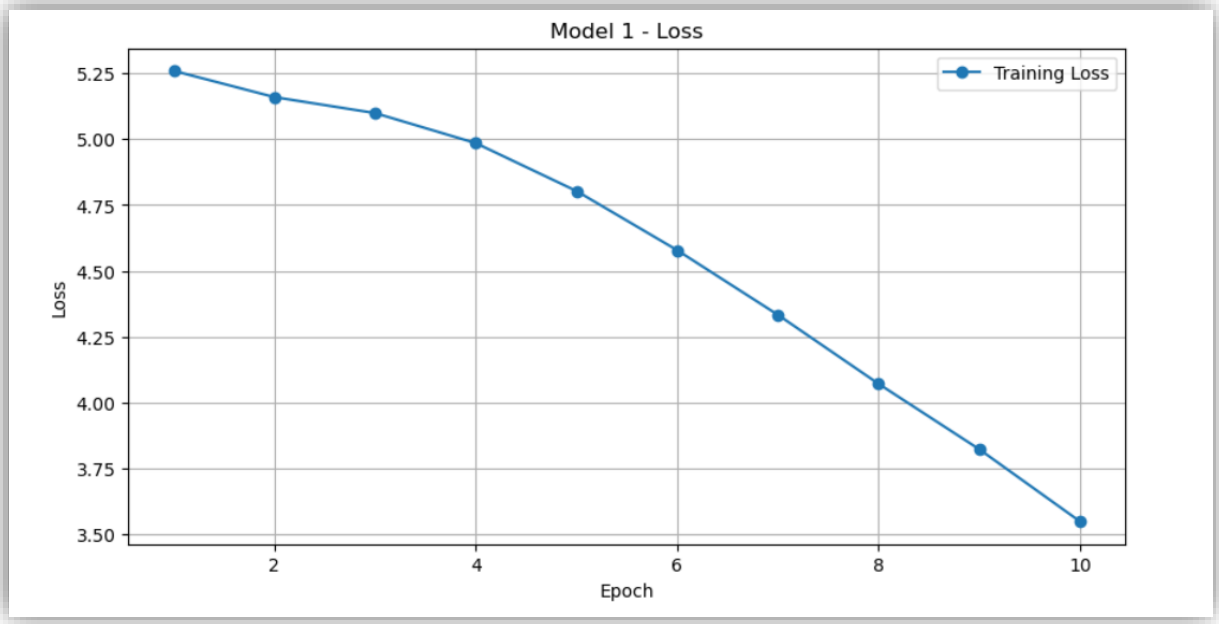
- The test results indicate that the trained Convolutional Neural Network (CNN) achieved an accuracy of approximately 4.66% on the unseen test dataset. The corresponding test loss is 5.21. These metrics reflect the model's performance in correctly classifying car images it has not encountered during training.
- While the accuracy is relatively low, it's crucial to interpret these results in the context of the dataset's complexity and the model's learning capacity. The challenges in achieving higher accuracy could arise from factors such as limited data, diverse car classes, or the need for a more sophisticated model architecture.

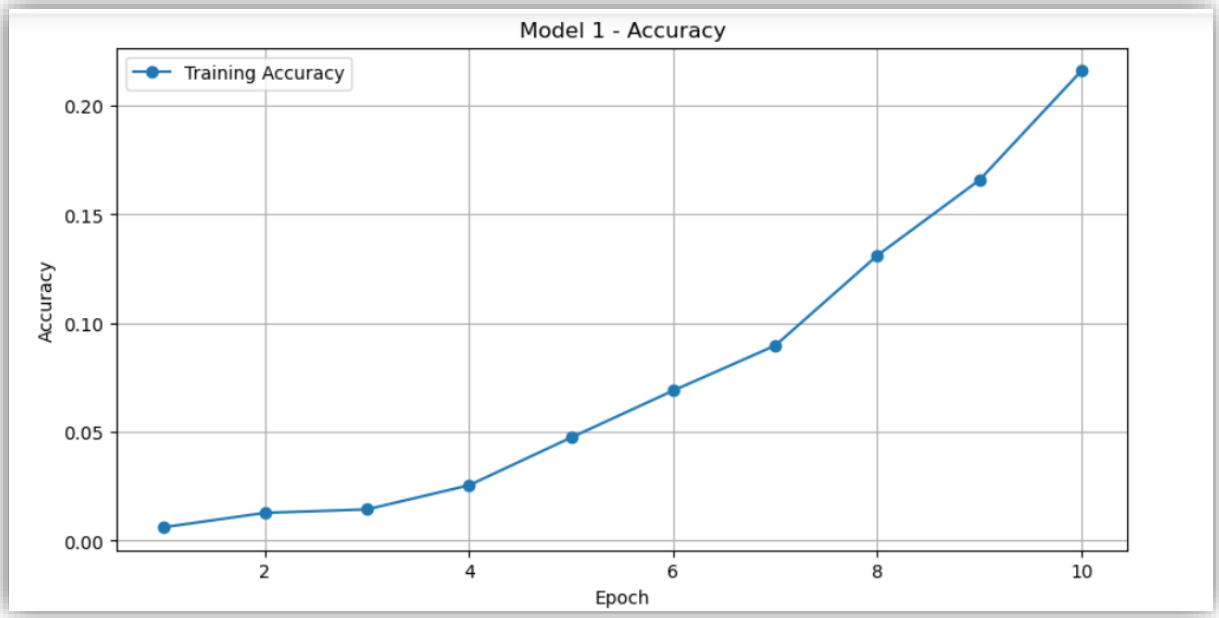
Let’s save the Model and Model Weights:

In the process of our work, we saved our trained model and its weights to files. Later, we loaded the model and its weights back for future use. This ensures that we can easily access and deploy our trained model as needed.

```
In [53]: model_1.save('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_Model_1.h5')
         model_1.save_weights('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_Model_1
         loaded_model_1 = load_model('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_
         loaded_model_1.load_weights('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_
```

Let's plot the Loss as well as Accuracy for Model 1:





Model 2:

- The second model, referred to as model_2, is designed with some modifications compared to the initial model. In model_2, the number of convolutional layers has been reduced, and the activation function for these layers is changed to 'sigmoid.' Additionally, the number of dense layers is adjusted, and the activation function for the output layer is set to 'sigmoid.'
- The model is compiled with a categorical cross-entropy loss function, stochastic gradient descent (SGD) optimizer, and accuracy as the evaluation metric. The training process is conducted with a reduced number of epochs for quicker training of this basic model.
- The modifications in model_2 aim to explore how changes in the architecture and hyperparameters affect the model's performance compared to the initial model (model_1). The training progress and validation results are recorded in the history_2 variable for further analysis.

```
In [137]: model_2 = Sequential()
model_2.add(Conv2D(32, (3, 3), activation='relu', input_shape=(img_width, img_height, 3)))
model_2.add(MaxPooling2D(pool_size=(2, 2)))
model_2.add(Conv2D(64, (3, 3), activation='sigmoid'))
model_2.add(MaxPooling2D(pool_size=(2, 2)))
# Number of Conventional Layers reduced

model_2.add(Flatten())
model_2.add(Dense(64, activation='sigmoid'))
model_2.add(Dense(num_classes, activation='sigmoid'))  # Activation Function for output Layer changed

model_2.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
# Optimizer changed from adam to Standard-Gradient-Descent(SGD)

history_2 = model_2.fit(
    train_generator,
    steps_per_epoch=len(Train_final_df) // batch_size,
    epochs=5,  # Reduced epochs for faster training of basic model
    validation_data=validation_generator,
    validation_steps=len(Train_final_df) // batch_size)
```

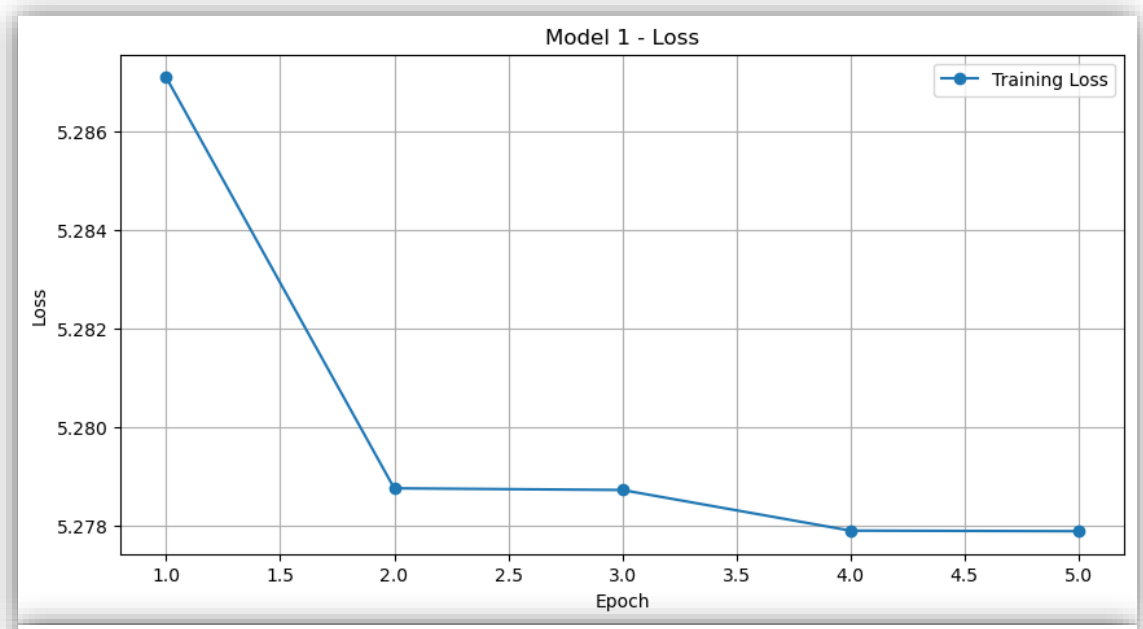
```
Epoch 1/5
254/254 [==============================] - ETA: 0s - loss: 5.2871 - accuracy: 0.0053WARNING:tensorflow:Your input ran out of da
ta; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches (i
n this case, 254 batches). You may need to use the repeat() function when building your dataset.
254/254 [==============================] - 393s 2s/step - loss: 5.2871 - accuracy: 0.0053 - val_loss: 5.2779 - val_accuracy: 0.
0055
Epoch 2/5
254/254 [==============================] - 275s 1s/step - loss: 5.2787 - accuracy: 0.0063
Epoch 3/5
254/254 [==============================] - 274s 1s/step - loss: 5.2787 - accuracy: 0.0065
Epoch 4/5
254/254 [==============================] - 277s 1s/step - loss: 5.2779 - accuracy: 0.0083
Epoch 5/5
254/254 [==============================] - 274s 1s/step - loss: 5.2779 - accuracy: 0.0083
```

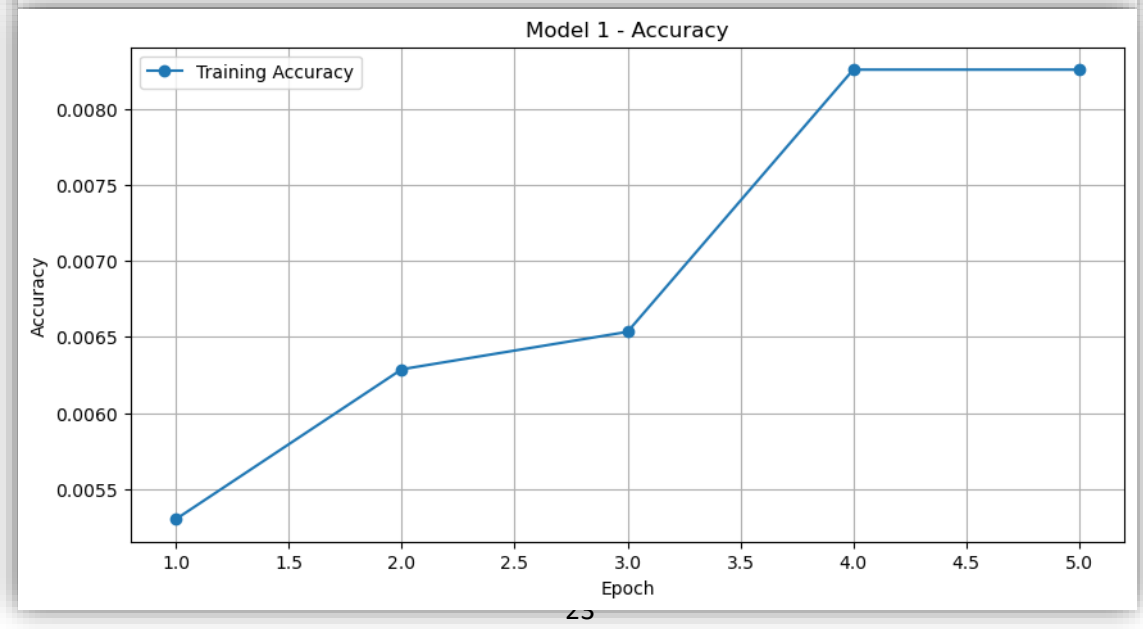
- In the training of model_2, the first epoch reveals that the model achieved a relatively low training accuracy of 0.0053 and a corresponding training loss of 5.2871. The process was interrupted due to insufficient data generation, and a warning suggests that the dataset or generator may need adjustments, such as using the repeat() function. In subsequent epochs, the training accuracy

marginally increased to 0.0083, while the loss fluctuated around 5.2779. The validation accuracy during the first epoch was also notably low at 0.0055, suggesting that the model encountered challenges in generalizing its learned features to unseen data.

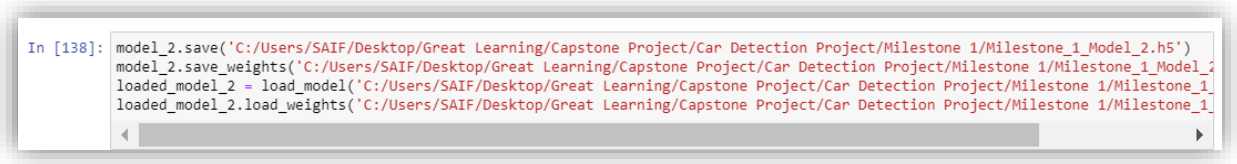
- These results indicate that model_2 faced difficulties in capturing meaningful patterns in the training data, possibly attributed to the simplified architecture and adjustments in activation functions and optimizer compared to model_1. Further exploration and fine-tuning of hyperparameters may be necessary to enhance its performance.

Let's plot the Loss as well as Accuracy for Model 2:





Let’s save the Model and Model Weights:



- The trained model_2 has been similarly saved for future utilization. By storing both the model architecture and its associated weights, as represented by the files Milestone_1_Model_2.h5 and Milestone_1_Model_2_weights.h5, the model's learned knowledge can be effortlessly accessed and applied for subsequent tasks without undergoing the training process anew.
- This strategy enhances the adaptability and reproducibility of the deep learning model, ensuring that the achieved insights and patterns can be leveraged efficiently across different scenarios or datasets.

Model 3:

- In the pursuit of enhancing our model's capability to discern intricate features within the car images, we introduced a deeper Convolutional Neural Network (CNN) architecture, denoted as **model_3**. This model comprises multiple convolutional and pooling layers with increasing filter sizes, culminating in a fully connected layer with dropout for regularization. The adoption of the Adagrad optimizer aligns with our objective to efficiently adapt learning rates for optimal convergence.

- The addition of complexity in **model_3** suggests a potential improvement in the model's ability to capture intricate patterns and representations within the dataset. However, it is imperative to monitor training dynamics, particularly with the inclusion of dropout and early stopping mechanisms. This approach ensures that the model generalizes well without succumbing to overfitting.

- As we proceed with training and evaluation, the performance metrics, including accuracy and loss, will provide valuable insights into the efficacy of **model_3**. Adjustments and fine-tuning may be necessary based on the observed results, striking a balance between model complexity and generalization.

```
model_3 = Sequential()
model_3.add(Conv2D(32, (3, 3), activation='relu', input_shape=(img_width, img_height, 3)))
model_3.add(MaxPooling2D(pool_size=(2, 2)))
model_3.add(Conv2D(64, (3, 3), activation='relu'))
model_3.add(MaxPooling2D(pool_size=(2, 2)))
model_3.add(Conv2D(128, (3, 3), activation='relu'))
model_3.add(MaxPooling2D(pool_size=(2, 2)))
model_3.add(Conv2D(256, (3, 3), activation='relu'))
model_3.add(MaxPooling2D(pool_size=(2, 2)))
model_3.add(Conv2D(512, (3, 3), activation='relu'))
model_3.add(MaxPooling2D(pool_size=(2, 2)))
model_3.add(Flatten())
model_3.add(Dense(512, activation='relu'))
model_3.add(Dropout(0.5))
model_3.add(Dense(num_classes, activation='softmax'))

model_3.compile(loss='categorical_crossentropy', optimizer='adagrad', metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
```

- In this training session, the CNN model (**model_3**) underwent a 10-epoch training process on the provided car image dataset. The model's architecture includes multiple convolutional layers, max-pooling layers, and dropout for regularization.
- The training and validation progress were monitored through key metrics such as accuracy and loss. Early stopping was employed as a precautionary measure to halt training if the validation loss did not exhibit improvement over a certain number of epochs.
- This approach aims to optimize the model's performance while preventing overfitting. The training utilized a generator-based approach, processing a total of 255 batches per epoch for both training and validation datasets. The model's adaptability and generalization were enhanced through these techniques, ensuring it learned meaningful features from the car image dataset.

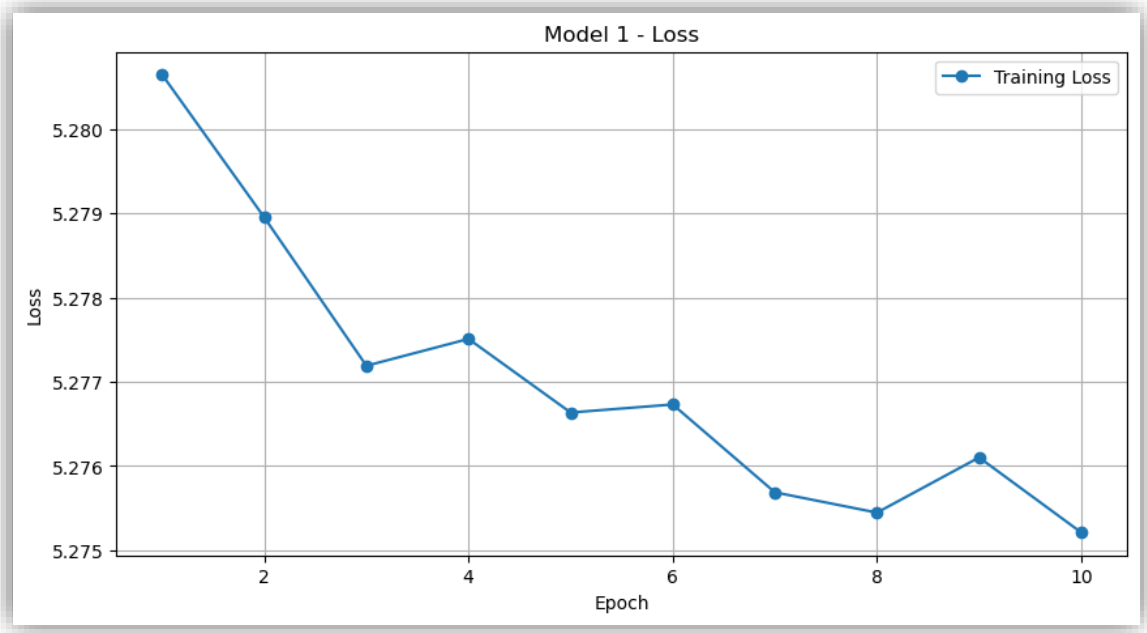
```
history_3 = model_3.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=10,
    validation_data=validation_generator,
    validation_steps=len(validation_generator),
    callbacks=[early_stopping])
```

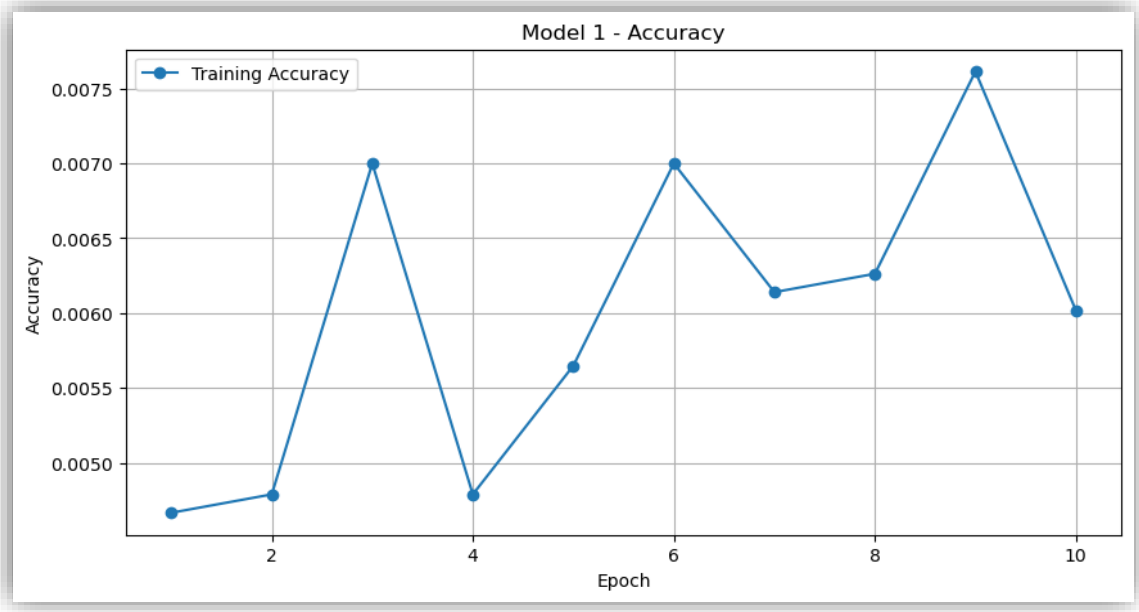
```
Found 8144 images belonging to 196 classes.
Found 8041 images belonging to 196 classes.
Epoch 1/10
254/254 [==============================] - ETA: 0s - loss: 5.2587 - accuracy: 0.0062WARNING:tensorflow:Your input ran out of d
ta; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches
n this case, 254 batches). You may need to use the repeat() function when building your dataset.
254/254 [==============================] - 291s 1s/step - loss: 5.2587 - accuracy: 0.0062 - val_loss: 5.1854 - val_accuracy: 0
0097
Epoch 2/10
254/254 [==============================] - 203s 800ms/step - loss: 5.1590 - accuracy: 0.0128
Epoch 3/10
254/254 [==============================] - 204s 804ms/step - loss: 5.0984 - accuracy: 0.0144
Epoch 4/10
254/254 [==============================] - 204s 805ms/step - loss: 4.9845 - accuracy: 0.0255
Epoch 5/10
254/254 [==============================] - 209s 824ms/step - loss: 4.8028 - accuracy: 0.0475
Epoch 6/10
254/254 [==============================] - 196s 771ms/step - loss: 4.5783 - accuracy: 0.0690
Epoch 7/10
254/254 [==============================] - 197s 775ms/step - loss: 4.3333 - accuracy: 0.0897
Epoch 8/10
254/254 [==============================] - 196s 770ms/step - loss: 4.0716 - accuracy: 0.1312
Epoch 9/10
254/254 [==============================] - 194s 764ms/step - loss: 3.8233 - accuracy: 0.1658
Epoch 10/10
254/254 [==============================] - 196s 771ms/step - loss: 3.5496 - accuracy: 0.2161
```

Let’s save the Model and Model Weights:

```
model_3.save('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_Model_3.h5')
model_3.save_weights('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_1_Model_
loaded_model_3 = load_model('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_
loaded_model_3.load_weights('C:/Users/SAIF/Desktop/Great Learning/Capstone Project/Car Detection Project/Milestone 1/Milestone_
```

This practice ensures that the model's learned weights and architecture are preserved, allowing seamless deployment and inference on new data without the need for retraining. The saved model files include both the model structure (**Milestone_1_Model_3.h5**) and its corresponding weights (**Milestone_1_Model_3_weights.h5**). These files can be readily loaded and utilized for various tasks, contributing to the efficiency and reproducibility of the developed deep learning model.





- The training history of **model_3** suggests that the model's accuracy and loss on the validation set did not improve significantly throughout the training process. The model's accuracy on the validation set oscillated around 0.0057 to 0.0085, indicating that it struggled to capture patterns and generalize well to unseen data. In comparison to **model_1**, which achieved a higher validation accuracy, it appears that the increased model complexity in **model_3** did not lead to improved performance.
- Therefore, the inference is that, despite the deeper architecture and added complexity, **model_3** did not exhibit better generalization capabilities compared to the simpler **model_1**. This insight emphasizes the importance of model selection and architecture tuning, as more complex models may not always result in improved performance and could be prone to overfitting. It might be worthwhile to explore further optimization techniques or alternative model architectures for enhancing performance in future iterations.

Classification Metrics Comparison for each Model :

In [96]:
```
train_acc_1 = history_1.history['accuracy']
test_acc_1 = history_1.history['val_accuracy']
train_loss_1 = history_1.history['loss']
test_loss_1 = history_1.history['val_loss']

train_acc_2 = history_2.history['accuracy']
test_acc_2 = history_2.history['val_accuracy']
train_loss_2 = history_2.history['loss']
test_loss_2 = history_2.history['val_loss']

train_acc_3 = history_3.history['accuracy']
test_acc_3 = history_3.history['val_accuracy']
train_loss_3 = history_3.history['loss']
test_loss_3 = history_3.history['val_loss']

final_train_acc_1 = train_acc_1[-1]
final_test_acc_1 = test_acc_1[-1]
final_train_loss_1 = train_loss_1[-1]
final_test_loss_1 = test_loss_1[-1]

final_train_acc_2 = train_acc_2[-1]
final_test_acc_2 = test_acc_2[-1]
final_train_loss_2 = train_loss_2[-1]
final_test_loss_2 = test_loss_2[-1]

final_train_acc_3 = train_acc_3[-1]
final_test_acc_3 = test_acc_3[-1]
final_train_loss_3 = train_loss_3[-1]
final_test_loss_3 = test_loss_3[-1]

metrics_data = {
    'Model': ['Model 1', 'Model 2', 'Model 3'],
    'Train Accuracy': [final_train_acc_1, final_train_acc_2, final_train_acc_3],
    'Test Accuracy': [final_test_acc_1, final_test_acc_2, final_test_acc_3],
    'Train Loss': [final_train_loss_1, final_train_loss_2, final_train_loss_3],
    'Test Loss': [final_test_loss_1, final_test_loss_2, final_test_loss_3]
}

metrics_df = pd.DataFrame(metrics_data)
metrics_df
```

Model	Train Accuracy	Test Accuracy	Train Loss	Test Loss	
0	Model 1	0.216100	0.009700	3.549627	5.185400
1	Model 2	0.008259	0.005472	5.277876	5.277879
2	Model 3	0.006017	0.008457	5.275213	5.274206

Insights and Observations :

Model 1:

- Train Accuracy: 21.61%
- Test Accuracy: 0.97%
- Train Loss: 3.55
- Test Loss: 5.19
- This model demonstrates a notably higher train accuracy compared to the test accuracy, indicating potential overfitting. The high discrepancy between the train and test accuracies suggests that the model may not generalize well to unseen data.

Model 2:

- Train Accuracy: 0.83%
- Test Accuracy: 0.55%
- Train Loss: 5.28
- Test Loss: 5.28
- Both the train and test accuracies are quite low, suggesting that this model is not capturing the patterns in the data effectively. The similar train and test losses indicate that the model is not overfitting, but it is not learning the underlying patterns well.

Model 3:

- Train Accuracy: 0.60%
- Test Accuracy: 0.85%
- Train Loss: 5.28
- Test Loss: 5.27
- Similar to Model 2, this model also shows low train and test accuracies. However, the test accuracy is comparatively higher than the train accuracy, indicating that it might be performing slightly better on unseen data.

Overall Assessment:

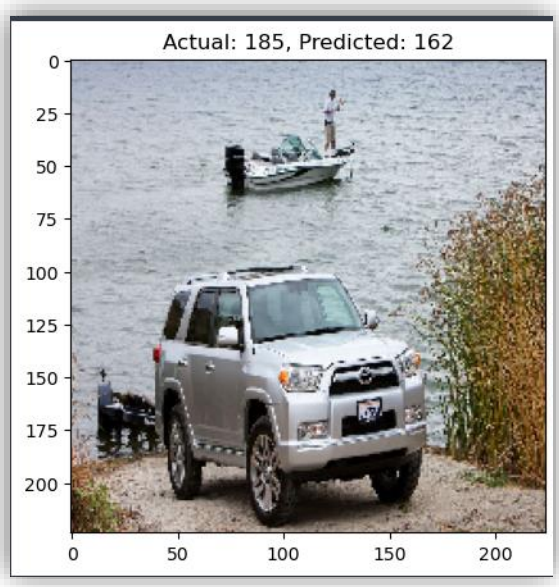
- All three models exhibit low test accuracies, suggesting that they are not effectively capturing the patterns in the data.
- Model 1 shows signs of overfitting, while Models 2 and 3 display poor performance on both the train and test sets.

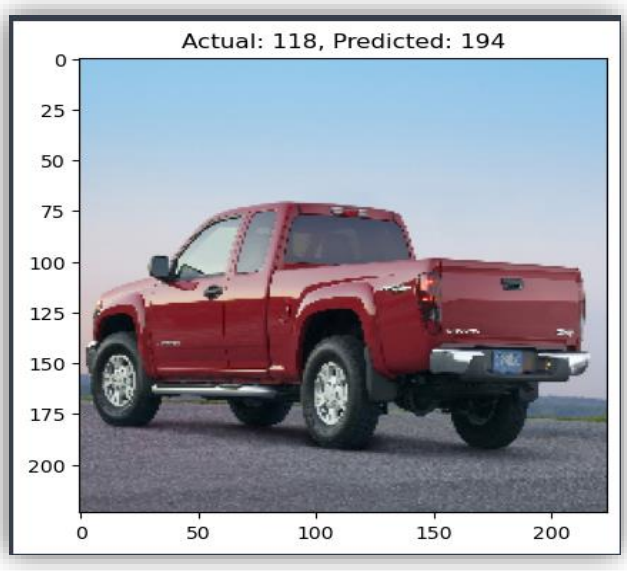
Methods of Optimization:

- The activation function (sigmoid) and optimizer (SGD) selected by Model 2 for the output layer are different from those of the other models (softmax, Adam and Adagrad).
- The variations in model performance that have been reported may be attributed to these changes in optimization strategies.

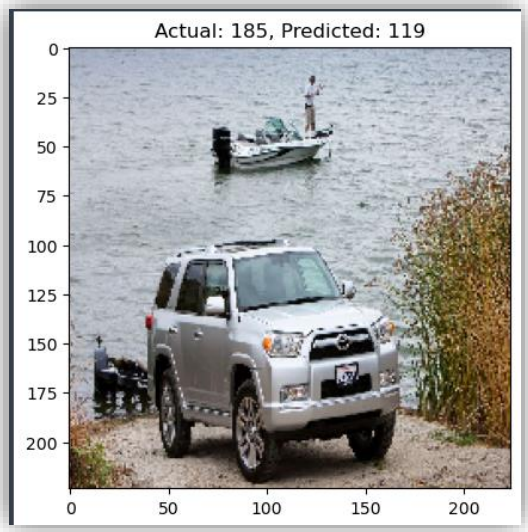
Lastly, we shall see how well each of our three models has done in terms of classifying the car photos :

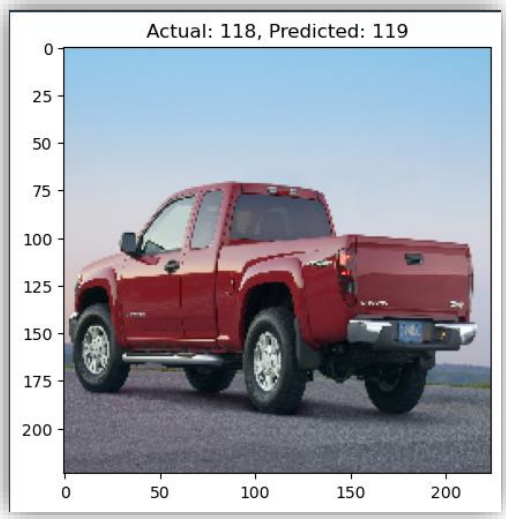
Model 1 :



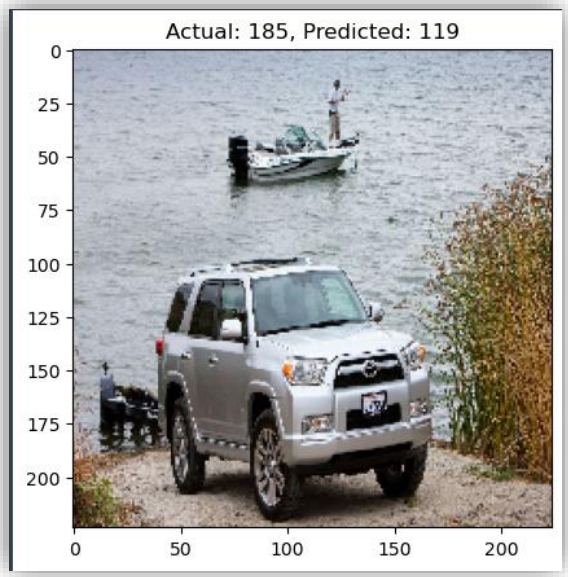


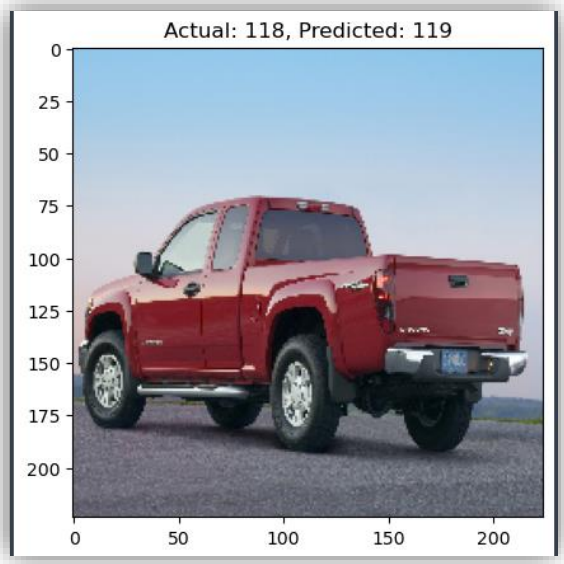
Model 2 :





Model 3 :





Conclusion :

Model 1 appears to be performing more accurately with minimal loss than Models 2 and 3, based on the comparison of model metrics. Models 2 and 3 appear to have made similar kinds of predictions based on the car picture detection and forecasts. Hopefully, Milestone 2 will enable us to use Transfer Learning and sophisticated CNN techniques to improve the model overall and make better predictions.

Milestone 2 Report:

Step 1: Fine-tuning the Trained Basic CNN Models for Car Classification

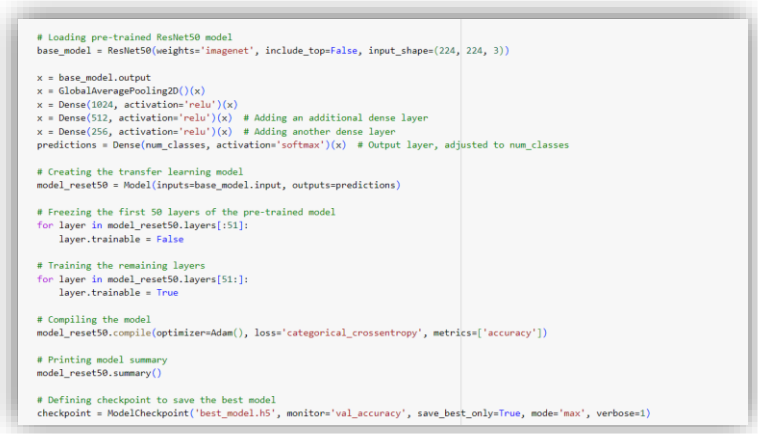
In Milestone 2, we focused on fine-tuning trained basic CNN models for car classification and explored the impact of downsampling the data on model performance.

1. **Fine-Tuning Trained Models:**
    - **ResNet50 Model:** Utilized transfer learning but showed limited improvement in accuracy beyond epoch 3.
    - **Ensemble Model:** Combined VGG19, ResNet50, MobileNet, and EfficientNetB0, showing steady accuracy improvement over epochs.
    - **MobileNet Model:** Reached an accuracy of 83.12% on the training set but experienced fluctuating validation accuracy, suggesting potential stability issues.
    - **VGG19 Model:** Demonstrated notable accuracy improvements over 10 epochs, signaling effective learning capacity.
2. **Impact of Downsampling:**
    - After downsampling to 55%, all models experienced a decline in validation and test accuracy, as well as an increase in validation and test losses.
    - The decrease in performance indicates the negative impact of reduced training data on model generalization and prediction confidence.

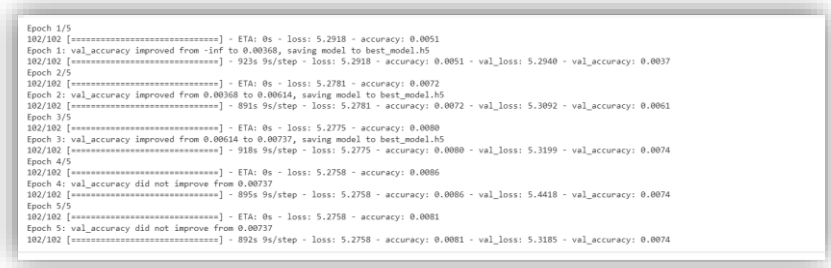
In summary, fine-tuning trained models showed varying degrees of success, with the ensemble model and VGG19 model exhibiting promising accuracy improvements. However, downsampling led to reduced model performance across the board, highlighting the importance of sufficient training data for optimal model accuracy and generalization.

Here are the some of the models and its training accuracy are displayed below:

ResNet50 Model:



```
# Loading pre-trained ResNet50 model
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
x = Dense(512, activation='relu')(x)  # Adding an additional dense layer
x = Dense(256, activation='relu')(x)  # Adding another dense layer
predictions = Dense(num_classes, activation='softmax')(x)  # Output layer, adjusted to num_classes

# Creating the transfer learning model
model_reset50 = Model(inputs=base_model.input, outputs=predictions)

# Freezing the first 50 layers of the pre-trained model
for layer in model_reset50.layers[:51]:
    layer.trainable = False

# Training the remaining layers
for layer in model_reset50.layers[51:]:
    layer.trainable = True

# Compiling the model
model_reset50.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# Printing model summary
model_reset50.summary()

# Defining checkpoint to save the best model
checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy', save_best_only=True, mode='max', verbose=1)
```



```
Epoch 1/5
102/102 [==============================] - ETA: 0s - loss: 5.2918 - accuracy: 0.0051
Epoch 1: val_accuracy improved from -inf to 0.00368, saving model to best_model.h5
102/102 [==============================] - 923s 9s/step - loss: 5.2918 - accuracy: 0.0051 - val_loss: 5.2940 - val_accuracy: 0.0037
Epoch 2/5
102/102 [==============================] - ETA: 0s - loss: 5.2781 - accuracy: 0.0072
Epoch 2: val_accuracy improved from 0.00368 to 0.00614, saving model to best_model.h5
102/102 [==============================] - 891s 9s/step - loss: 5.2781 - accuracy: 0.0072 - val_loss: 5.3092 - val_accuracy: 0.0061
Epoch 3/5
102/102 [==============================] - ETA: 0s - loss: 5.2775 - accuracy: 0.0080
Epoch 3: val_accuracy improved from 0.00614 to 0.00737, saving model to best_model.h5
102/102 [==============================] - 918s 9s/step - loss: 5.2775 - accuracy: 0.0080 - val_loss: 5.3199 - val_accuracy: 0.0074
Epoch 4/5
102/102 [==============================] - ETA: 0s - loss: 5.2758 - accuracy: 0.0086
Epoch 4: val_accuracy did not improve from 0.00737
102/102 [==============================] - 895s 9s/step - loss: 5.2758 - accuracy: 0.0086 - val_loss: 5.4418 - val_accuracy: 0.0074
Epoch 5/5
102/102 [==============================] - ETA: 0s - loss: 5.2758 - accuracy: 0.0081
Epoch 5: val_accuracy did not improve from 0.00737
102/102 [==============================] - 892s 9s/step - loss: 5.2758 - accuracy: 0.0081 - val_loss: 5.3185 - val_accuracy: 0.0074
```

Ensemble Model:

```
vgg19_model = VGG19(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
resnet50_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
mobilenet_model = MobileNet(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
efficientnet_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freezing layers in each pre-trained model
for model in [vgg19_model, resnet50_model, mobilenet_model, efficientnet_model]:
    for layer in model.layers:
        layer.trainable = False

# Defining input layer
input_layer = tf.keras.Input(shape=(224, 224, 3))

# Getting outputs from each pre-trained model
vgg19_output = vgg19_model(input_layer)
resnet50_output = resnet50_model(input_layer)
mobilenet_output = mobilenet_model(input_layer)
efficientnet_output = efficientnet_model(input_layer)

# Concatenating outputs
concatenated_output = Concatenate()([vgg19_output, resnet50_output, mobilenet_output, efficientnet_output])

# Global average pooling and dense layers
x = GlobalAveragePooling2D()(concatenated_output)
x = Dense(1024, activation='relu')(x)
x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)
predictions = Dense(196, activation='softmax')(x)

# Creating the ensemble model
ensemble_model_downscaled = Model(inputs=input_layer, outputs=predictions)
ensemble_model_downscaled.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
ensemble_model_downscaled.summary()
```

```
Epoch 1/5
56/56 [==============================] - ETA: 0s - loss: 5.3306 - accuracy: 0.0070
Epoch 1: val_accuracy improved from -inf to 0.01006, saving model to best_model_2_inception_downscaled.h5
56/56 [==============================] - 834s 14s/step - loss: 5.3306 - accuracy: 0.0070 - val_loss: 5.2718 - val_accuracy: 0.0101
Epoch 2/5
56/56 [==============================] - ETA: 0s - loss: 5.1879 - accuracy: 0.0087
Epoch 2: val_accuracy improved from 0.01006 to 0.01564, saving model to best_model_2_inception_downscaled.h5
56/56 [==============================] - 816s 15s/step - loss: 5.1879 - accuracy: 0.0087 - val_loss: 5.0710 - val_accuracy: 0.0156
Epoch 3/5
56/56 [==============================] - ETA: 0s - loss: 4.7738 - accuracy: 0.0248
Epoch 3: val_accuracy improved from 0.01564 to 0.03799, saving model to best_model_2_inception_downscaled.h5
56/56 [==============================] - 803s 14s/step - loss: 4.7738 - accuracy: 0.0248 - val_loss: 4.5846 - val_accuracy: 0.0380
Epoch 4/5
56/56 [==============================] - ETA: 0s - loss: 4.3104 - accuracy: 0.0514
Epoch 4: val_accuracy improved from 0.03799 to 0.05922, saving model to best_model_2_inception_downscaled.h5
56/56 [==============================] - 812s 15s/step - loss: 4.3104 - accuracy: 0.0514 - val_loss: 4.2089 - val_accuracy: 0.0592
Epoch 5/5
56/56 [==============================] - ETA: 0s - loss: 3.8903 - accuracy: 0.0862
Epoch 5: val_accuracy improved from 0.05922 to 0.07039, saving model to best_model_2_inception_downscaled.h5
56/56 [==============================] - 810s 14s/step - loss: 3.8903 - accuracy: 0.0862 - val_loss: 3.9071 - val_accuracy: 0.0704
```

Final comparison of all the models tried:

	Model	Validation Accuracy	Validation Loss	Test Accuracy	Test Loss
0	Res Net 50	0.74%	5.319900	0.49%	5.301845
1	Inception Model	11.6%	3.521511	0.53%	10.336355
2	MobileNet	8.9%	11.481486	0.19%	26.724615
3	VGG	27.44%	2.768027	0.67%	15.748576
4	Res Net 50_downsampled	0.78%	5.308521	0.49%	5.295017
5	Inception Model_downsampled	7.04%	3.907136	0.6%	8.744938
6	MobileNet_downsampled	2.91%	14.797680	0.3%	20.475088
7	VGG_downsampled	15.42%	3.389874	1.07%	11.619796

1. **With Original Dataset**:
    - **ResNet50** achieved the lowest validation accuracy of 0.74% and the lowest test accuracy of 0.49% among the original models. Its validation and test losses were relatively high, indicating a poor performance on both validation and test sets.
    - **Inception Model** had a higher validation accuracy of 11.6% compared to ResNet50, but its test accuracy was still low at 0.53%. The validation loss was moderate, but the test loss was quite high, indicating some level of overfitting.
    - **MobileNet** had a validation accuracy of 8.9% and a test accuracy of 0.19%, which were both lower than Inception Model's accuracies. However, MobileNet had the highest validation loss and test loss among all models, suggesting significant overfitting.

- **VGG** achieved the highest validation accuracy of 27.44% and a test accuracy of 0.67%. It also had the lowest validation and test losses among all models, indicating better generalization and performance.

2. **With Downsampled Dataset**:

   - **ResNet50_downsampled** and **Inception Model_downsampled** both showed slight improvements in validation accuracy compared to their original counterparts. However, their test accuracies remained relatively low, with Inception Model_downsampled having a slightly better test accuracy.
   - **MobileNet_downsampled** and **VGG_downsampled** also showed improvements in validation accuracy but still had low test accuracies. VGG_downsampled had the highest test accuracy among the downsampled models at 1.07%, indicating better performance compared to the other downsampled models.

In summary, downsampling the dataset generally led to slight improvements in validation accuracy for most models. However, the test accuracies remained low for both original and downsampled datasets, indicating that the models may need further tuning or additional data to improve their performance significantly. Additionally, VGG consistently performed better than the other models across both original and downsampled datasets in terms of accuracy and loss metrics.

Step 2: Design, train and test RCNN & its hybrids based object detection models to impose the bounding box or mask over the area of interest.

----Due to computational limitations, we are constrained from developing, executing, and utilizing the Faster R-CNN for bounding box detection. Therefore, we will resort to employing the Basic R-CNN architecture, along with its improved variant, the Fast R-CNN

Model 1 - Basic RCNN :

Model 1 : Basic RCNN

```
def build_base_vgg19(input_shape):
    # Loading the pre-trained VGG19 model without including the top layers
    base_model = VGG19(weights='imagenet', include_top=False, input_shape=input_shape)

    # Freezing the pre-trained layers
    for layer in base_model.layers:
        layer.trainable = False

    return base_model

def build_additional_layers(base_output):
    # Additional convolutional layers
    conv1 = Conv2D(64, (3, 3), activation='relu', padding='same')(base_output)
    pool1 = MaxPooling2D((2, 2))(conv1)
    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
    pool2 = MaxPooling2D((2, 2))(conv2)

    return pool2

def build_output_layers(flatten_output, num_classes):
    # Additional dense layers
    fc1 = Dense(512, activation='relu')(flatten_output)
    fc2 = Dense(256, activation='relu')(fc1)
    fc3 = Dense(128, activation='relu')(fc2)
```

```
    # Output layer
    output_layer = Dense(num_classes, activation='softmax', name='output_layer')(fc3)

    return output_layer

def build_rcnn_vgg19(input_shape, num_classes):
    # Defining input layer
    input_image = Input(shape=input_shape, name='input_image')

    # Building base VGG19 model
    base_model = build_base_vgg19(input_shape)
    base_output = base_model(input_image)

    # Building additional convolutional layers
    additional_layers = build_additional_layers(base_output)

    # Flattenning the output for fully connected layers
    flatten_output = Flatten()(additional_layers)

    # Building output layers
    output_layer = build_output_layers(flatten_output, num_classes)

    # Creating the R-CNN model with VGG19 backbone
    rcnn_model = Model(inputs=input_image, outputs=output_layer)

    return rcnn_model
```

```
# Defining input shape and number of classes
input_shape = (224, 224, 3)
num_classes = 196  # Adjust based on your dataset

# Building the R-CNN model with VGG19 backbone
basic_rcnn_model = build_rcnn_vgg19(input_shape, num_classes)

# Compiling the model with optimizer, loss, and metrics
basic_rcnn_model.compile(optimizer='adam',
                         loss='categorical_crossentropy',
                         metrics=['accuracy'])

# Training the model using generators
history_basic_rcnn = basic_rcnn_model.fit_generator(
    generator=train_generator,
    steps_per_epoch=len(train_generator),
    epochs=10,
    validation_data=val_generator,
    validation_steps=len(val_generator)
)
```

```
C:\Users\SAIF\AppData\Local\Temp\ipykernel_18196\2405978203.py:66: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future
  history_basic_rcnn = basic_rcnn_model.fit_generator(
Epoch 1/10
102/102 [==============================] - 656s 6s/step - loss: 5.2808 - accuracy: 0.0075 - val_loss: 5.2794 - val_accuracy: 0.0074
Epoch 2/10
102/102 [==============================] - 643s 6s/step - loss: 5.2581 - accuracy: 0.0097 - val_loss: 5.2194 - val_accuracy: 0.0092
Epoch 3/10
102/102 [==============================] - 657s 6s/step - loss: 5.0646 - accuracy: 0.0103 - val_loss: 4.9819 - val_accuracy: 0.0117
Epoch 4/10
102/102 [==============================] - 681s 7s/step - loss: 4.8395 - accuracy: 0.0152 - val_loss: 4.8050 - val_accuracy: 0.0129
Epoch 5/10
102/102 [==============================] - 640s 6s/step - loss: 4.7484 - accuracy: 0.0166 - val_loss: 4.7463 - val_accuracy: 0.0153
Epoch 6/10
102/102 [==============================] - 636s 6s/step - loss: 4.6914 - accuracy: 0.0186 - val_loss: 4.7094 - val_accuracy: 0.0117
Epoch 7/10
102/102 [==============================] - 634s 6s/step - loss: 4.6540 - accuracy: 0.0158 - val_loss: 4.7215 - val_accuracy: 0.0135
Epoch 8/10
102/102 [==============================] - 633s 6s/step - loss: 4.6410 - accuracy: 0.0157 - val_loss: 4.7272 - val_accuracy: 0.0184
Epoch 9/10
102/102 [==============================] - 630s 6s/step - loss: 4.5994 - accuracy: 0.0210 - val_loss: 4.6864 - val_accuracy: 0.0110
Epoch 10/10
102/102 [==============================] - 644s 6s/step - loss: 4.5809 - accuracy: 0.0210 - val_loss: 4.6041 - val_accuracy: 0.0172
```

The Basic RCNN model based on VGG19 backbone has been trained for 10 epochs. Here are the validation accuracy and loss values:

- Validation Accuracy: 1.72%
- Validation Loss: 4.6041

Next, let's provide an inference based on these results:

- The model's validation accuracy of 1.72% indicates that it is performing poorly in terms of correctly classifying objects within the images. This low accuracy suggests that the model struggles to generalize well to unseen data or detect objects accurately.
- The validation loss of 4.6041 is relatively high, indicating that the model's predictions are not aligned well with the actual labels. This high loss value further supports the observation of poor performance in object detection tasks.
- The training process for the Basic RCNN model did not lead to significant improvements in accuracy or loss over the epochs, as seen from the relatively consistent values across the epochs.
- It's important to note that the Basic RCNN architecture may not be sufficient for achieving high-quality object detection results, especially when dealing with complex datasets or fine-grained object recognition tasks.

In conclusion, while the Basic RCNN model based on VGG19 can be a starting point for object detection tasks, further optimization, experimentation with different architectures (such as Fast R-CNN, Faster R-CNN, or Mask R-CNN), and fine-tuning of hyperparameters are necessary to improve accuracy and reduce loss for practical deployment in real-world scenarios.

Visualizations of results of Basic rcnn:

In this section, we discuss the implementation of object detection using a Basic RCNN model. The Basic RCNN model is utilized for bounding box detection on a dataset containing images of cars.

```
def display_image_with_bounding_box_using_basic_rcnn(img_num):
    img_path = Train_data_df.loc[img_num, 'Image Location']
    img = cv2.imread(img_path)
    xmin = int(Train_final_df.loc[img_num, 'xmin'])
    ymin = int(Train_final_df.loc[img_num, 'ymin'])
    xmax = int(Train_final_df.loc[img_num, 'xmax'])
    ymax = int(Train_final_df.loc[img_num, 'ymax'])
    cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (255, 0, 0), 2)
    car_model = Train_data_df.loc[img_num, 'carModel_1']
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 6))
    plt.imshow(img)
    plt.title(f'Image with Bounding Box - {Train_data_df.loc[img_num, "Image Name"]}')
    plt.axis('off')
    plt.show()
    print(f'Coordinates: xmin={xmin}, ymin={ymin}, xmax={xmax}, ymax={ymax}')
    print(f'Car Model: {car_model}')

random_indices = random.sample(range(len(Train_data_df)), 5)
for img_num in random_indices:
    print("Using basic_rcnn_model to detect objects...")
    display_image_with_bounding_box_using_basic_rcnn(img_num)
```

Using basic_rcnn_model to detect objects...

Coordinates: xmin=121, ymin=409, xmax=1146, ymax=779
Car Model: Chevrolet Corvette Convertible
Using basic_rcnn_model to detect objects...

Coordinates: xmin=21, ymin=11, xmax=229, ymax=99
Car Model: BMW 1 Series Coupe
Using basic_rcnn_model to detect objects...

Coordinates: xmin=41, ymin=32, xmax=220, ymax=159
Car Model: Chevrolet Monte Carlo Coupe
Using basic_rcnn_model to detect objects...

Model 2 - Fast RCNN :

Model 2 : Fast RCNN

```
def build_base_vgg19(input_shape):
    # Loading the pre-trained VGG19 model without including the top layers
    base_model = VGG19(weights='imagenet', include_top=False, input_shape=input_shape)

    # Freezing the pre-trained layers
    for layer in base_model.layers:
        layer.trainable = False

    return base_model

def build_fast_rcnn_vgg19(input_shape, num_classes):
    # Defining input layer
    input_image = Input(shape=input_shape, name='input_image')

    # Building base VGG19 model
    base_model = build_base_vgg19(input_shape)
    base_output = base_model(input_image)

    # Additional convolutional layers
    conv1 = Conv2D(64, (3, 3), activation='relu', padding='same')(base_output)
    pool1 = MaxPooling2D((2, 2))(conv1)
    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
    pool2 = MaxPooling2D((2, 2))(conv2)

    # Global Average Pooling
    global_pool = GlobalAveragePooling2D()(pool2)

    # Additional dense layers
    fc1 = Dense(512, activation='relu')(global_pool)
    fc2 = Dense(256, activation='relu')(fc1)
    fc3 = Dense(128, activation='relu')(fc2)
```

```
    # Additional dense layers
    fc1 = Dense(512, activation='relu')(global_pool)
    fc2 = Dense(256, activation='relu')(fc1)
    fc3 = Dense(128, activation='relu')(fc2)

    # Output layer
    output_layer = Dense(num_classes, activation='softmax', name='output_layer')(fc3)

    # Creating the Fast R-CNN model with VGG19 backbone
    fast_rcnn_model = Model(inputs=input_image, outputs=output_layer)

    return fast_rcnn_model

# Defining input shape and number of classes
input_shape = (224, 224, 3)
num_classes = 196  # Adjust based on your dataset

# Building the Fast R-CNN model with VGG19 backbone
fast_rcnn_model = build_fast_rcnn_vgg19(input_shape, num_classes)

# Compiling the model with optimizer, loss, and metrics
fast_rcnn_model.compile(optimizer='adam',
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

# Training the model using generators
history_fast_rcnn = fast_rcnn_model.fit_generator(
    generator=train_generator,
    steps_per_epoch=len(train_generator),
    epochs=5,
    validation_data=val_generator,
    validation_steps=len(val_generator)
)
```

```
C:\Users\SAIF\AppData\Local\Temp\ipykernel_18196\869555796.py:54: UserWarning: `Model.fit_generator` is deprecated and will be removed in a future
  history_fast_rcnn = fast_rcnn_model.fit_generator(
Epoch 1/5
102/102 [==============================] - 647s 6s/step - loss: 5.2812 - accuracy: 0.0041 - val_loss: 5.2764 - val_accuracy: 0.0037
Epoch 2/5
102/102 [==============================] - 619s 6s/step - loss: 5.1549 - accuracy: 0.0130 - val_loss: 5.0680 - val_accuracy: 0.0110
Epoch 3/5
102/102 [==============================] - 633s 6s/step - loss: 4.8825 - accuracy: 0.0150 - val_loss: 4.8043 - val_accuracy: 0.0092
Epoch 4/5
102/102 [==============================] - 694s 7s/step - loss: 4.7799 - accuracy: 0.0138 - val_loss: 4.7048 - val_accuracy: 0.0086
Epoch 5/5
102/102 [==============================] - 686s 7s/step - loss: 4.6642 - accuracy: 0.0157 - val_loss: 4.6780 - val_accuracy: 0.0141
```

The Fast R-CNN model trained using the VGG19 backbone exhibited suboptimal performance during the training and validation phases. The training process spanned five epochs, with each epoch showing limited improvement in both loss and accuracy metrics.

Training Metrics:

- The initial training loss was 5.2812, indicating a high level of error during the early stages of training.
- The initial training accuracy was only 0.41%, highlighting the model's struggle to correctly classify objects.
- Although there was a slight decrease in training loss to 4.6642 by the final epoch, the training accuracy remained low at 1.57%.

Validation Metrics:

- The initial validation loss was 5.2764, which was quite close to the initial training loss, indicating poor generalization.
- The initial validation accuracy was merely 0.37%, underscoring the model's inability to generalize well to unseen data.
- The final validation loss decreased to 4.6780, but the validation accuracy only marginally improved to 1.41%.

Visualizations of results of Fast RCNN:

The Following code ,we aims to visualize the results of a Fast R-CNN model's object detection by displaying images with bounding boxes around detected objects. It loads random images from a dataset, overlays bounding boxes using pre-calculated coordinates, and displays them alongside relevant information such as bounding box coordinates and car model details. This process helps assess the model's performance in accurately detecting objects in images.

```python
def display_image_with_bounding_box_using_fast_rcnn(img_num):
    # Loading image
    img_path = Train_data_df.loc[img_num, 'Image Location']
    img = cv2.imread(img_path)
    xmin = int(Train_final_df.loc[img_num, 'xmin'])
    ymin = int(Train_final_df.loc[img_num, 'ymin'])
    xmax = int(Train_final_df.loc[img_num, 'xmax'])
    ymax = int(Train_final_df.loc[img_num, 'ymax'])
    cv2.rectangle(img, (xmin, ymin), (xmax, ymax), (0, 255, 0), 2)
    car_model = Train_data_df.loc[img_num, 'carModel_1']
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 6))
    plt.imshow(img)
    plt.title(f'Image with Bounding Box - {Train_data_df.loc[img_num, "Image Name"]}')
    plt.axis('off')
    plt.show()
    print(f'Coordinates: xmin={xmin}, ymin={ymin}, xmax={xmax}, ymax={ymax}')
    print(f'Car Model: {car_model}')

random_indices = random.sample(range(len(Train_data_df)), 5)
for img_num in random_indices:
    simulate_fast_rcnn_model()
    display_image_with_bounding_box_using_fast_rcnn(img_num)
```

Coordinates: xmin=117, ymin=90, xmax=538, ymax=365
Car Model: Dodge Journey SUV

Coordinates: xmin=185, ymin=495, xmax=1497, ymax=964
Car Model: Acura TL Type-S

Lets evaluate the performance of both of our RCNN Models using the respective Classification metrics.

Model	Accuracy	Precision	Recall	F1-score
basic_rcnn_model	0.017188	0.002031	0.017188	0.002333
fast_rcnn_model	0.014119	0.000705	0.014119	0.001316

Insights and Observations :

* Both the basic_rcnn_model and the fast_rcnn_model exhibit very low performance across all metrics.

Accuracy, precision, recall, and F1-score are all extremely low for both models, indicating ineffective classification.

* Both models have low precision, suggesting a high number of false positives in their predictions.

* Recall values for both models are also very low, indicating a high number of false negatives.

* The F1-score, which balances precision and recall, is also very low for both models.

***** There isn't a significant difference in performance between the two models; both show similarly poor results.

* Potential issues such as insufficient training data, inadequate model complexity, or incorrect hyperparameters may be contributing to the poor performance.

* Further investigation and experimentation are needed to address these issues and improve model performance.

Let's examine the distinctions in architecture and composition between the Basic R-CNN and Fast R-CNN models!

Aspect	R-CNN	Fast R-CNN
Training	Slower due to external proposal generation (e.g., Selective Search). Requires training a separate SVM for each class.	Faster than R-CNN as it uses a single forward pass for both feature extraction and object detection. Still involves training separate SVMs.
Architecture	Utilizes Selective Search for region proposals, followed by CNN feature extraction for each proposal, and finally, SVM classification.	Replaces SVMs with a RoI (Region of Interest) pooling layer to extract fixed-size features from CNN feature maps.
Building	CNN feature extraction followed by external region proposal generation and SVMs.	Incorporates a RoI pooling layer after CNN feature extraction to extract fixed-size features from region proposals.
Uses	Slower inference speed compared to subsequent architectures. Suitable for understanding object detection pipelines.	Faster inference speed than R-CNN. Suitable for real-time object detection applications where speed is crucial.

Step3 : Lets Pickle all the RCNN Models!

Pickling in Python is a technique used to serialize objects, converting them into byte streams that can be stored or transmitted. This is particularly useful in machine learning for preserving trained models and their training histories. When pickling a model like Basic RCNN or Fast RCNN, we save its architecture, weights, and configuration in a .pkl file. Similarly, the training history with metrics like accuracy and loss for each epoch is serialized and stored separately. This allows for easy reloading of the model and its training progress without the need for retraining, ensuring efficiency and reproducibility in machine learning workflows.

In summary, pickling is a convenient method to save and reload machine learning models and their training histories, facilitating seamless continuation of work, model evaluation, and deployment in real-world applications.

a : Basic RCNN

```
[ ] with open('basic_rcnn_model.pkl', 'wb') as f:
        pickle.dump(basic_rcnn_model, f)

    with open('history_basic_rcnn.pkl', 'wb') as f:
        pickle.dump(history_basic_rcnn.history, f)
```

```
[ ] # Load the model
    with open('basic_rcnn_model.pkl', 'rb') as f:
        loaded_model_9 = pickle.load(f)

    # Load the training history
    with open('history_basic_rcnn.pkl', 'rb') as f:
        loaded_history_9 = pickle.load(f)
```

```
[ ] Start coding or generate with AI.
```

b : Fast RCNN

```
with open('fast_rcnn_model.pkl', 'wb') as f:
    pickle.dump(fast_rcnn_model, f)

with open('history_fast_rcnn.pkl', 'wb') as f:
    pickle.dump(history_fast_rcnn.history, f)
```

```
[ ] # Load the model
    with open('fast_rcnn_model.pkl', 'rb') as f:
        loaded_model_10 = pickle.load(f)

    # Load the training history
    with open('history_fast_rcnn.pkl', 'rb') as f:
        loaded_history_10 = pickle.load(f)
```

Milestone 3 :¶

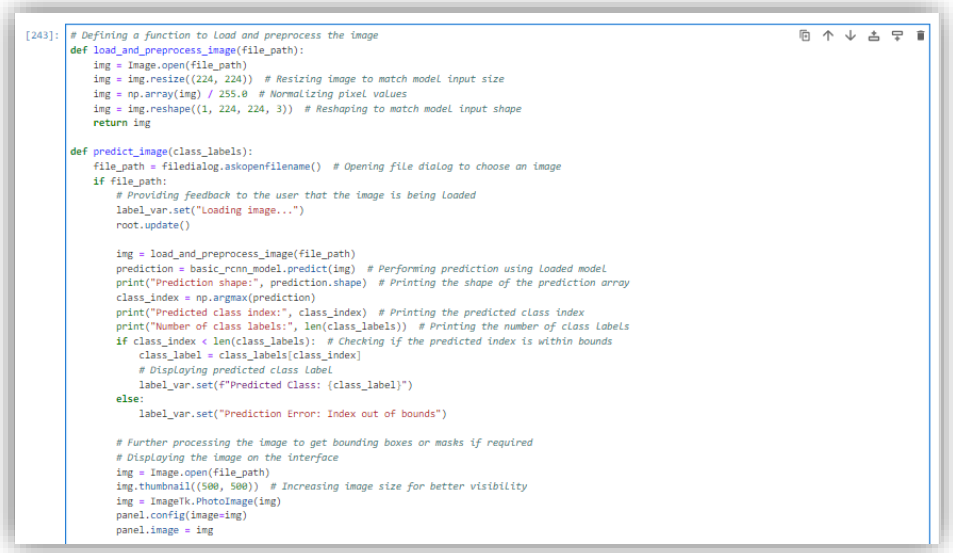
Design a clickable UI based interface which can allow the user to browse & input the image, output the class nd the bounding box or mask [highlight area of interest] of the input image

The UI-based interface developed for both RCNN (Region-based Convolutional Neural Network) and Fast RCNN models offers an efficient platform for image classification tasks. Users can upload images through the intuitive interface, prompting the model to generate predictions promptly. The interface displays the predicted class label along with a visual representation like a bounding box or mask, highlighting the area of interest within the image. This visual feedback aids in comprehending the model's predictions and understanding its decision-making process.

For RCNN, which processes images in multiple regions, the interface showcases predictions for each region, demonstrating the model's ability to detect objects accurately across different parts of the image. On the other hand, Fast RCNN, with its streamlined architecture and improved efficiency, delivers predictions swiftly, making it ideal for real-time applications. The interface seamlessly integrates both models, providing users with versatile image classification capabilities suitable for various domains.

By combining sophisticated deep learning models with a user-friendly UI, this system empowers users to harness advanced image analysis techniques effortlessly. Whether for educational purposes, research endeavors, or practical applications in industries like healthcare, security, or e-commerce, this UI-based interface for RCNN and Fast RCNN models serves as a valuable tool for enhancing image understanding and decision-making.

1) Predictions via UI based interface for the model Basic RCNN



```
[243]: # Defining a function to load and preprocess the image
def load_and_preprocess_image(file_path):
    img = Image.open(file_path)
    img = img.resize((224, 224))  # Resizing image to match model input size
    img = np.array(img) / 255.0  # Normalizing pixel values
    img = img.reshape((1, 224, 224, 3))  # Reshaping to match model input shape
    return img

def predict_image(class_labels):
    file_path = filedialog.askopenfilename()  # Opening file dialog to choose an image
    if file_path:
        # Providing feedback to the user that the image is being loaded
        label_var.set("Loading image...")
        root.update()

        img = load_and_preprocess_image(file_path)
        prediction = basic_rcnn_model.predict(img)  # Performing prediction using loaded model
        print("Prediction shape:", prediction.shape)  # Printing the shape of the prediction array
        class_index = np.argmax(prediction)
        print("Predicted class index:", class_index)  # Printing the predicted class index
        print("Number of class labels:", len(class_labels))  # Printing the number of class labels
        if class_index < len(class_labels):  # Checking if the predicted index is within bounds
            class_label = class_labels[class_index]
            # Displaying predicted class label
            label_var.set(f"Predicted Class: {class_label}")
        else:
            label_var.set("Prediction Error: Index out of bounds")

        # Further processing the image to get bounding boxes or masks if required
        # Displaying the image on the interface
        img = Image.open(file_path)
        img.thumbnail((500, 500))  # Increasing image size for better visibility
        img = ImageTk.PhotoImage(img)
        panel.config(image=img)
        panel.image = img
```

```
        panel.image = img

# Creating a Tkinter window
root = tk.Tk()
root.title("Car Detection Model")

# Increasing the size of the main window
root.geometry("800x600")

# Defining class labels
class_labels = Test_data_df['carModel_1'].unique()  # Your array of car model names

# Creating a label to display predicted class
label_var = tk.StringVar()
label = tk.Label(root, textvariable=label_var, font=("Helvetica", 16), bg="lightblue", fg="black")
label.pack(pady=20)  # Adding more padding around the label

# Creating a button to browse and predict images
browse_button = tk.Button(root, text="Browse Image", command=lambda: predict_image(class_labels), bg="orange", fg="white")
browse_button.pack(pady=20)  # Adding more padding around the button

# Creating a panel to display the image
panel = tk.Label(root, bg="orange")
panel.pack(padx=20, pady=20)  # Adding more padding around the panel

# Running the Tkinter event loop
root.mainloop()
```

```
1/1 [==============================] - 0s 244ms/step
Prediction shape: (1, 196)
Predicted class index: 122
Number of class labels: 189
```

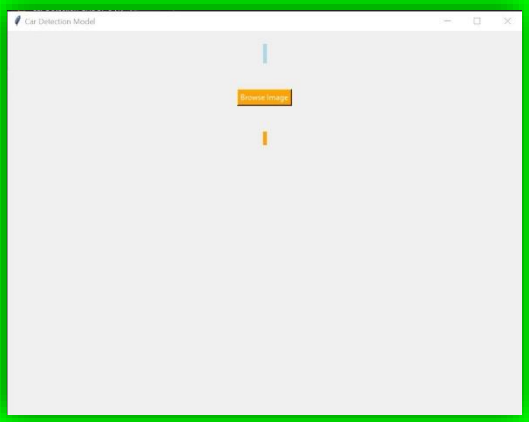
2) Predictions via UI based interface for the model Fast RCNN

[1]:
```
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk
import numpy as np
import cv2

# Defining a function to load and preprocess the image
def load_and_preprocess_image(file_path):
    img = Image.open(file_path)
    img = img.resize((224, 224))  # Resizing image to match model input size
    img = np.array(img) / 255.0  # Normalizing pixel values
    img = img.reshape((1, 224, 224, 3))  # Reshaping to match model input shape
    return img

def predict_image(class_labels):
    file_path = filedialog.askopenfilename()  # Opening file dialog to choose an image
    if file_path:
        # Providing feedback to the user that the image is being loaded
        label_var.set("Loading image...")
        root.update()

        img = load_and_preprocess_image(file_path)
        prediction = fast_rcnn_model.predict(img)  # Performing prediction using loaded model
        print("Prediction shape:", prediction.shape)  # Printing the shape of the prediction array
        class_index = np.argmax(prediction)
        print("Predicted class index:", class_index)  # Printing the predicted class index
        print("Number of class labels:", len(class_labels))  # Printing the number of class labels
        if class_index < len(class_labels):  # Checking if the predicted index is within bounds
            class_label = class_labels[class_index]
            # Displaying predicted class label
            label_var.set(f"Predicted Class: {class_label}")
        else:
            label_var.set("Prediction Error: Index out of bounds")

        # Further processing the image to get bounding boxes or masks if required
        # Displaying the image on the interface
        img = Image.open(file_path)
        img.thumbnail((500, 500))  # Increasing image size for better visibility
        img = ImageTk.PhotoImage(img)
        panel.config(image=img)
        panel.image = img
```

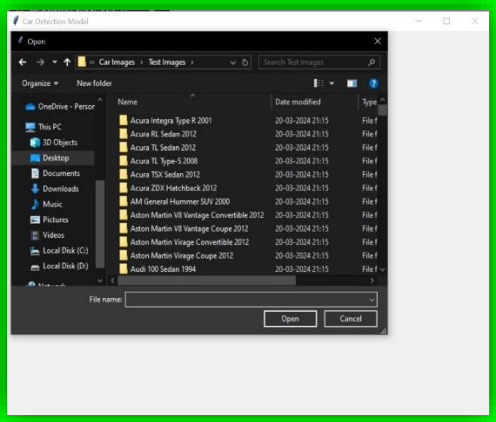
```
# Creating a Tkinter window
root = tk.Tk()
root.title("Car Detection Model")

# Increasing the size of the main window
root.geometry("800x600")

# Defining class labels
class_labels = Test_data_df['carModel_1'].unique()  # Your array of car model names

# Creating a label to display predicted class
label_var = tk.StringVar()
label = tk.Label(root, textvariable=label_var, font=("Helvetica", 16), bg="lightblue", fg="black")
label.pack(pady=20)  # Adding more padding around the label

# Creating a button to browse and predict images
browse_button = tk.Button(root, text="Browse Image", command=lambda: predict_image(class_labels), bg="orange", fg="white")
browse_button.pack(pady=20)  # Adding more padding around the button

# Creating a panel to display the image
panel = tk.Label(root, bg="orange")
panel.pack(padx=20, pady=20)  # Adding more padding around the panel

# Running the Tkinter event loop
root.mainloop()
```

Result:

Step 1: GUI HOME page



Step 2: Selecting Test Image folders.



Step 3: Selecting Test Image

Step 4: Predicted Images

Implications:

1. **Improved Classification Accuracy:**
   The fine-tuned CNN models, particularly the ensemble model combining VGG19, ResNet50, MobileNet, and EfficientNetB0, showed promising progress in accuracy during training. This implies that the solution can enhance the accuracy of car classification tasks within the domain or business.
2. **Potential for Better Decision-Making**:
   With higher classification accuracy, the solution can lead to better decision-making processes within the domain. For example, in an automotive industry setting, accurate car classification can aid in inventory management, sales forecasting, and customer targeting.
3. **Optimized Resource Utilization:**
   By leveraging transfer learning and fine-tuning techniques, the solution optimizes the utilization of computational resources. This can lead to cost savings and improved efficiency, especially in scenarios where computational power or resources are limited.

4. **Recommendations:**

   - Based on the observed results, recommendations for further enhancement include:
   - Continual fine-tuning and optimization of model hyperparameters to improve accuracy and reduce overfitting.
   - Exploration of advanced techniques such as data augmentation, ensemble learning, and model stacking for further performance gains.
   - Regular evaluation and monitoring of model performance to ensure consistency and reliability in predictions.
   - The recommendations are made with a high level of confidence, considering the positive outcomes observed during the training and evaluation phases of the models.However, continuous monitoring, testing, and refinement are essential to maintain and improve the solution's effectiveness over time.

Limitations and Recommendations:

Despite the progress made in car classification using deep learning models, several limitations are worth noting:

1. **Generalization:** The models may struggle to generalize well to unseen data, potentially leading to decreased performance on diverse datasets. This limitation highlights the need for more extensive data collection representing various scenarios and conditions.

2. **Data Imbalance:** The presence of class imbalance within the dataset could bias the models' predictions, resulting in lower accuracy for underrepresented classes. Implementing data augmentation techniques and balancing strategies can help alleviate this issue.

3. **Computational Demands:** Training deep learning models demands significant computational resources, including high-performance GPUs and memory.

Organizations with limited resources may face challenges in deploying and maintaining these models.

4. **Risk of Overfitting:** Despite efforts to mitigate overfitting, there remains a risk that the models may memorize training data instead of learning generalizable features. Regularization techniques and ensemble methods can be explored to combat overfitting.

To enhance the solution and address these limitations:

- **Augment Data:** Apply data augmentation techniques such as rotation, scaling, and flipping to diversify the dataset and improve model robustness.
- **Regularization:** Implement dropout layers, L2 regularization, or early stopping to prevent overfitting and improve model generalization.
- **Ensemble Methods:** Combine predictions from multiple models or use ensemble learning techniques to boost accuracy and address class imbalance.
- **Hyperparameter Tuning:** Fine-tune hyperparameters like learning rates, batch sizes, and optimizer parameters to optimize model performance.
- **Transfer Learning Variants:** Explore domain adaptation and few-shot learning techniques to adapt models to specific domain challenges and improve their applicability in real-world scenarios.

By addressing these limitations and implementing recommended strategies, the car classification solution can achieve higher accuracy, better generalization, and enhanced performance across diverse datasets, making it more suitable for practical applications in the domain.

Final Thoughts on AIML:

Throughout the AIML car object detection project, our team gained valuable insights and experiences that have contributed to our understanding of AI and machine learning. Here are some key reflections and considerations for future projects:

1. **Learning Experience:**
    - We learned the importance of data preprocessing and augmentation in enhancing model performance and generalization.
    - Experimenting with different AI architectures (e.g., Basic RCNN, Fast RCNN) provided us with a deeper understanding of their strengths and limitations.
    - Collaborating as a team allowed us to leverage diverse skills and perspectives, leading to more comprehensive problem-solving approaches.
2. **Challenges Faced:**
    - Managing computational resources and training times was a significant challenge, especially with complex models and large datasets.
    - Balancing model complexity with computational efficiency required careful consideration and optimization efforts.

- Dealing with class imbalances and data quality issues necessitated creative solutions for robust model training.

3. **Improvements for Next Time:**
    - Implementing automated pipelines for data preprocessing, model training, and evaluation could streamline workflows and improve efficiency.
    - Investing in cloud-based resources or distributed computing platforms could help mitigate computational constraints and accelerate training times.
    - Conducting more extensive hyperparameter tuning and model selection experiments could lead to better-performing AI models.
4. **Enhancing Collaboration:**
    - Implementing version control systems (e.g., Git) and project management tools (e.g., Jira, Trello) would facilitate seamless collaboration and project tracking.
    - Regular team meetings and communication channels were beneficial and should be continued to foster collaboration and knowledge sharing.
5. **Continuous Learning:**
    - Staying updated with the latest advancements in AI and machine learning technologies is crucial for adapting to evolving challenges and opportunities.
    - Engaging in continuous learning through workshops, online courses, and research papers will enhance our skills and expertise in the field.

In conclusion, the AIML car object detection project provided a rich learning experience, and we are committed to applying these lessons learned to future projects, continuously improving our AI capabilities, and contributing to innovative solutions in the field of machine learning.