Capstone Project

Computer Vision – Object Classification & Detection

Team:

Mirunalani S
Shreyas Gawade
Nikhilesh Kumar
Santosh Kumar
Sathish Sridharababu

Contents

1. INTRODUCTION

1.1 Project Summary

The project's goal is to develop an automotive surveillance system capable of automatically detecting cars. To achieve this, the predictive modeling leverages Computer Vision techniques combined with Deep Learning. Convolutional layers are used to extract feature maps from images, which are then fed into a neural network to learn and recognize patterns, aligning with the project's core objective.

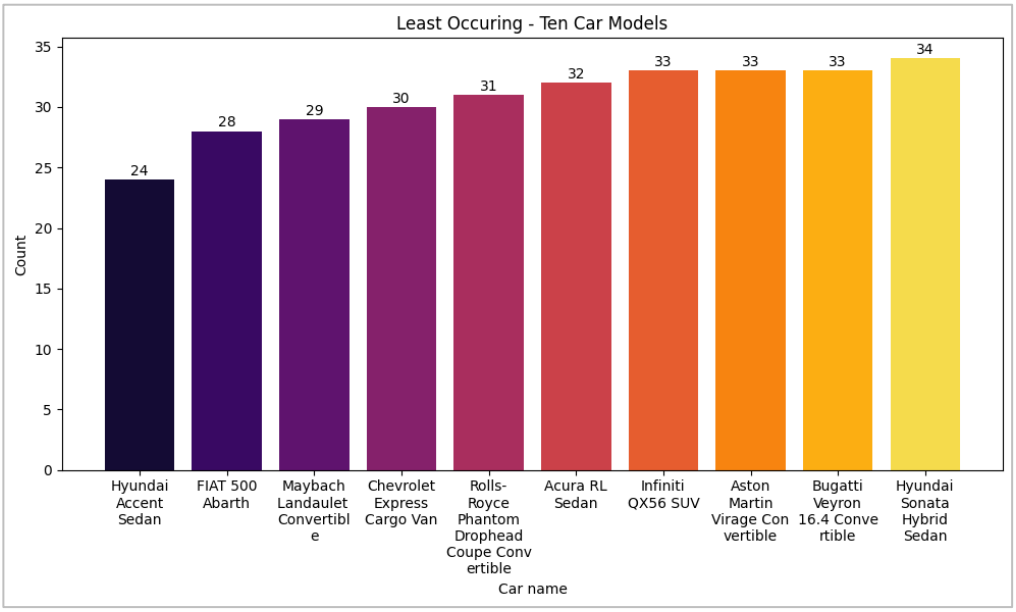
1.2 Data Description

A total of 16k images are segregated by car class provided for the project. Of a total of 16k, 8,144 training images and 8,041 testing images are provided along with image location i.e., bounding box coordinates

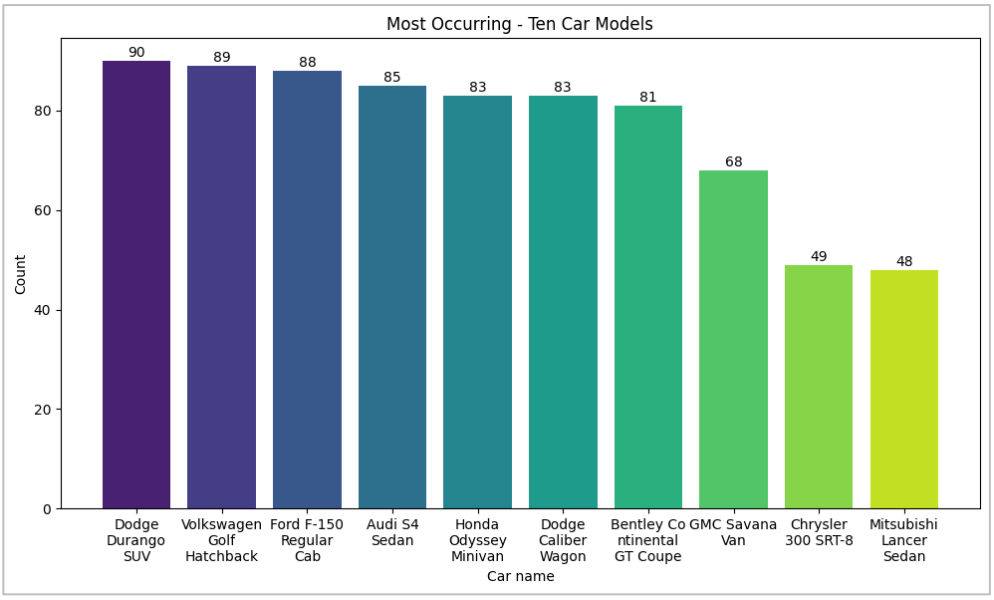
2. DATA ANALYSIS AND PREPROCESSING

2.1 EDA – Most and Least Occurring Car Models

Car names are processed to extract the models by removing the year information, and then records are aggregated by ‘car model’ to count occurrences. The below chart shows the 10 least common car models in the training dataset. Notably, the ‘Hyundai Accent Sedan’ appears the least frequently, highlighting the class imbalance in the dataset.

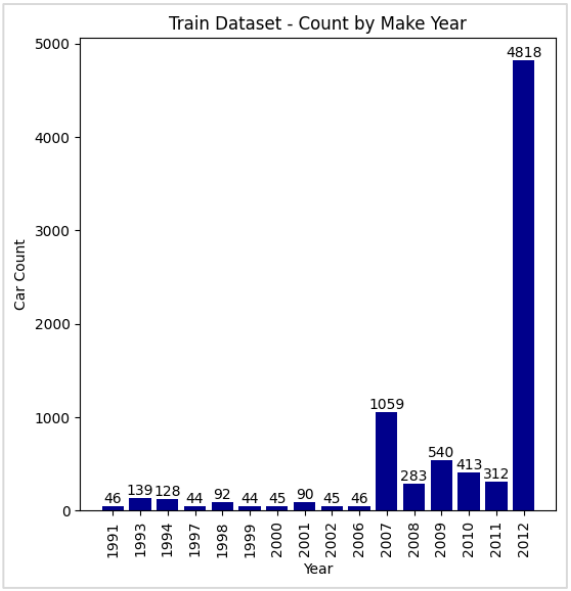


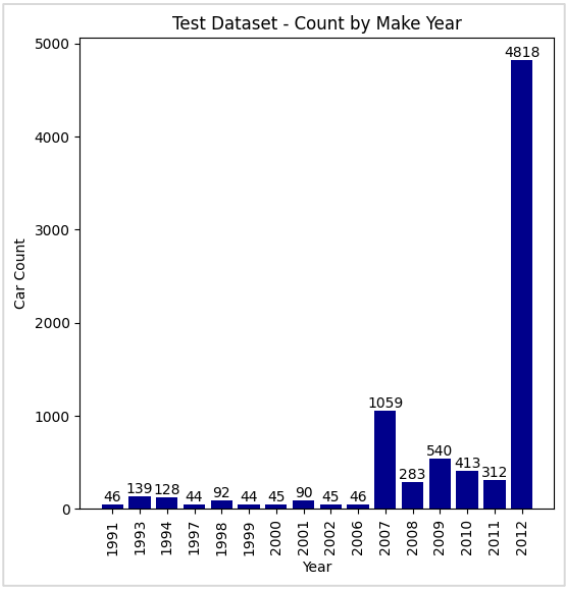
The plot below displays the 10 most frequent car models in the dataset. The 'Dodge Durango SUV' is the most prevalent, with 90 occurrences out of total 8,144 images.



2.2 EDA – Count by Make Year

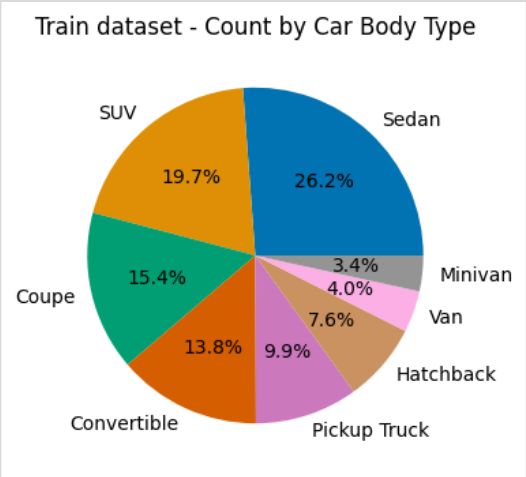
The Make year of cars are extracted from the car name. It can be seen from the plots below that the cars are mostly from the year range of 2007 to 2012 contributing to 90% particularly from years 2007 and 2012.

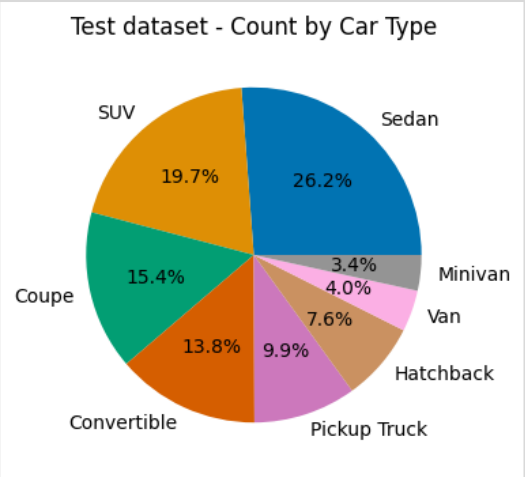




2.3 EDA – Analysis on Car Body Type

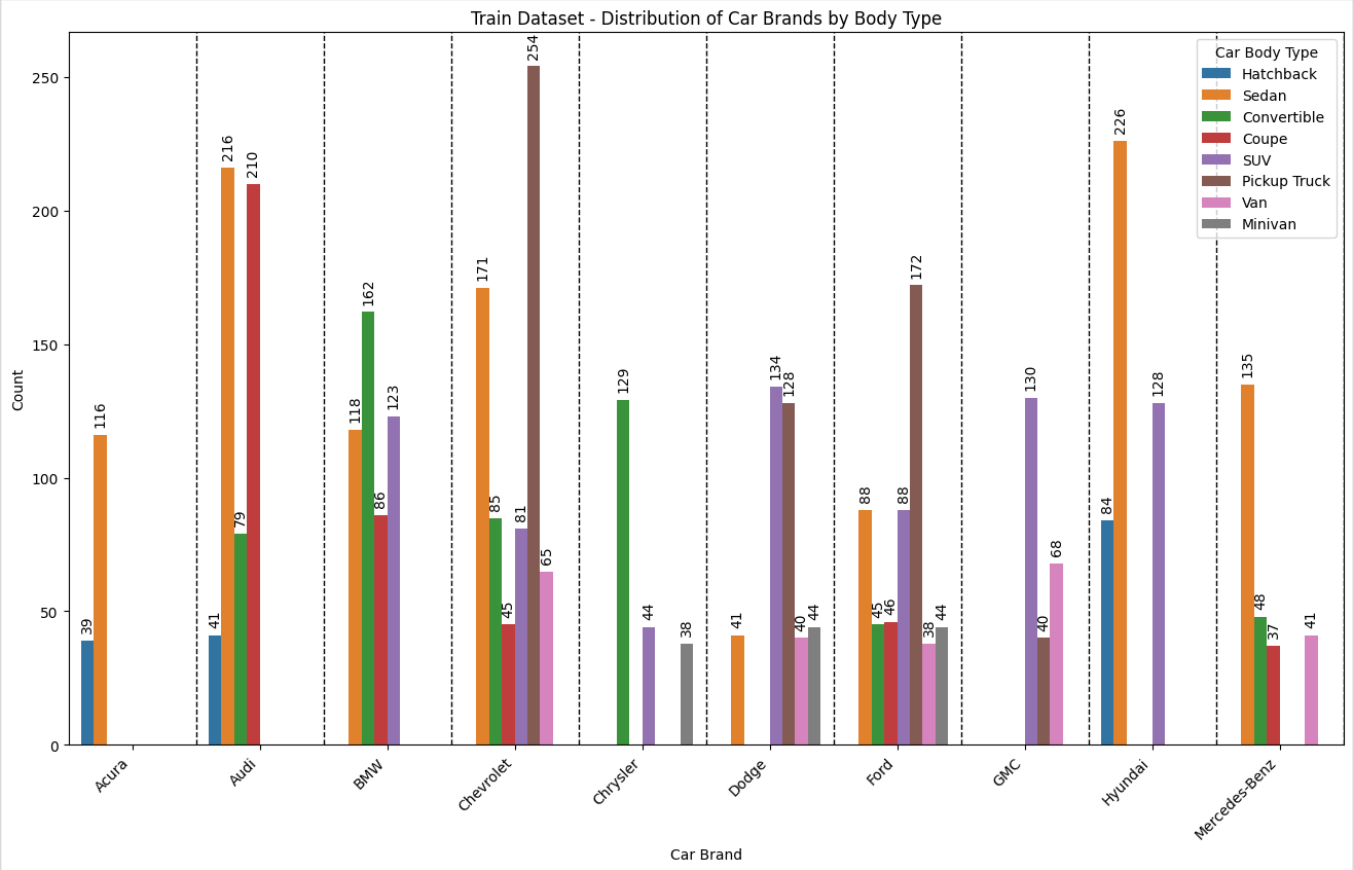
Car body type is mapped to car names using a list of keywords that is captured by the below dictionary. It can be seen that ‘Sedan’, ‘SUV’, ‘Coupe’ and ‘Convertible’ make 75% of both train and test dataset

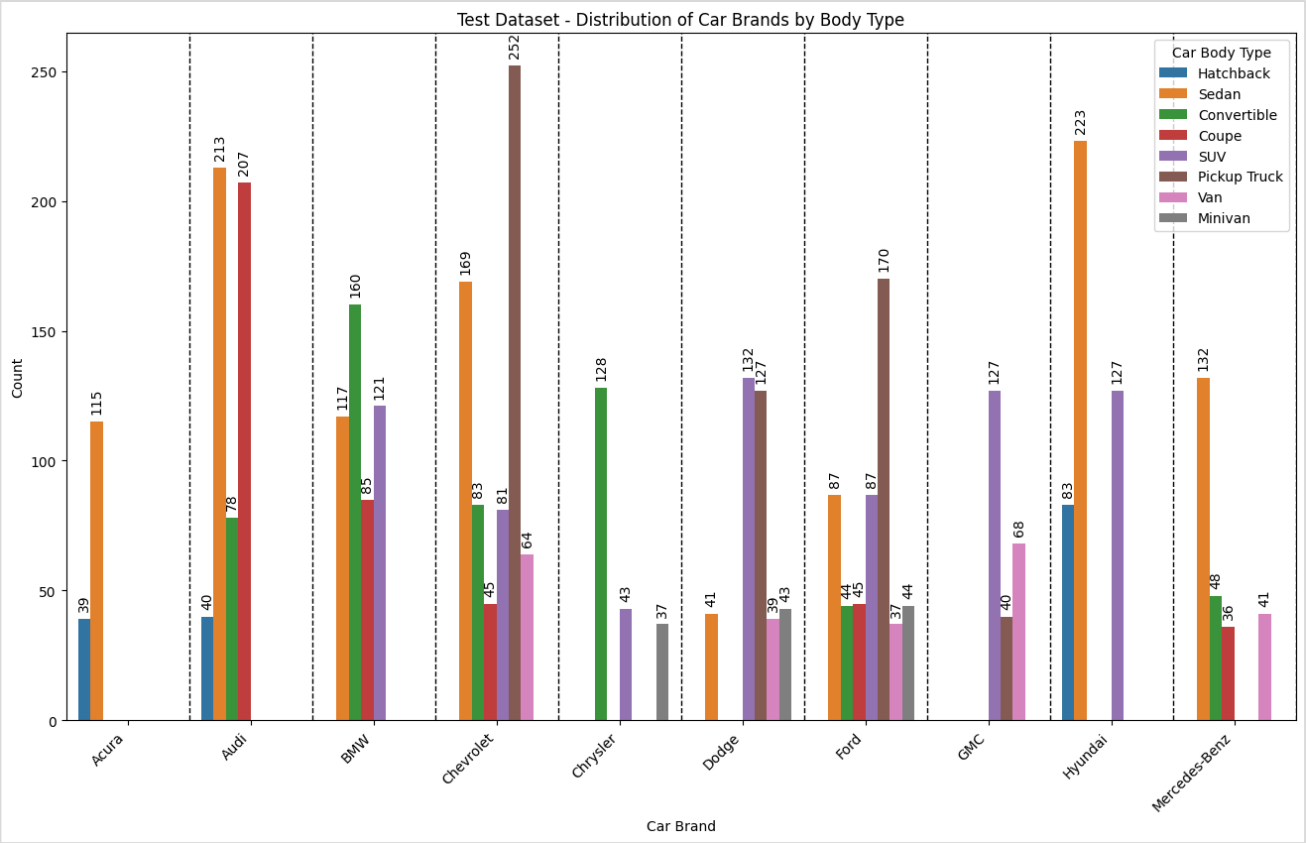




2.4 EDA – Car types within Top Brands

Brands are mapped to the images using keywords on the car names. The below analysis is done to provide a view wherein it can be seen that the 'Chrevolet' is the most occurring brand wherein 'Pickup Truck' is the most occurring car type followed by 'Audi' wherein ' Sedan' and 'coupe' are the most occurring body type





3. IMAGE PRE-PROCESSING

Initially, the images are mapped to their classes and bounding box co-ordinates using the following functions:

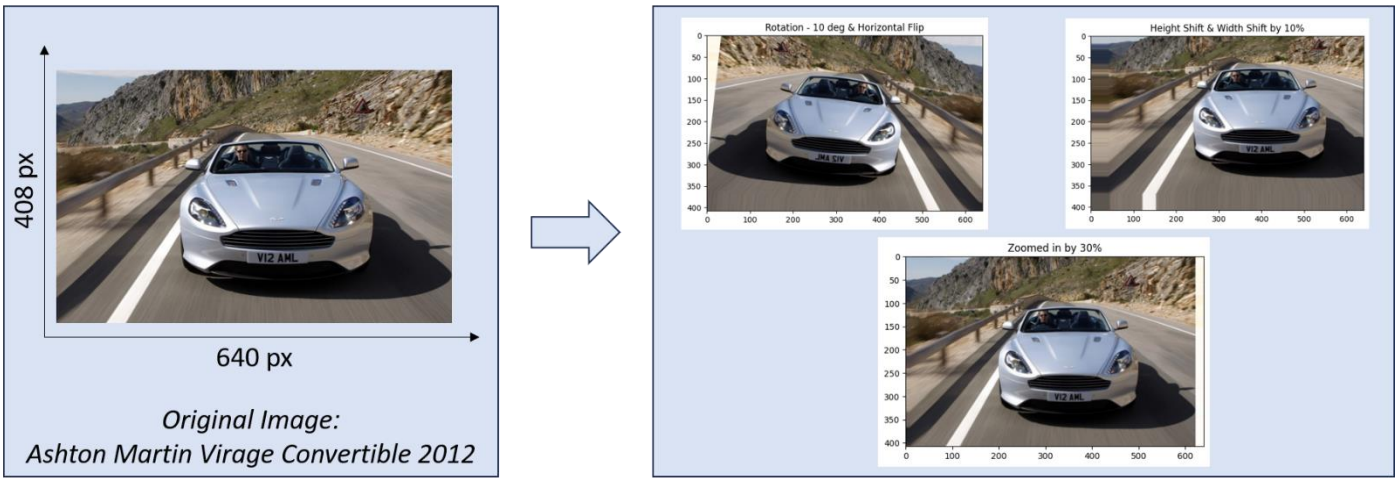
```
def get_image_class_map(image_path):
    image_class_map = {}
    for root, dirs, files in os.walk(image_path):
        # Filter out hidden directories
        dirs[:] = [d for d in dirs if not d.startswith('.')]
        for file in files:
            if file.endswith('.jpg') or file.endswith('.png'):
                class_name = os.path.basename(root)
                image_class_map[file] = class_name
    return image_class_map

train_image_class_map = get_image_class_map(train_images_path)
test_image_class_map = get_image_class_map(test_images_path)
```

```
def get_image_annotations(image_class_map, annotations):
    image_annotations_map = {}
    for index, row in annotations.iterrows():
        image_name = row['image_name'] ## car image name
        image_class = row['image_class'] ## car class - number
        if image_name in image_class_map:
            bbox = [row['x_min'], row['y_min'], row['x_max'], row['y_max']]
            image_car_name = image_class_map[image_name]
            image_annotations_map[image_name] = (bbox, image_car_name,image_class)
    return image_annotations_map

train_image_annotations_map = get_image_annotations(train_image_class_map, train_annotations)
test_image_annotations_map = get_image_annotations(test_image_class_map, test_annotations)
```

The following operations are performed on the images for model training:

1. Resizing
2. Normalization
3. Data Augmentation
4. Class imbalance rectification

Training and testing images are resized to a consistent size and their pixel values are normalized. To address class imbalance, images from classes with fewer samples are augmented using ImageDataGenerator to generate variations until the number of images matches the class with the highest count. The augmentation process includes various operations such as rotation, width and height shifts, zooming, flipping, shearing, and adjustments to brightness and contrast.



4. CLASSIFICATION MODEL

4.1. Model Building

A basic CNN model is built from scratch for image classification. The initial layers of the CNN extract generic features, while the deeper layers capture more specific features related to individual images. A neural network layer is then constructed on top of these CNN layers, adjusting the weights applied to the feature maps to classify the car images accurately.

4.2. CNN Models

Two CNN models are created: one with 7.4 million parameters and another with 25.4 million parameters. The first model was trained using the given training dataset whereas the second model was trained on upsampled train dataset

S.No	Parameters	CNN Model 1	CNN Model 2
1	Train dataset	8,144 train dataset	10,780 train dataset (196*68*80%)
2	Validation dataset	8,041 test dataset	2,548 train dataset (196*68*20%)
3	Weights	Default	HELinear
4	Activation Functions	ReLU	ReLU
5	Optimizers	Adam	SGD

6	Batch Size	32	32
7	Epochs	30	15

CNN Model 1:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 126, 126, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 63, 63, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 61, 61, 64)        18,496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D) (None, 30, 30, 64)      0
_________________________________________________________________
flatten (Flatten)            (None, 57,600)            0
_________________________________________________________________
dense (Dense)                (None, 128)               7,372,928
_________________________________________________________________
dense_1 (Dense)              (None, 196)               25,284
=================================================================
Total params: 7,416,604
Trainable params: 7,416,604
Non-trainable params: 0
```

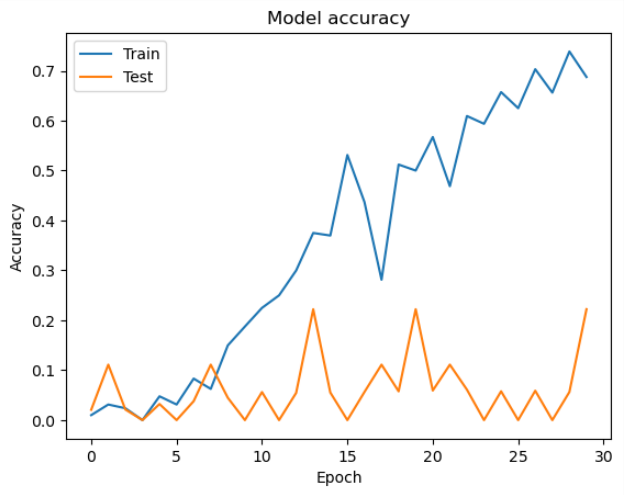
CNN Model 2:

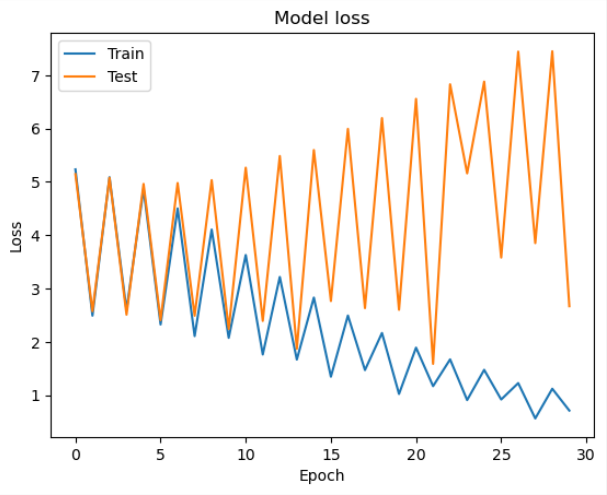
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_64_3by3 (Conv2D)      (None, 254, 254, 64)      1792
_________________________________________________________________
conv2d_64_3by3_1 (Conv2D)    (None, 252, 252, 64)      36928
_________________________________________________________________
activation (Activation)      (None, 252, 252, 64)      0
_________________________________________________________________
maxpool2d_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
conv2d_128_3by3 (Conv2D)     (None, 124, 124, 128)     73856
_________________________________________________________________
conv2d_128_3by3_1 (Conv2D)   (None, 122, 122, 128)     147584
_________________________________________________________________
activation_1 (Activation)    (None, 122, 122, 128)     0
_________________________________________________________________
maxpool2d_2 (MaxPooling2D)   (None, 61, 61, 128)       0
_________________________________________________________________
conv2d_512_3by3 (Conv2D)     (None, 59, 59, 512)       590336
_________________________________________________________________
activation_2 (Activation)    (None, 59, 59, 512)       0
_________________________________________________________________
maxpool2d_3 (MaxPooling2D)   (None, 29, 29, 512)       0
_________________________________________________________________
conv2d_512_3by3_1 (Conv2D)   (None, 27, 27, 512)       2359808
_________________________________________________________________
activation_3 (Activation)    (None, 27, 27, 512)       0
_________________________________________________________________
maxpool2d_4 (MaxPooling2D)   (None, 13, 13, 512)       0
_________________________________________________________________
flatten (Flatten)            (None, 86528)             0
_________________________________________________________________
dense (Dense)                (None, 256)               22133760
_________________________________________________________________
```

Observations:

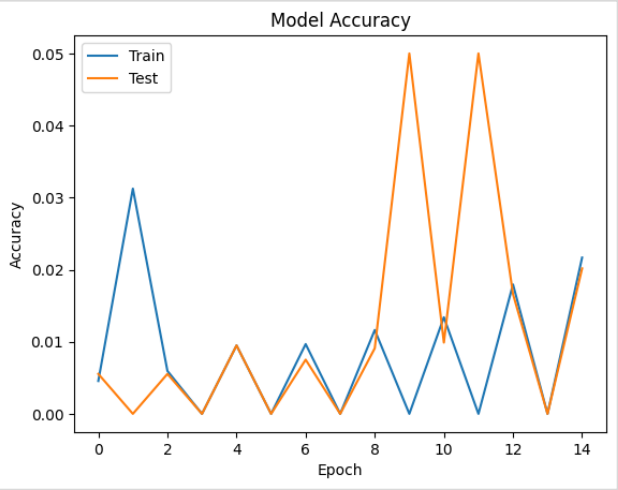
CNN Models:

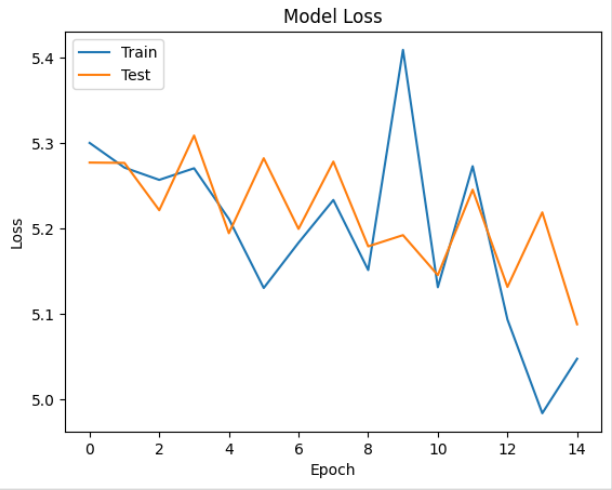
With CNN Model 1, the training accuracy consistently reaches 75%, while the validation/test accuracy fluctuates and only reaches 22%. This discrepancy indicates that the model is overfitting to the training dataset.





With CNN Model 2, both the training and validation accuracy reach 2%. Given the high resolution of the images and the model's 25 million parameters, it shows moderate performance. Increasing the number of epochs could potentially improve accuracy for both datasets. In terms of losses, both the training and validation losses decrease steadily over the epochs.





Below is the list of a few classes that were predicted accurately to a certain extent.

Classes	Precision	Recall	F1-Score	Support
Acura Integra Type R 2001	0.02	0.02	0.02	44
Audi S5 Convertible 2012	**0.04**	0.02	0.03	42
Bentley Continental Flying Spur Sedan 2007	0.03	0.02	0.03	44

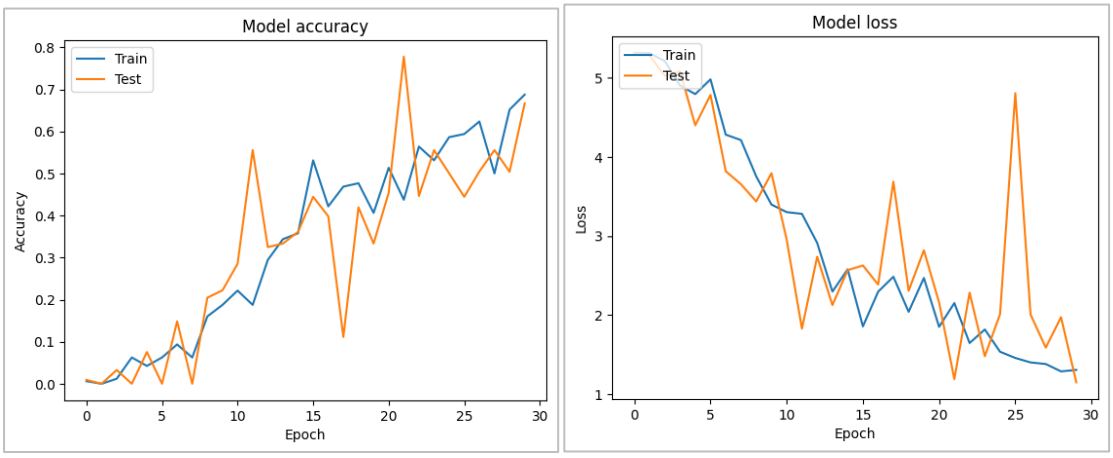
FIAT 500 Abarth 2012	0.01	**0.07**	0.01	27
Isuzu Ascender SUV 2008	**0.17**	0.03	0.04	40
Suzuki Kizashi Sedan 2012	**0.20**	0.02	0.04	46

4.3. Object Classification – Pre-trained models:

Pre-trained models like VGG16, ResNet, and GoogleNet were trained on these data with class imbalance and data from the upsampling techniques to improve accuracy, outperforming custom CNN models built from scratch.

S.No	Model Name	Dataset (trained on)	Train Accuracy	Test Accuracy
1	VGG16	Class-imbalanced dataset	7%	11%
		Class-balanced dataset	8%	14%
2	ResNet50	Class-imbalanced dataset	97%	41%
		Class-balanced dataset	44%	30%
3	GoogleNet	Class-imbalanced dataset	71%	46%
		Class-balanced dataset	76%	52%

Among all the models trained for object classification, GoogleNet performed the best, achieving a test accuracy of 52%. The train and test accuracies are closely aligned, indicating good generalization. The plot below shows the relationship between epochs and accuracy and losses for the GoogleNet model. It highlights that the accuracy for both training and validation data follows a similar trend, and the losses steadily decrease across epochs for both datasets.



5. R-CNN Model Building

5.1 Basic R-CNN model architecture

Data Preparation:

The Dataset class is designed to process each image individually by resizing it to a standard size of (224,224). It retrieves the bounding box/ground truth values for each image, scales them according to the resized image dimensions, and stores the processed images and bounding box values as tensors for further use.

ROIs Generation:

Regions of Interest (ROIs) are generated using the Selective Search algorithm. ROIs with a width or height smaller than 50 pixels are discarded, while the remaining ROIs are resized and passed to a CNN model for feature map extraction.

```
def selective_search(image, min_size=50):
    if image is None:
        raise ValueError("Image not loaded correctly. Check the file path.")

    # Create a selective search segmentation object
    ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
    ss.setBaseImage(image)
    ss.switchToSelectiveSearchFast()
    rects = ss.process()

    # Filter out regions based on the minimum size threshold
    filtered_rects = [rect for rect in rects if rect[2] > min_size and rect[3] > min_size]
    return filtered_rects
```

R-CNN Layers:

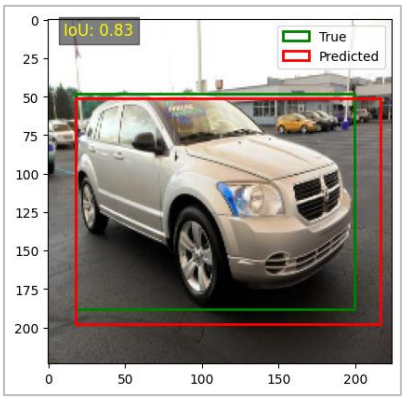
A simple CNN model with three convolutional layers is built to extract feature maps. The resulting feature maps are then flattened and passed through additional convolutional layers, which output four values representing the bounding box coordinates. 'Mean Squared Error' is the loss metrics calculated between the Ground Truth and predicted box coordinates and backpropagation is performed to minimize the loss and improve accuracy.

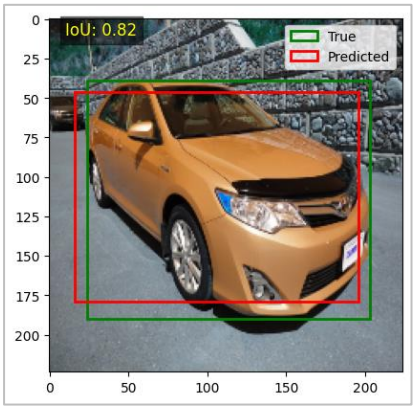
```
class SimpleCNN(nn.Module):
    def __init__(self, input_size=(224, 224)):
        super(SimpleCNN, self).__init__()
        self.input_size = input_size
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.AvgPool2d(kernel_size=2, stride=2),  # 112x112
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.AvgPool2d(kernel_size=2, stride=2),  # 56x56
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.AvgPool2d(kernel_size=2, stride=2)   # 28x28
        )
    def forward(self, x):
        if x.size(2) != self.input_size[0] or x.size(3) != self.input_size[1]:
            x = F.interpolate(x, size=self.input_size, mode='bilinear', align_corners=False)
        x = self.features(x)
        x = x.view(x.size(0), -1)  # Flatten the output
        return x
```

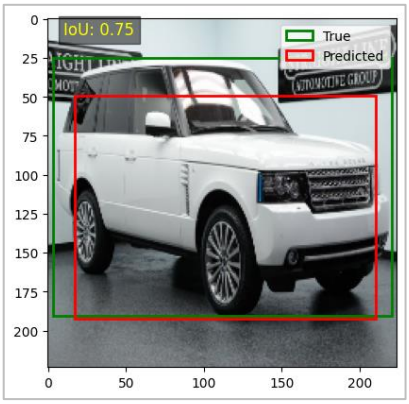
```
class BoundingBoxRegressor(nn.Module):
    def __init__(self, input_size):
        super(BoundingBoxRegressor, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 4)  # Predicting 4 coordinates (x1, y1, x2, y2)
        )
    def forward(self, x):
        return self.fc(x)
```

Model Accuracy:

The convolutional layers generate bounding box coordinates, which are compared with the ground truth coordinates, achieving an average IoU of 61% on a sample (around 20) of test images. The below screenshots present the images with true and predicted bounding boxes along with IoU mentioned.







5.2 Faster R-CNN Model:

A Faster R-CNN model is selected for training the data for object localization. Given that the dataset contains over 14k images, it is reduced to 2000 images—1000 for training and 1000 for testing. The images are processed in batches through a custom class that resizes them and converts the bounding box coordinates into tensors.

A faster R-CNN model instance (Faster R-CNN with ResNet backbone) is being called onto function and the output layers are modified to match the number of classes.

```
def get_model():
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained = True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features , num_classes)
    return model
```

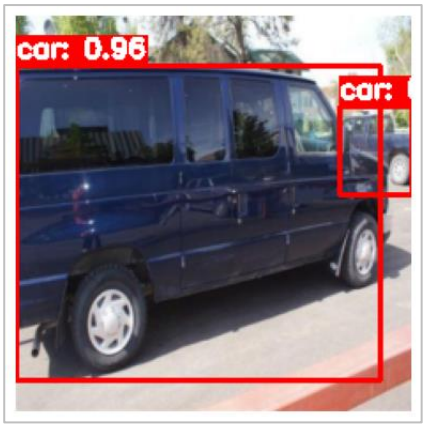
```
# Function for Creating Train loading data with image and target values.
class OpenDataset(Dataset):
    def __init__(self, df, image_dir):
        self.w, self.h = 224, 224
        self.image_dir = image_dir
        self.files = glob.glob( self.image_dir + '/*' )
        self.df = df.copy()
        self.image_infos = df.ImageID.unique()
        self.find_image_path = find(self.files)

    def __getitem__(self, ix):
        # load images and masks
        image_id = self.image_infos[ix]
        img_path = image_id
        img = Image.open(img_path).convert("RGB")
        data = self.df[self.df['ImageID'] == image_id]
        H, W , _ = np.array(img).shape
        # Normalise the dimension of the bounding boxes
        data.loc[:, ['x_min', 'y_min', 'x_max', 'y_max']] /= [W, H, W, H]
        # Normalise the value of the images from 0 to 1
        img = np.array(img.resize((self.w, self.h), resample=Image.BILINEAR))/255.
        labels = ['car'] * len(data)
        data = data[['x_min','y_min','x_max','y_max']].values
        # Regenerate the dimension of the bounding boxes of the new size (224,224)
        data[:,[0,2]] *= self.w
        data[:,[1,3]] *= self.h
        boxes = data.astype(np.uint32).tolist()
        target = {}
        # For each image we are creating the boxes and labels
        target["boxes"] = torch.Tensor(boxes).float()
        target["labels"] = torch.Tensor([label2target[i] for i in labels]).long()
        # Processing the image
        img = preprocess_image(img)
        # Return the image
        return img, target
    def collate_fn(self, batch):
        """ The Function takes a batch of data and rearranges it into a format suitable for processing. """
        return tuple(zip(*batch))
    def __len__(self):
        return len(self.image_infos)
```

Once the dataset is processed through the class 'Dataset' where the data is batched, and modified to match the input layers of the model and it is then passed to the model to be trained, and below screenshot gives the improvisations done during epochs based on the losses

```
Epoch 1/5, Loss: 0.0510, Loc Loss: 0.0176, Regr Loss: 0.0257, Objectness Loss: 0.0061, RPN Box Reg Loss: 0.0017
Epoch 1/5, Val Loss: 0.0376, Val Loc Loss: 0.0136, Val Regr Loss: 0.0202, Val Objectness Loss: 0.0008, Val RPN Box Reg Loss: 0.0030
Epoch 2/5, Loss: 0.0349, Loc Loss: 0.0134, Regr Loss: 0.0201, Objectness Loss: 0.0001, RPN Box Reg Loss: 0.0013
Epoch 2/5, Val Loss: 0.0291, Val Loc Loss: 0.0126, Val Regr Loss: 0.0133, Val Objectness Loss: 0.0002, Val RPN Box Reg Loss: 0.0029
Epoch 3/5, Loss: 0.0334, Loc Loss: 0.0124, Regr Loss: 0.0197, Objectness Loss: 0.0002, RPN Box Reg Loss: 0.0011
Epoch 3/5, Val Loss: 0.0304, Val Loc Loss: 0.0126, Val Regr Loss: 0.0126, Val Objectness Loss: 0.0025, Val RPN Box Reg Loss: 0.0026
Epoch 4/5, Loss: 0.0285, Loc Loss: 0.0106, Regr Loss: 0.0168, Objectness Loss: 0.0001, RPN Box Reg Loss: 0.0011
Epoch 4/5, Val Loss: 0.0207, Val Loc Loss: 0.0074, Val Regr Loss: 0.0106, Val Objectness Loss: 0.0002, Val RPN Box Reg Loss: 0.0024
Epoch 5/5, Loss: 0.0241, Loc Loss: 0.0091, Regr Loss: 0.0139, Objectness Loss: 0.0001, RPN Box Reg Loss: 0.0010
Epoch 5/5, Val Loss: 0.0230, Val Loc Loss: 0.0075, Val Regr Loss: 0.0128, Val Objectness Loss: 0.0005, Val RPN Box Reg Loss: 0.0023
<All keys matched successfully>
```

Model Performance:

The model is performing well on both the training and test datasets, demonstrating strong accuracy on the test set. Below are some successful object localizations along with their corresponding probability scores identified by the model.



The model with the best weights is saved as pytorch object for future predictions

```
# Saving the Faster R-CNN to a pytorch object
torch.save(model.state_dict(), "tut2-model.pt")

# Loading the Faster R-CNN onto the python notebook
model.load_state_dict(torch.load("tut2-model.pt"))
```

6. Observations:

Learnings:

- Building models from scratch provided a deeper understanding of the components that make up existing models like VGG16, GoogleNet, R-CNN, and YOLO. This knowledge is beneficial for fine-tuning parameters such as image resizing, weight initializers, optimizers, number of epochs, and activation functions.
- Acquiring hands-on experience in preparing data for various models involves working with different data structures and exploring model configurations.
- The project clarifies the challenges associated with training large datasets and highlights strategies for addressing any issues encountered.

Limitations of the project:

- **Class Imbalance:** The model may struggle with learning certain classes (e.g., different types of cars) if they are underrepresented in the dataset.
- **Pre-trained Weights:** Using weights from datasets like ImageNet or COCO might limit performance if those datasets don't match the specific case (e.g., cars). The model might not perform well on other objects like animals, humans, or furniture.
- **Enhancements:**
  - Increase the number of epochs and train on more diverse datasets.
  - Better manage class balance during training.
  - Choose a pre-trained model that has been trained on cars to boost accuracy.
  - Add more data variety and explore advanced model architectures to improve performance.

Recommendations:

- Utilizing transfer learning by fine-tuning pre-trained models on similar tasks to leverage existing knowledge and reduce training time. Models like YOLO and SSD can improve accuracy, as these models are trained on various object types and excel in vehicle detection.
- Collecting a wide variety of refined images for each class enhances the likelihood of achieving a successful model.

- Investigating and fine-tuning all model parameters can lead to better results. Systematically experimenting with different hyperparameters (e.g., learning rate, batch size) to find the optimal settings for model training.
- Using cross-validation to better understand the model’s performance and avoid overfitting by testing on multiple subsets of the data.
- Monitoring additional performance metrics (e.g., precision, recall, F1 score) beyond accuracy to gain a comprehensive understanding of the model's effectiveness.