Computer Vision

OBJECT DETECTION - CAR

A Capstone Project

Submitted by:

Lakshmiprasad Bhatta

Narasimha Rao

P C Sekhar Reddy

Ramsewar Jena

Vibhas Narayanan

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:
Venkatesha K H

February 2021

Table of Contents

1. Introduction

Computer vision has been greatly used to automate supervision and generate appropriate action and trigger if the event is predicted from the image of interest. For example, a car moving on the road can be easily identified by a camera as make of the car, type, color, number plates, etc.

Object detection is a computer vision technique in which a software system can detect, locate, and trace the object from a given image or video. The special attribute of object detection is that it identifies the class of object (person, table, chair, etc.) and their location-specific coordinates in the given image. The location is pointed out by drawing a bounding box around the object. The bounding box may or may not accurately locate the position of the object. The ability to locate the object inside an image defines the performance of the algorithm used for detection. This object detection application is widely used in the automotive and surveillance domains

1.1. Problem Statement

In this project, we have a 'Cars' dataset which contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, Year, e.g. 2012 Tesla Model S or 2012 BMW M3 Coupe. The aim of this project is to build a deep learning model to predict the bounding boxes around the car and the corresponding Class for the car for a randomly picked car image.

It will be a single object detection in an input image. GUI has been developed for end-user interface to input an image from a directory and get an output image with a corresponding label.

A deep learning model has been built using Open CV, TensorFlow, Keras. Pre-trained models like Mobile Net, VGG16, and DenseNet have been tried as base models to train the images along with bounding boxes and class labels.

CNN architecture has been built with dual targets; bounding boxes and classes. To predict both bounding box and class label together, a two-headed model has been created. A regression layers head to predict bounding boxes and a multi-class layers head to predict class labels. It needs a fully-connected layer head with two branches:

#1: A regression layer set, just like in the single-class object detection case

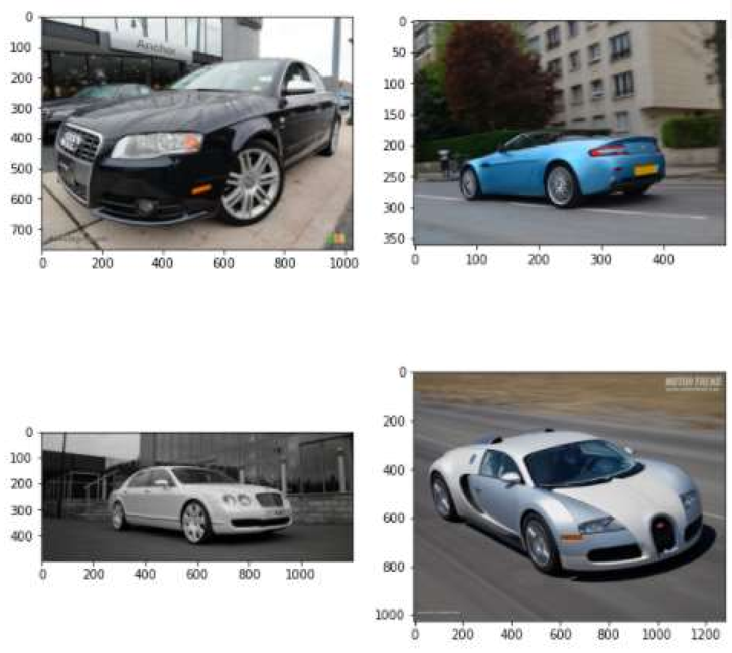
#2: An additional layer set, this one with a softmax classifier used to predict class labels

1.2. Exploring the Input data

Input data has 8,144 training images and 8,041 testing images. Few images were randomly chosen and plotted using Matplotlib imread. The images are having different dimensions in height and width, the cars are in different orientations, colors and backgrounds. A few of the images are displayed below and a few class names are also listed.

Sample Images:

```
<matplotlib.image.AxesImage at 0x7f16722fdba8>
```



Class names: Make Model Year

```
class_names[:200:10]
```

```
['AM General Hummer SUV 2000',
 'Aston Martin Virage Coupe 2012',
 'Audi S5 Convertible 2012',
 'BMW 6 Series Convertible 2007',
 'Bentley Mulsanne Sedan 2011',
 'Cadillac CTS-V Sedan 2012',
 'Chevrolet Impala Sedan 2007',
 'Chevrolet Express Van 2007',
 'Chrysler PT Cruiser Convertible 2008',
 'Dodge Dakota Club Cab 2007',
 'Ferrari FF Coupe 2012',
 'Ford Ranger SuperCab 2011',
 'GMC Acadia SUV 2012',
 'Hyundai Santa Fe SUV 2012',
 'Infiniti G Coupe IPL 2012',
 'Lamborghini Aventador Coupe 2012',
 'Mercedes-Benz 300-Class Convertible 1993',
 'Nissan 240SX Coupe 1998',
 'Suzuki Aerio Sedan 2007',
 'Volkswagen Golf Hatchback 1991']
```

The annotations for both train and test data are given in the csv format. Each csv file contains the following information; Image name, corresponding bounding box coordinates and the class label of the car in the image.

Annotations for the first 5 images from Train Annotations.csv are shown below.

Image Name	Bounding Box coordinates				Image class
00001.jpg	39	116	569	375	14
00002.jpg	36	116	868	587	3
00003.jpg	85	109	601	381	91
00004.jpg	621	393	1484	1096	134
00005.jpg	14	36	133	99	106

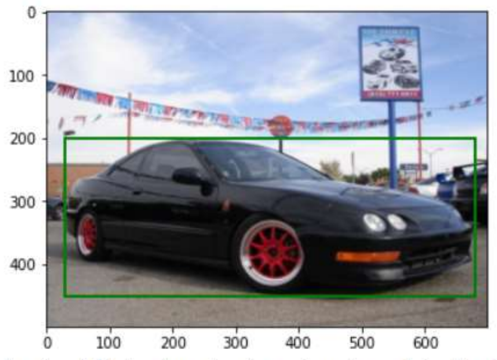
1.3 Mapping Images to their Classes

The image class names are the same as the folder names. In this step, all the images are mapped to their respective classes i.e., folder paths. Used the glob utility to read data through the folders. Created separated lists for image paths, bounding boxes, and labels and in the later step, they are converted to arrays.

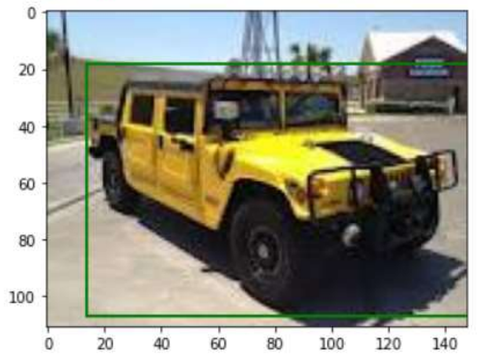
```
trainImagePaths=[]
trainBboxes =[]
trainLabels=[]

train_folder=glob.glob(dataset_folder + 'train/*/*')
for j in range(len(train_folder)):
  index=file_names.index(train_folder[j].split('/')[-1])

  (x1,y1,x2,y2)=bboxes[index]
  filename=train_folder[j].split('/')[-1]
  classname=train_folder[j].split('/')[-2]
  file_path=os.path.join(dataset_folder + 'train/'+classname+'/'+filename)
  trainImagePaths.append(file_path) ## image paths
  trainLabels.append(classname) ## image labes
```

After mapping is done, few images with their bounding boxes are plotted as shown below.

```
/content/drive/MyDrive/Capstone/car_data/train/Acura Integra Type R 2001/07800.jpg
<class 'str'>
```



```
/content/drive/MyDrive/Capstone/car_data/train/AM General Hummer SUV 2000/07684.jpg
<class 'str'>
```

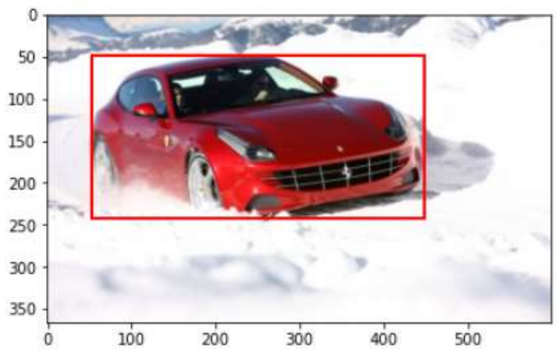


2. Data Preparation for the Model Building

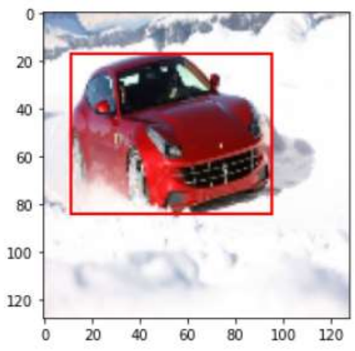
In the data preparation, all the images need to be resized as required for model input and accordingly, their bounding boxes also need to be resized. We need to convert the labels to one hot encoding. With that, the data will be ready to be used for the deep learning model.

Here is the information of what we started with for the model building

- the original train images are resized to 128X128 and used directly without augmentation for the model training



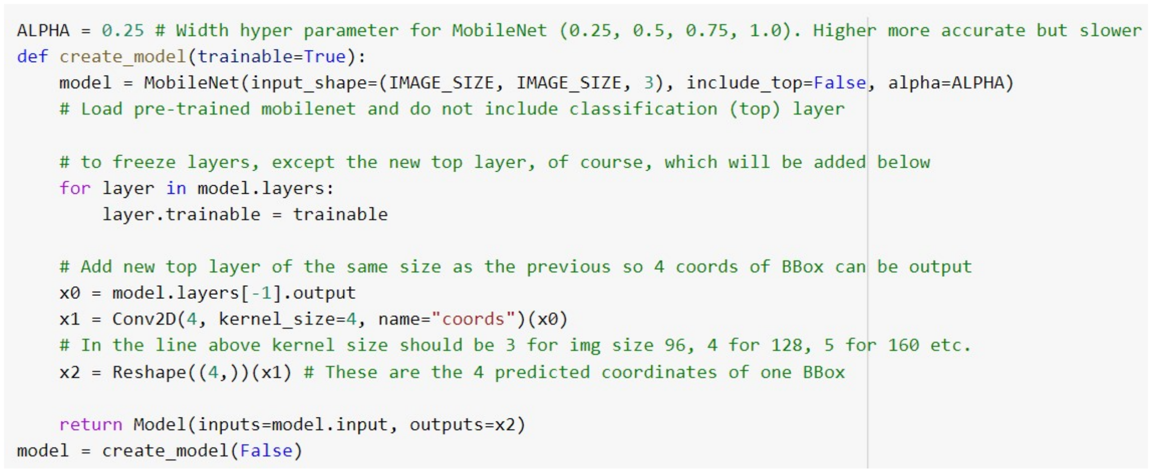
Original Image



Resized image

- Converted the images to cv2 arrays and preprocessed the images for standardization
- Resized the bounding boxes according to the resized images
- Two different CNN models were used for the Regression and Classification
- Mobile Net Regression model has used bounding boxes and Dense Net Classification model used for label prediction

Mobile Net CNN model for the Regression:



```
ALPHA = 0.25 # Width hyper parameter for MobileNet (0.25, 0.5, 0.75, 1.0). Higher more accurate but slower
def create_model(trainable=True):
    model = MobileNet(input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3), include_top=False, alpha=ALPHA)
    # Load pre-trained mobilenet and do not include classification (top) layer

    # to freeze layers, except the new top layer, of course, which will be added below
    for layer in model.layers:
        layer.trainable = trainable

    # Add new top layer of the same size as the previous so 4 coords of BBox can be output
    x0 = model.layers[-1].output
    x1 = Conv2D(4, kernel_size=4, name="coords")(x0)
    # In the line above kernel size should be 3 for img size 96, 4 for 128, 5 for 160 etc.
    x2 = Reshape((4,))(x1) # These are the 4 predicted coordinates of one BBox

    return Model(inputs=model.input, outputs=x2)
model = create_model(False)
```

```
reg_model.compile(loss="mean_squared_error", optimizer="adam", metrics=[IoU]) # Regression loss is MSE
```

Dense Net CNN model for the classification:

```
def build_model():
    base_model = densenet.DenseNet121(input_shape=(img_width, img_height, 3),
                                     weights='G:/AIML_UTA/15_Capstone_Project/ref/densenet121_weights_tf_dim_ordering_tf_kernels_notop.h5',
                                     include_top=False,
                                     pooling='avg')
    for layer in base_model.layers:
        layer.trainable = False

    x = base_model.output
    x = Dense(1000, kernel_regularizer=regularizers.l1_l2(0.01), activity_regularizer=regularizers.l2(0.01))(x)
    x = Activation('relu')(x)
    x = Dense(500, kernel_regularizer=regularizers.l1_l2(0.01), activity_regularizer=regularizers.l2(0.01))(x)
    x = Activation('relu')(x)
    predictions = Dense(n_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=predictions)

    return model
model = build_model()
```

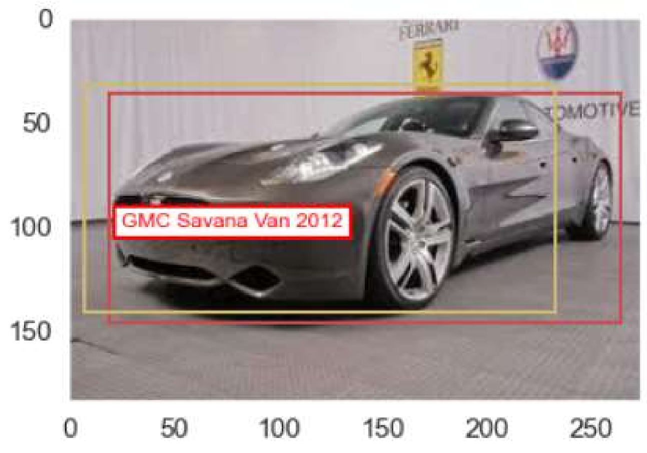
```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc', 'mse'])
```

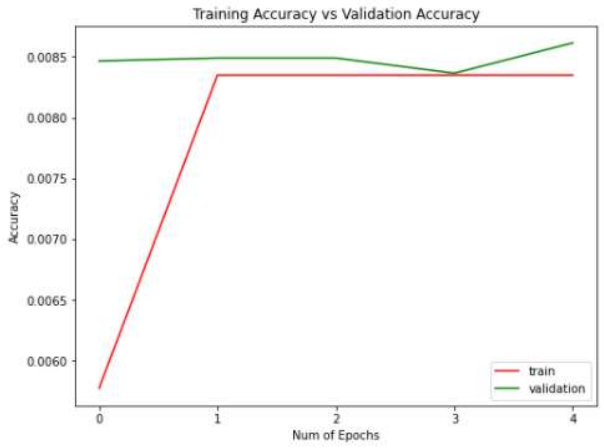
- For the model evaluation metrics, Intersection of Union (IoU), Mean squared error (MSE) is used for Regression and Categorical cross-entropy is used for classification.
- Batch size 32 and epochs 5 are used for both models.
- Early stopping is used for callbacks for both the models

```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=5, batch_size=32, callbacks=[callback])
```

- CPU laptop with I7 and 8GB RAM is used to run the code

The below picture shows bounding box prediction (yellow box) which is very close to the original bounding box (red box) but it shows incorrect label prediction.





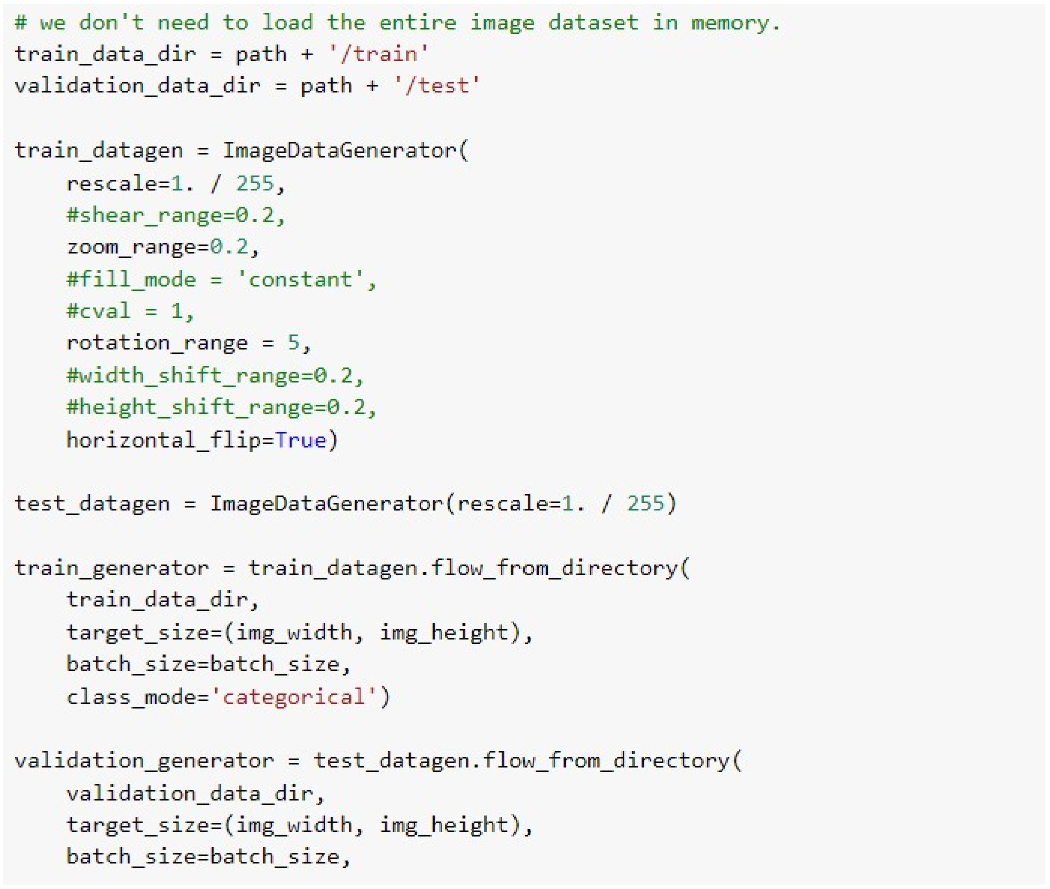
For the Classification model, the accuracy is < 1% as shown in the picture.

To improve the classification accuracy, we need to increase the number of epochs and try a different model. Also, we need to train more images with the use of data augmentation techniques.

2.1. Image Data Augmentation

Image augmentation increases the complexity for model training and prevents overfitting of the model and improves the validation class accuracy.

ImagaDataGenarator can be used to augment the images and generate more input images for the model training. It generates batches of tensor image data with real-time data augmentation. It applies a transformation to an image according to given parameters. “flow_from_directory” function takes the path to a directory & generates batches of augmented data. Here is the code for the ImageDataGenerator.



```
# we don't need to load the entire image dataset in memory.
train_data_dir = path + '/train'
validation_data_dir = path + '/test'

train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    #shear_range=0.2,
    zoom_range=0.2,
    #fill_mode = 'constant',
    #cval = 1,
    rotation_range = 5,
    #width_shift_range=0.2,
    #height_shift_range=0.2,
    horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1. / 255)

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

validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
```

ImageDataGenarator does the augmentation only for the images and not for the bounding boxes. And since it is used only for the categorical prediction, it can be used only to improve the class label accuracy and a second model is still required to predict the bounding box predictions.

Since it is recommended to use a single model for predicting both bounding boxes and class label, we found a way to use a two head CNN model which can be attached to a pretrained deep learning model. This modified CNN model allows us to pass both bounding boxes and class labels as targets along with the input images.

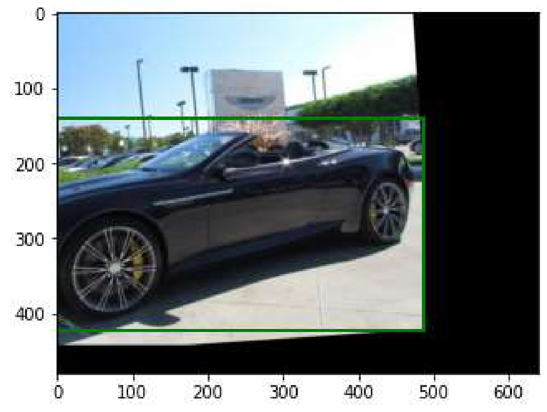
Due to the limitation of the ImageDataGenarator to use only for the classification, we found another way to augment the images along with bounding box realignment which is explained below. The augmented data can be combined with the original train data and input to the model with both bounding boxes and class labels together.

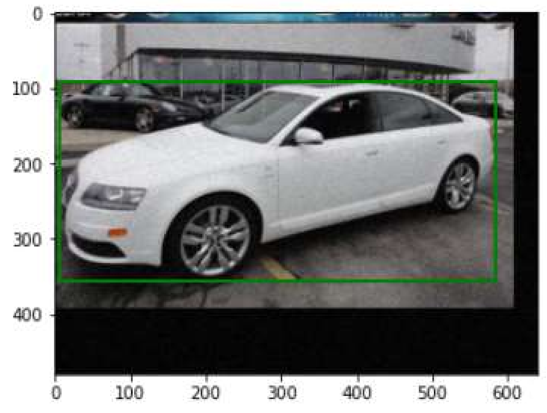
To augment the images and realign their corresponding bounding boxes, we used the following packages and the augmentor functions.

```
## packages related to image Augmeantation
import imgaug as ia
ia.seed(1)
from imgaug.augmentables.bbs import BoundingBox, BoundingBoxesOnImage
from imgaug import augmenters as iaa

# function to convert BoundingBoxesOnImage object into DataFrame
def bbs_obj_to_df(bbs_object):
#     convert BoundingBoxesOnImage object into array
    bbs_array = bbs_object.to_xyxy_array()
#     convert array into a DataFrame ['xmin', 'ymin', 'xmax', 'ymax'] columns
    df_bbs = pd.DataFrame(bbs_array, columns=['xmin', 'ymin', 'xmax', 'ymax'])
    return df_bbs
```

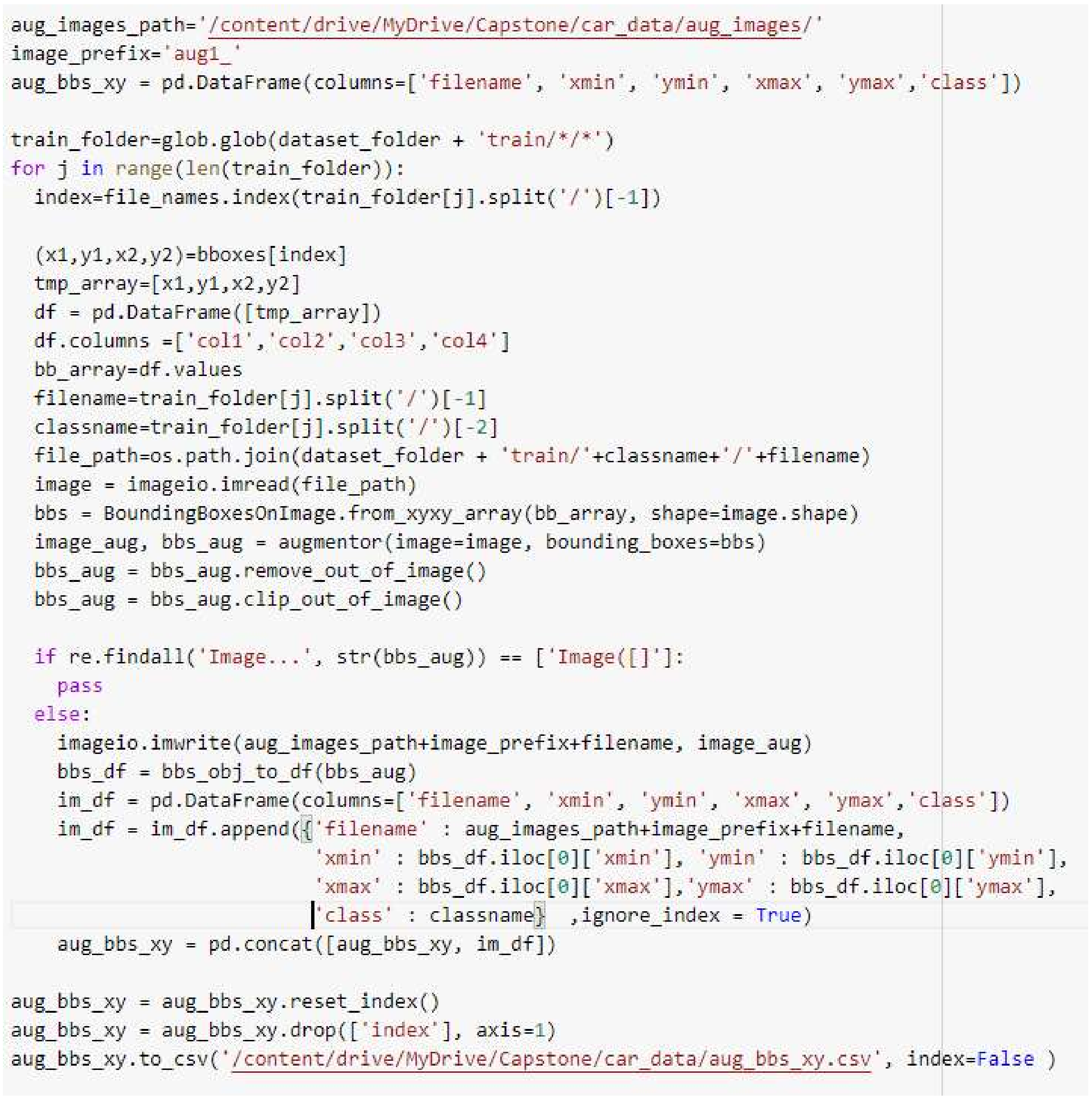
```
# This setup of augmentation parameters will pick two of four given augmenters
augmentor = iaa.SomeOf(2, [
    iaa.Affine(scale=(0.5, 1.5)),
    iaa.Affine(rotate=(-60, 60)),
    iaa.Affine(translate_percent={"x": (-0.3, 0.3), "y": (-0.3, 0.3)}),
    iaa.Fliplr(1),
    iaa.Multiply((0.5, 1.5)),
    iaa.GaussianBlur(sigma=(1.0, 3.0)),
    iaa.AdditiveGaussianNoise(scale=(0.03*255, 0.05*255))
])
```

With help of GPU capabilities from cloud services we could augment all 8144 train images and realign their corresponding bounding boxes. Here is the display of few images to check if the bounding boxes are correctly overlayed on the augmented images.





The following code is used to augment the images and bounding boxes together.



```
aug_images_path='/content/drive/MyDrive/Capstone/car_data/aug_images/'
image_prefix='aug1_'
aug_bbs_xy = pd.DataFrame(columns=['filename', 'xmin', 'ymin', 'xmax', 'ymax','class'])

train_folder=glob.glob(dataset_folder + 'train/*/*')
for j in range(len(train_folder)):
  index=file_names.index(train_folder[j].split('/')[-1])

  (x1,y1,x2,y2)=bboxes[index]
  tmp_array=[x1,y1,x2,y2]
  df = pd.DataFrame([tmp_array])
  df.columns =['col1','col2','col3','col4']
  bb_array=df.values
  filename=train_folder[j].split('/')[-1]
  classname=train_folder[j].split('/')[-2]
  file_path=os.path.join(dataset_folder + 'train/'+classname+'/'+filename)
  image = imageio.imread(file_path)
  bbs = BoundingBoxesOnImage.from_xyxy_array(bb_array, shape=image.shape)
  image_aug, bbs_aug = augmentor(image=image, bounding_boxes=bbs)
  bbs_aug = bbs_aug.remove_out_of_image()
  bbs_aug = bbs_aug.clip_out_of_image()

  if re.findall('Image...', str(bbs_aug)) == ['Image([]']:
    pass
  else:
    imageio.imwrite(aug_images_path+image_prefix+filename, image_aug)
    bbs_df = bbs_obj_to_df(bbs_aug)
    im_df = pd.DataFrame(columns=['filename', 'xmin', 'ymin', 'xmax', 'ymax','class'])
    im_df = im_df.append({'filename' : aug_images_path+image_prefix+filename,
                          'xmin' : bbs_df.iloc[0]['xmin'], 'ymin' : bbs_df.iloc[0]['ymin'],
                          'xmax' : bbs_df.iloc[0]['xmax'],'ymax' : bbs_df.iloc[0]['ymax'],
                          'class' : classname}  ,ignore_index = True)
    aug_bbs_xy = pd.concat([aug_bbs_xy, im_df])

aug_bbs_xy = aug_bbs_xy.reset_index()
aug_bbs_xy = aug_bbs_xy.drop(['index'], axis=1)
aug_bbs_xy.to_csv('/content/drive/MyDrive/Capstone/car_data/aug_bbs_xy.csv', index=False )
```

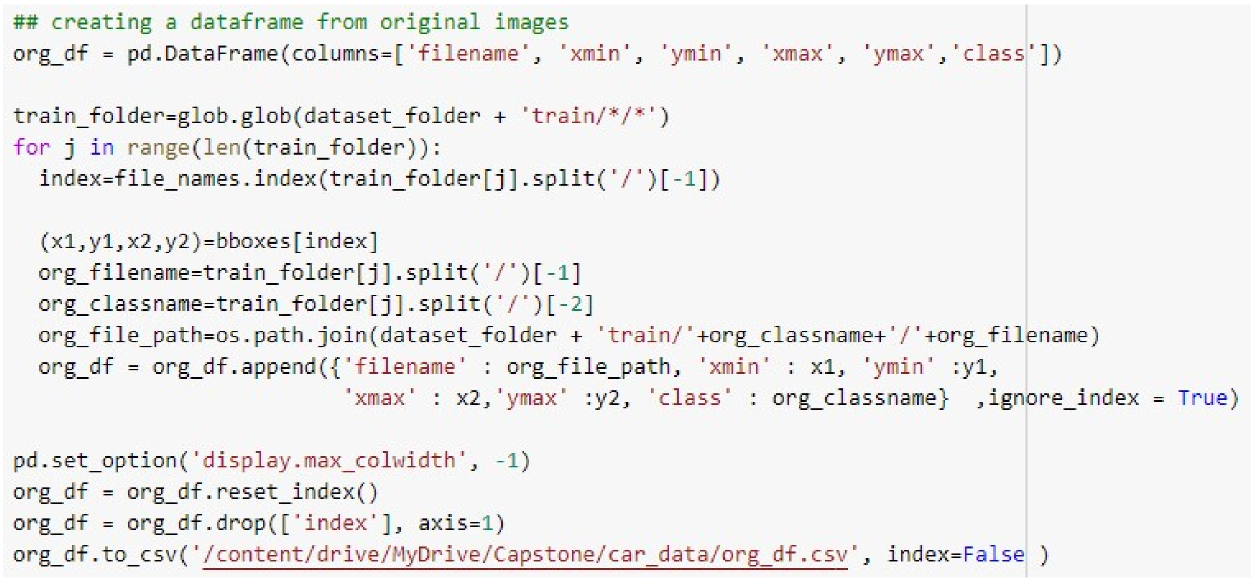
The glob module is used to retrieve files/pathnames matching a specified pattern. The pattern rules of glob follow standard Unix path expansion rules. It is faster than other methods to match pathnames in directories. With glob, we can also use wildcards ("*, ?, [ranges]) apart from exact string search to make path retrieval more simple and convenient.

All the augmented images are saved to a different folder "aug_images". Now combining these images with the original train images, the input train images are doubled.

Also, the corresponding bounding boxes and labels have been saved to a CSV file. These are further combined with the original train images and bounding boxes which is explained in the next step.

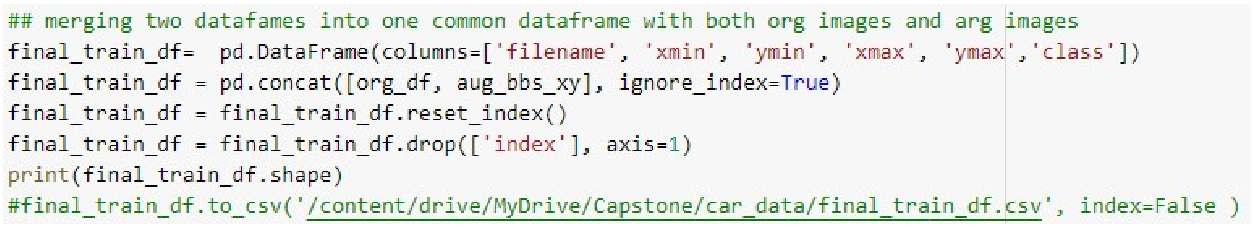
2.2. Combining the Training Images

In this step, the following code is used to create the pandas DataFrame with filenames, bounding boxes and classes for the original train images. And then, combine it with augmented images DataFrame.



```
## creating a dataframe from original images
org_df = pd.DataFrame(columns=['filename', 'xmin', 'ymin', 'xmax', 'ymax','class'])

train_folder=glob.glob(dataset_folder + 'train/*/*')
for j in range(len(train_folder)):
  index=file_names.index(train_folder[j].split('/')[-1])

  (x1,y1,x2,y2)=bboxes[index]
  org_filename=train_folder[j].split('/')[-1]
  org_classname=train_folder[j].split('/')[-2]
  org_file_path=os.path.join(dataset_folder + 'train/'+org_classname+'/'+org_filename)
  org_df = org_df.append({'filename' : org_file_path, 'xmin' : x1, 'ymin' :y1,
                          'xmax' : x2,'ymax' :y2, 'class' : org_classname}  ,ignore_index = True)

pd.set_option('display.max_colwidth', -1)
org_df = org_df.reset_index()
org_df = org_df.drop(['index'], axis=1)
org_df.to_csv('/content/drive/MyDrive/Capstone/car_data/org_df.csv', index=False )
```

Combining the dataframes for augmented images and the original images:



```
## merging two datafames into one common dataframe with both org images and arg images
final_train_df=  pd.DataFrame(columns=['filename', 'xmin', 'ymin', 'xmax', 'ymax','class'])
final_train_df = pd.concat([org_df, aug_bbs_xy], ignore_index=True)
final_train_df = final_train_df.reset_index()
final_train_df = final_train_df.drop(['index'], axis=1)
print(final_train_df.shape)
#final_train_df.to_csv('/content/drive/MyDrive/Capstone/car_data/final_train_df.csv', index=False )
```

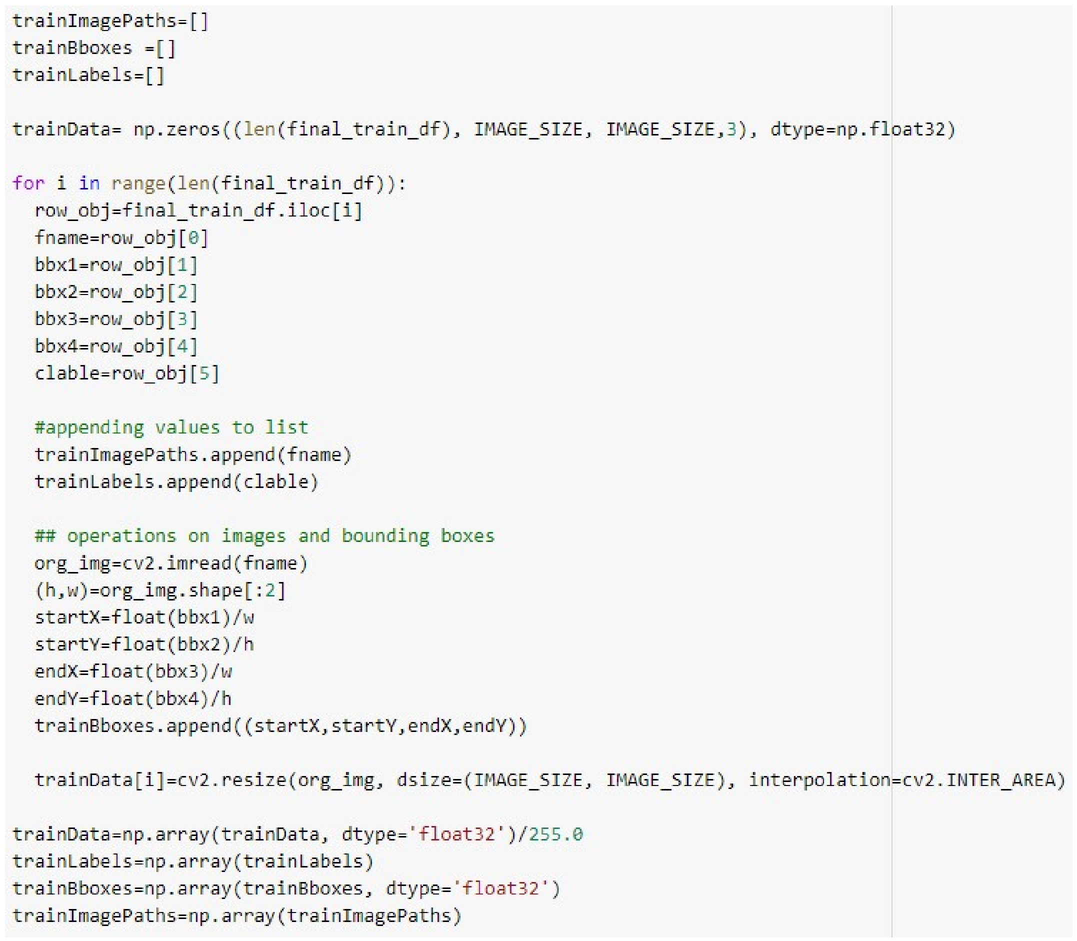
Combined 16288 train images and their corresponding bounding boxes are now ready for the data preprocessing before passing them for model training.

For the validation, we can preprocess the original 8041 test images and bounding boxes without any augmentation.

The detailed processing of the images and the bounding boxes are explained in the next step.

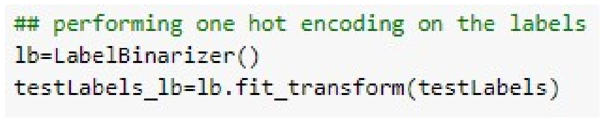
2.3. Image Data Preprocessing

In this step, all the train images and test images are resized to 224 X 224 as required for the model input. Corresponding bounding boxes are also resized accordingly. The images are converted to cv2 arrays and further standardized by dividing with 255. All the image arrays, bounding boxes, labels, and folder paths have been saved into numpy arrays.



```
trainImagePaths=[]
trainBboxes =[]
trainLabels=[]

trainData= np.zeros((len(final_train_df), IMAGE_SIZE, IMAGE_SIZE,3), dtype=np.float32)

for i in range(len(final_train_df)):
  row_obj=final_train_df.iloc[i]
  fname=row_obj[0]
  bbx1=row_obj[1]
  bbx2=row_obj[2]
  bbx3=row_obj[3]
  bbx4=row_obj[4]
  clable=row_obj[5]

  #appending values to list
  trainImagePaths.append(fname)
  trainLabels.append(clable)

  ## operations on images and bounding boxes
  org_img=cv2.imread(fname)
  (h,w)=org_img.shape[:2]
  startX=float(bbx1)/w
  startY=float(bbx2)/h
  endX=float(bbx3)/w
  endY=float(bbx4)/h
  trainBboxes.append((startX,startY,endX,endY))

  trainData[i]=cv2.resize(org_img, dsize=(IMAGE_SIZE, IMAGE_SIZE), interpolation=cv2.INTER_AREA)

trainData=np.array(trainData, dtype='float32')/255.0
trainLabels=np.array(trainLabels)
trainBboxes=np.array(trainBboxes, dtype='float32')
trainImagePaths=np.array(trainImagePaths)
```

LabelBinarazer is used to convert all 196 class labels to hot encoding. Label Encoding is an important pre-processing step, it refers to converting the labels into the numeric form to convert it into the machine-readable form.



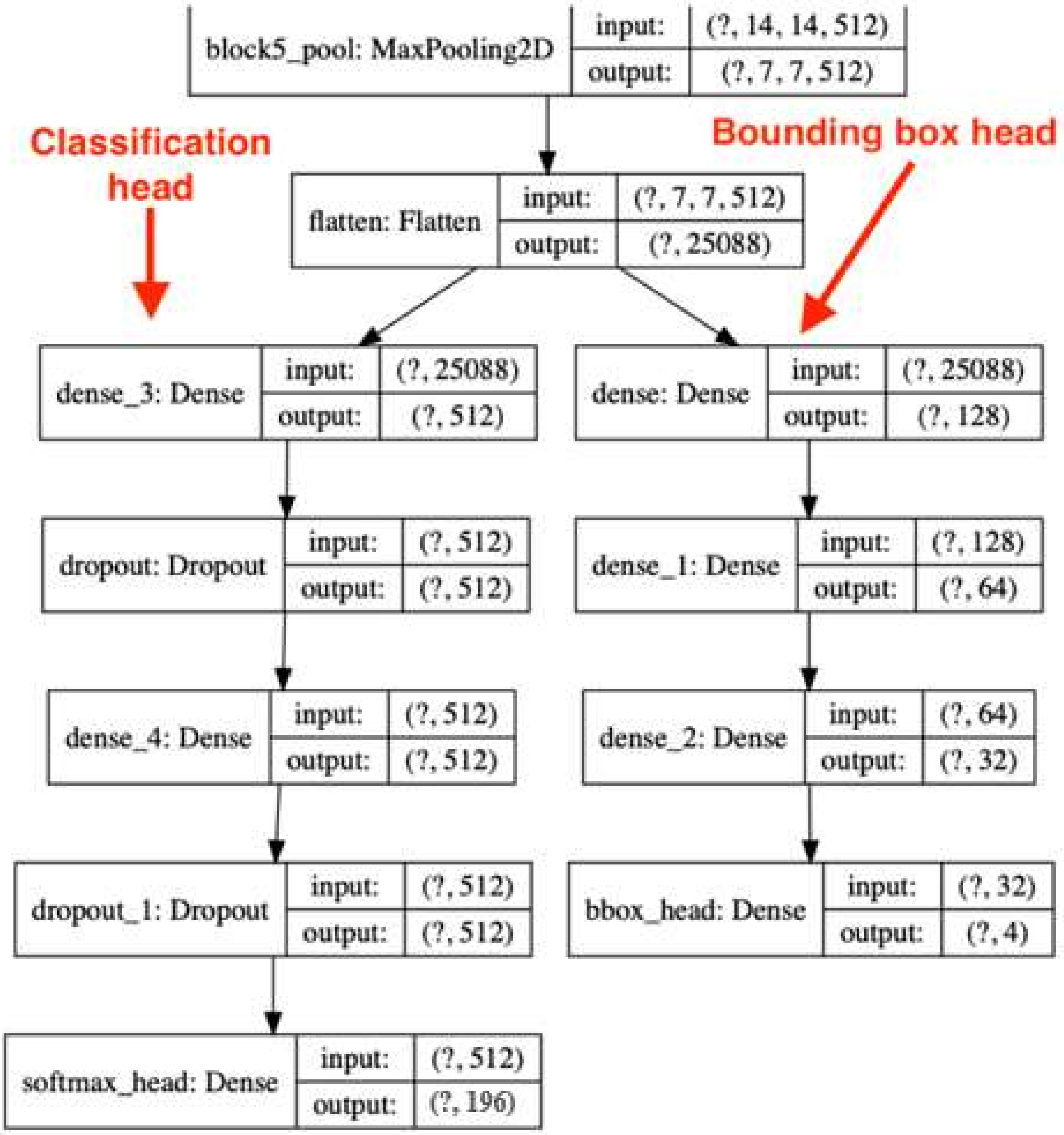
```
## performing one hot encoding on the labels
lb=LabelBinarizer()
testLabels_lb=lb.fit_transform(testLabels)
```

The same above procedure has been followed for the test images and saved into numpy arrays. With that, both train and test arrays are ready for the model input.

3. Model Building

Single model architecture is used to train both the Regression and Classification with two different heads attached to the pre-trained CNN model head.

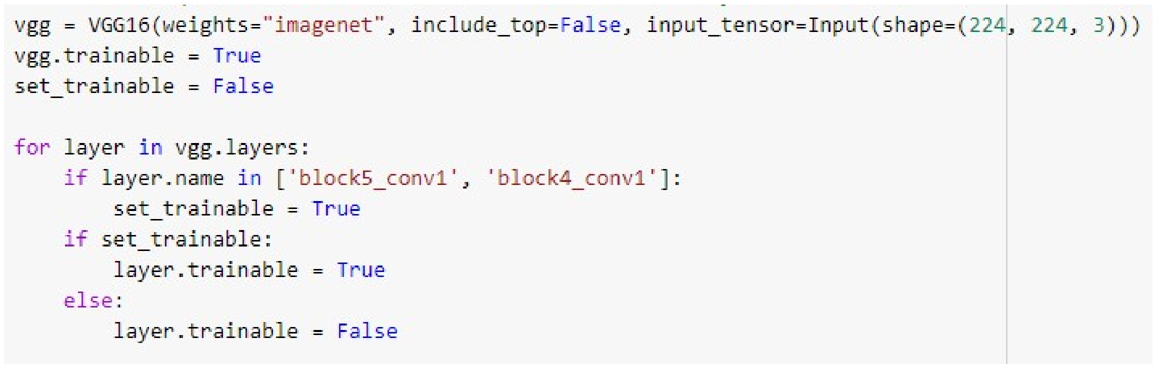
The figure below shows the visualization of two-headed CNN model. The classification head (left) outputs the class label for the corresponding bounding box prediction (right).:


block5_pool: MaxPooling2D | input: (?, 14, 14, 512) | output: (?, 7, 7, 512)
Classification head
Bounding box head
flatten: Flatten | input: (?, 7, 7, 512) | output: (?, 25088)
dense_3: Dense | input: (?, 25088) | output: (?, 512)
dense: Dense | input: (?, 25088) | output: (?, 128)
dropout: Dropout | input: (?, 512) | output: (?, 512)
dense_1: Dense | input: (?, 128) | output: (?, 64)
dense_4: Dense | input: (?, 512) | output: (?, 512)
dense_2: Dense | input: (?, 64) | output: (?, 32)
dropout_1: Dropout | input: (?, 512) | output: (?, 512)
bbox_head: Dense | input: (?, 32) | output: (?, 4)
softmax_head: Dense | input: (?, 512) | output: (?, 196)

This model requires single input and dual targets for both train and validation/test data. Image arrays are input as X. Bounding boxes, Labels are the dual targets and their dataframes are input as Y.

To create a multi-class object detector from scratch with Keras and TensorFlow, we have modified the network head of our architecture. The order of operations is below:

#1: Take VGG16 (pre-trained on ImageNet) and remove the fully-connected (FC) layer head. For the initial run, we can set the trainable to False. To increase the number of trainable parameters, the second half of the layers can be set as trainable as shown below.



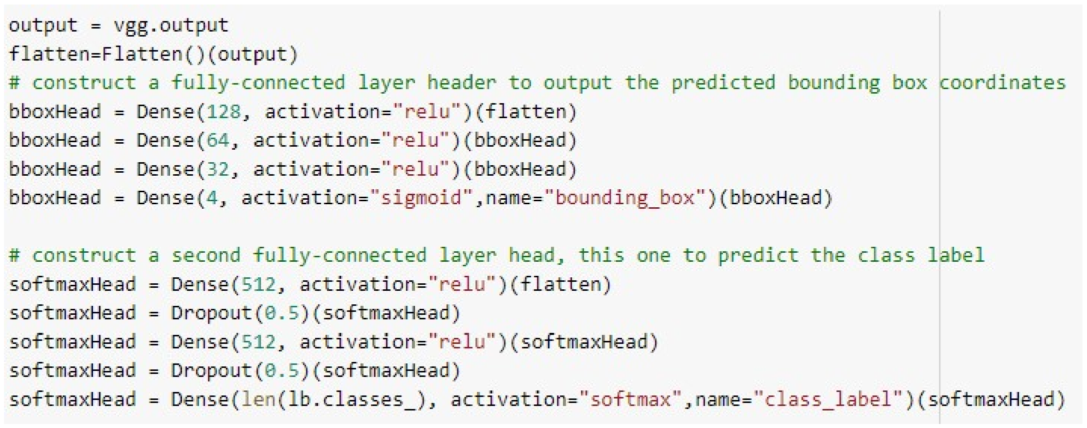
```
vgg = VGG16(weights="imagenet", include_top=False, input_tensor=Input(shape=(224, 224, 3)))
vgg.trainable = True
set_trainable = False

for layer in vgg.layers:
    if layer.name in ['block5_conv1', 'block4_conv1']:
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

#2: Construct a new FC layer head with two branches:

Branch #1: A series of FC layers that end with a layer with (1) four neurons, corresponding to the top-left and bottom-right (x, y)-coordinates of the predicted bounding box and (2) a sigmoid activation function, such that the output of each four neurons lies in the range [0, 1]. This branch is responsible for bounding box predictions.

Branch #2: Another series of FC layers, but this one with a softmax classifier at the end. This branch is in charge of making class label predictions.

#3: Flatten the VGG output and place it on top of the new FC layer head (with the two branches).

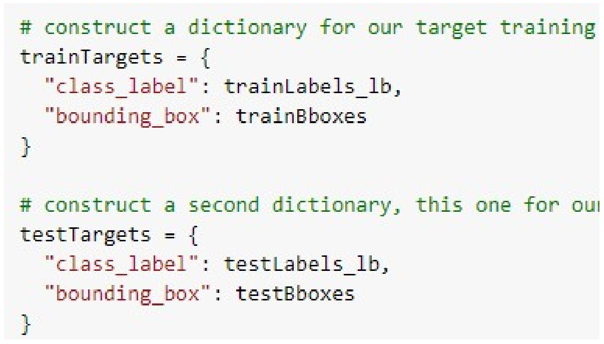


```
output = vgg.output
flatten=Flatten()(output)
# construct a fully-connected layer header to output the predicted bounding box coordinates
bboxHead = Dense(128, activation="relu")(flatten)
bboxHead = Dense(64, activation="relu")(bboxHead)
bboxHead = Dense(32, activation="relu")(bboxHead)
bboxHead = Dense(4, activation="sigmoid",name="bounding_box")(bboxHead)

# construct a second fully-connected layer head, this one to predict the class label
softmaxHead = Dense(512, activation="relu")(flatten)
softmaxHead = Dropout(0.5)(softmaxHead)
softmaxHead = Dense(512, activation="relu")(softmaxHead)
softmaxHead = Dropout(0.5)(softmaxHead)
softmaxHead = Dense(len(lb.classes_), activation="softmax",name="class_label")(softmaxHead)
```

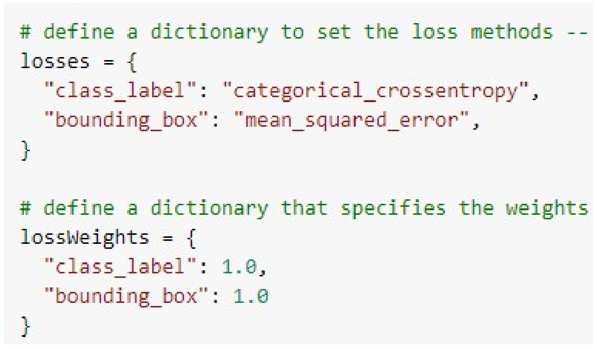
#4: Fine-tune the entire network for end-to-end object detection

This is a Modified Convolutional Neural Network trained/fine-tuned on the car dataset for object detection and classification.

This model has dual heads, so data frames are created for the dual targets, dual losses, and their weights, for both train and test images.



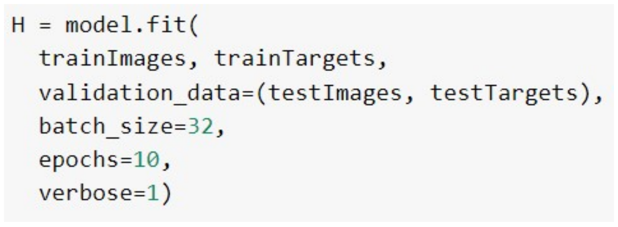
```
# construct a dictionary for our target training
trainTargets = {
  "class_label": trainLabels_lb,
  "bounding_box": trainBboxes
}

# construct a second dictionary, this one for ou
testTargets = {
  "class_label": testLabels_lb,
  "bounding_box": testBboxes
}
```



```
# define a dictionary to set the loss methods --
losses = {
  "class_label": "categorical_crossentropy",
  "bounding_box": "mean_squared_error",
}

# define a dictionary that specifies the weights
lossWeights = {
  "class_label": 1.0,
  "bounding_box": 1.0
}
```

3.1. Model Training

For the model training, we initially used the original train images without augmentation and test images with the batch size of 32, epochs 10 without callbacks. To improve the model performance, we will be use augmented images, increase the number of epochs and use the callbacks with reduced learning rates and early stopping.



```
H = model.fit(
  trainImages, trainTargets,
  validation_data=(testImages, testTargets),
  batch_size=32,
  epochs=10,
  verbose=1)
```

```
Epoch 10/10
Train: bounding_box_accuracy: 0.2622, class_label_accuracy: 0.0389
Val: val_bounding_box_accuracy: 0.2736, val_class_label_accuracy: 0.0528
```

At the end of the 10th epoch,

Bounding box validation accuracy is 28%

Label validation accuracy is <6%

3.2. Model Saving

Finally, the model weights are saved in h5 format, model architecture is saved as JSON. Model history and labels are saved as pickle files

```
# Serialize the model structure to json
model_json = model.to_json()
with open( BASE_OUTPUT + "model.json","w") as json_file:
    json_file.write(model_json)
```

```
# Serialize the model weights to HDF5
print("[INFO] saving object detector model...")
model.save(BASE_OUTPUT +"detector.h5", save_format='h5')
```

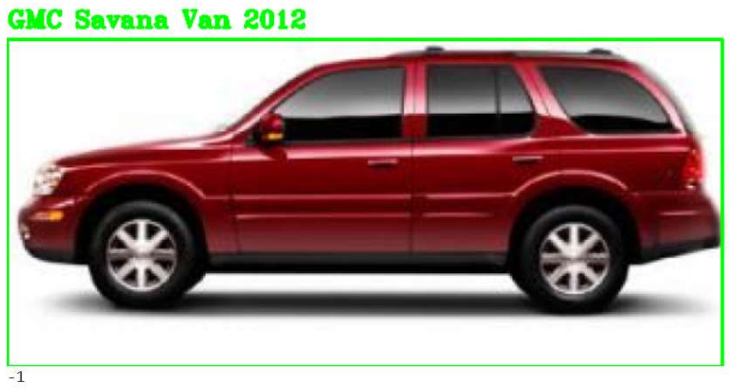
```
[INFO] saving object detector model...
```

```
#saving model history
with open(BASE_OUTPUT +'trainHistory', 'wb') as file_pi:
    pickle.dump(H.history, file_pi)

#Reading modelhistory
#H = pickle.load(open(BASE_OUTPUT +'trainHistory', "rb"))
```

```
#serialize the label binarizer to disk
f=open(BASE_OUTPUT + "lb.pickle", "wb")
f.write(pickle.dumps(lb))
f.close()
```

3.3. Model Prediction

The picture below shows the bounding box and class label predictions plotted on the image. The model could predict the boundary box very accurately but the class label prediction is not matching with the original car label. Class label prediction accuracy need to be improved further.

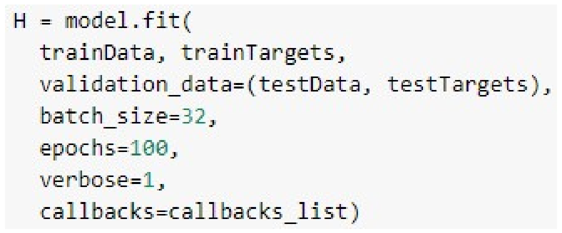


Original Car Label: Buick Rainier SUV 2007

Model performance can be improved by using image data augmentation, hyperparameter tuning, more number of epochs, training the last layers of the base model, adding more dense layers..etc. These require higher computation capabilities but a balance between the accuracy and the cost can be achieved. The model improvement studies are covered in the next steps.

4. Model Improvement

For the model improvement and training, increased the number of epochs to 100, used callbacks with reduced learning rate and early stopping. Changed the optimizer from adam (lr= 1e-4) to RMSprop(lr=1e-5).



```
H = model.fit(
  trainData, trainTargets,
  validation_data=(testData, testTargets),
  batch_size=32,
  epochs=100,
  verbose=1,
  callbacks=callbacks_list)
```

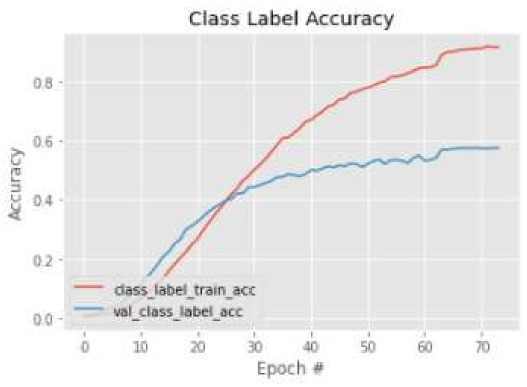
Two different models were tried with the same CNN architecture with dual heads:

Mobile Net and VGG16

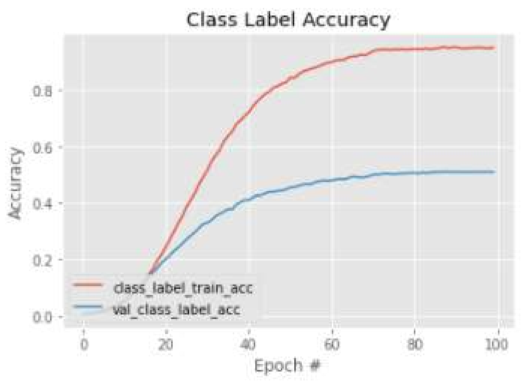
4.1. Model Evaluation

For the model evaluation, we looked at the loss and accuracy. Here are the comparisons of the two models:

VGG16 model:



Mobile Net model:

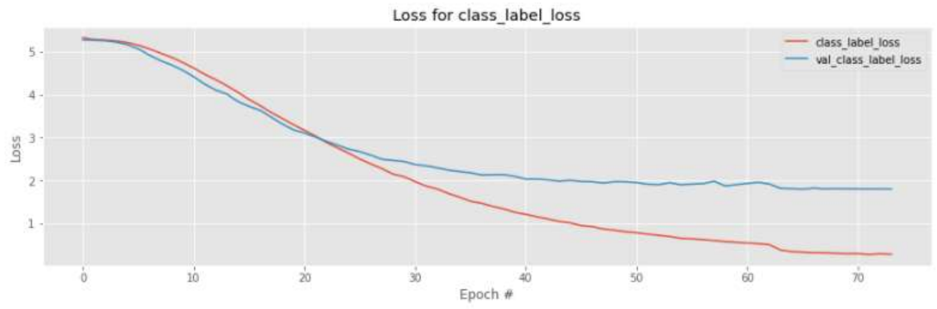


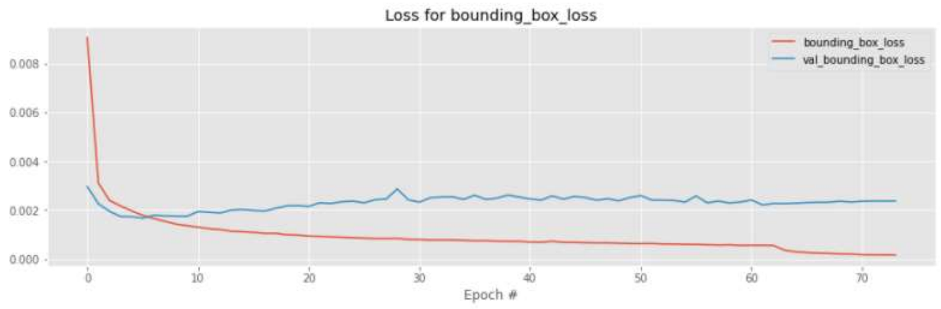
Model Accuracy (%)	Bounding Box (Train/Test)	Class Label (Train/Test)
VGG (74 epochs with early stop)	90 / 84.3	91.6 / **57.5**
Mobile Net (100 epochs)	91.4 / 86.6	94.9 / 50.8

Both the models are showing over 80% accuracy for the Regression and over 50% accuracy for the Classification. VGG16 model is giving better class validation accuracy compared to the Mobile Net as shown in the figure. Bounding box accuracies are not too far between train and test for both the models but the class label accuracies are under predicted on validation set with 30% less accuracy.

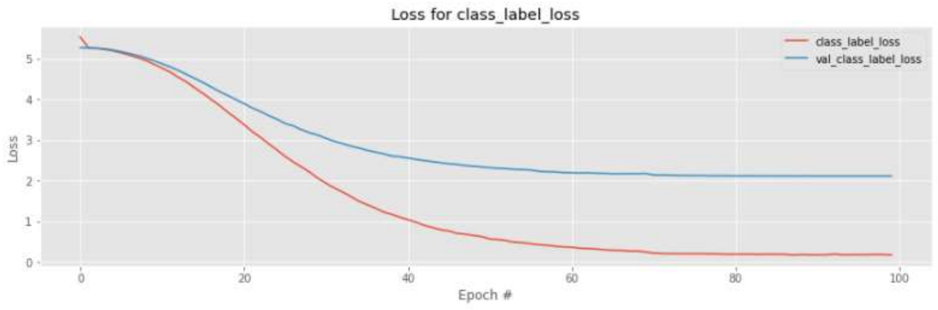
Loss Comparison:

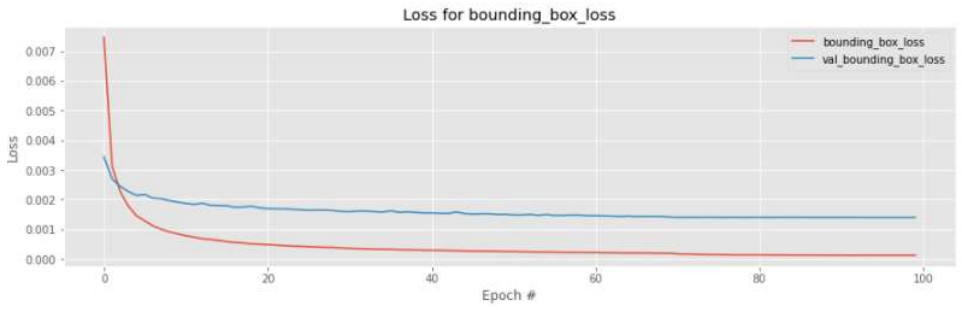
VGG16:





Mobile Net:





The losses are almost stabilized after 70 epochs in the both models. VGG16 model was stopped earlier after 70 epochs but Mobile Net model could run completely up to 100 epochs.

5. Comparison to Benchmark

Our benchmark for this object detection model is to predict the bounding box around the car and predict the correct label which is make, model and year of the car, for the input test image.

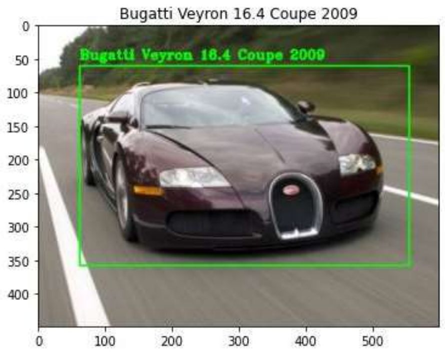
There are several competitions on Kaggle for the object detection where the objects are detected, localized and labeled for very generic categories. For the Stanford car dataset, there are models which showed 86% class label accuracy in just 4 epochs but those were not trained for predicting both bounding boxes and labels. The complexity in our task was to predict both in a single model without referring to any available models in the competitions or online. Though our model could produce satisfactory results, there is always an opportunity for the improvement.

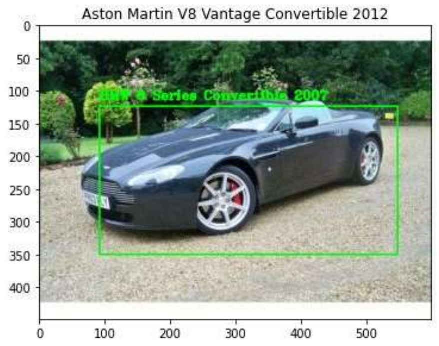
With the given time and availability of resources we could train our model to achieve above 80% validation accuracy for the bounding boxes but only less than 60% accuracy on the class label prediction. That means, we are able to detect the car with the correct bounding boxes and fulfill our first requirement. And, we are able to predict the correct class label for almost 60% of the test images, which is not really bad after looking at the complexity of the various car constructions. Car styles vary from a sleek sports car to a large pickup truck under so many brands.

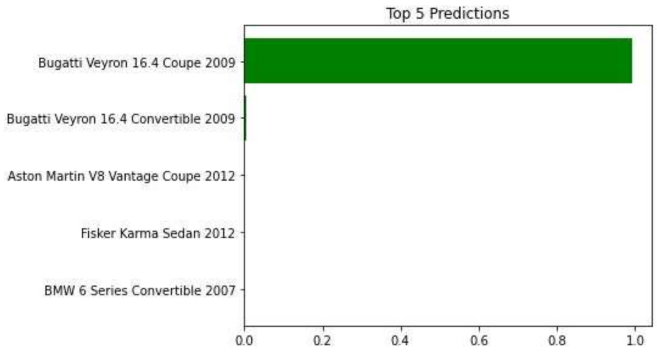
However, the model could predict the correct label for more than 90% of the test car images at least in top 5 predictions. Overall, the prediction for the bounding boxes and the class labels are satisfactory compared to the benchmark. The comparison of the predictions between VGG16 and Mobile net models are shown in the next step.

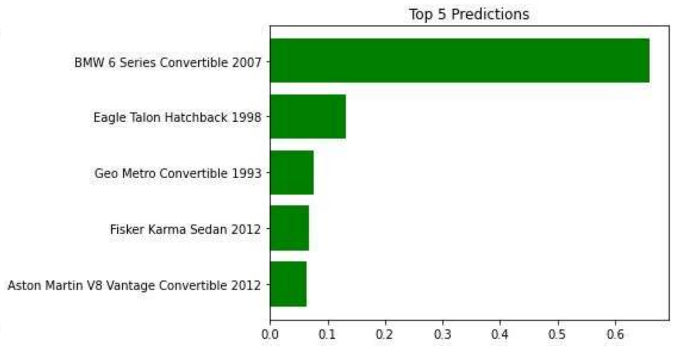
6. Visualization of Model Predictions

VGG model could predict almost 60% of car labels correctly and it could predict the correct label at least in top5 predictions for more than 30% in the remaining cars.

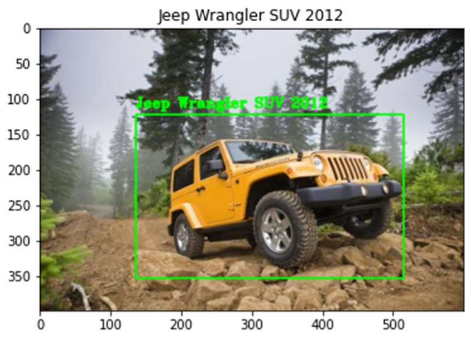

Bugatti Veyron 16.4 Coupe 2009
Bugatti Veyron 16.4 Coupe 2009

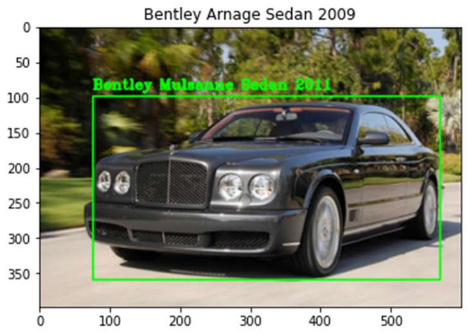

Aston Martin V8 Vantage Convertible 2012
BMW 6 Series Convertible 2007

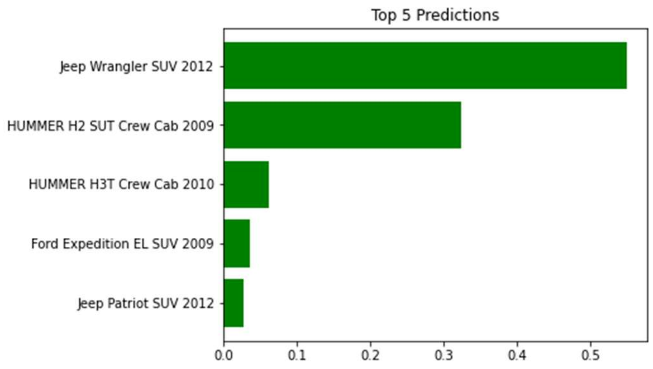


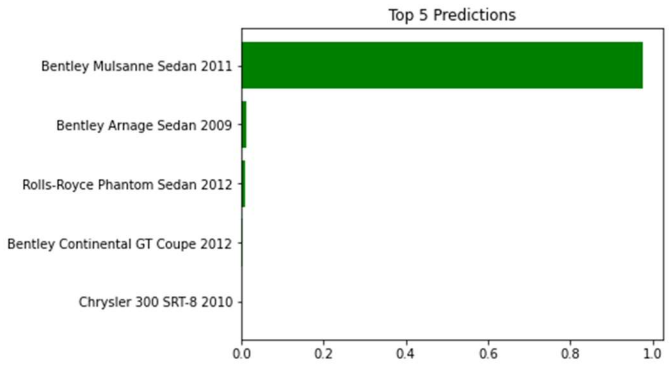


Mobile Net model also giving similar results compared to VGG16 but with less accuracy.









6.1. GUI Creation and Model Deployment

Model Deployment:

In a typical machine learning and deep learning project, we usually start by defining the problem statement followed by data collection and preparation, understanding of the data, and model building.

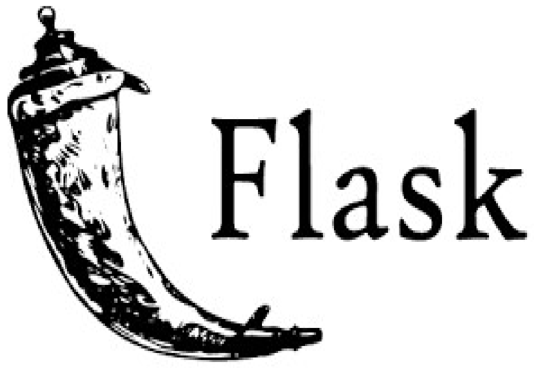
But, in the end, we want our model to be available for the end-users so that they can make use of it. Model Deployment is one of the last stages of any machine learning project and can be a little tricky. How do you get your machine learning model to your client/stakeholder? What are the different things you need to take care of when putting your model into production? And how can you even begin to deploy a model?

Here comes the role of Flask. And, the following text will walk through all the steps in brief as our agenda is to deploy the model and use it on our localhost and it can be further extended to deploy on the web as well using Heroku platform.

About Flask:

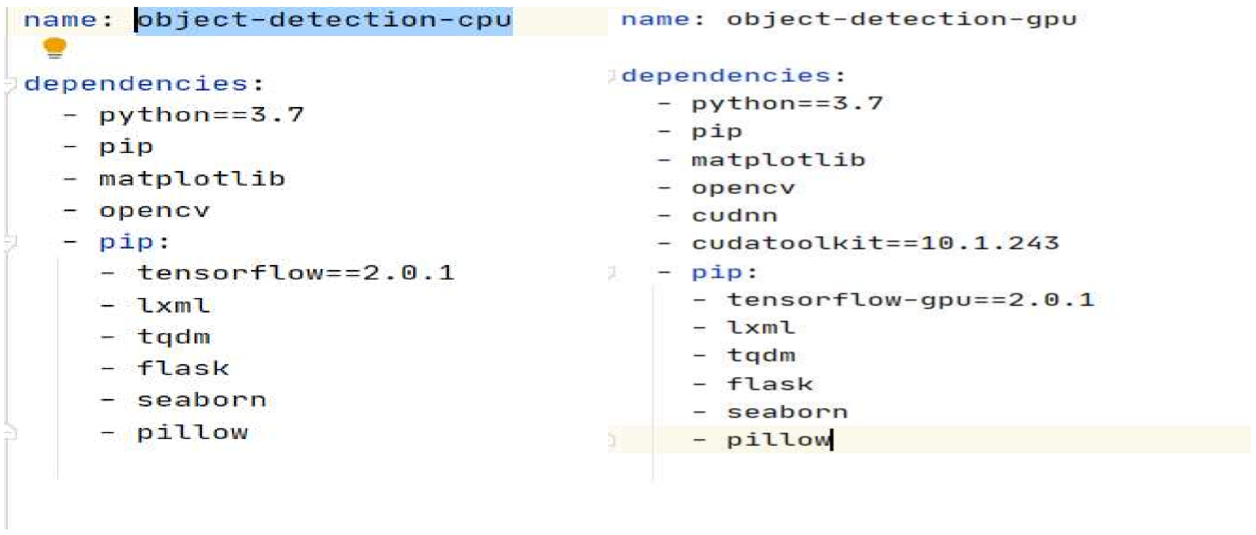
Flask is a web application framework written in Python. It has multiple modules that make it easier for a web developer to write applications without having to worry about the details like protocol management, thread management, etc.

Flask gives us a variety of choices for developing web applications and it gives us the necessary tools and libraries that allow us to build a web application.



Installing Libraries:

If you have an anaconda installed, then you need to follow this to install all the required libraries in an environment. This will create a new environment. The environment name can be changed in the .yml file.



```
name: object-detection-cpu

dependencies:
  - python==3.7
  - pip
  - matplotlib
  - opencv
  - pip:
    - tensorflow==2.0.1
    - lxml
    - tqdm
    - flask
    - seaborn
    - pillow
```

```
name: object-detection-gpu

dependencies:
  - python==3.7
  - pip
  - matplotlib
  - opencv
  - cudnn
  - cudatoolkit==10.1.243
  - pip:
    - tensorflow-gpu==2.0.1
    - lxml
    - tqdm
    - flask
    - seaborn
    - pillow
```

The new environment with respective dependencies will get created. Now, to use the newly created environment you need to activate it.

Activating Environment

Command (CPU): *conda object-detection-cpu*

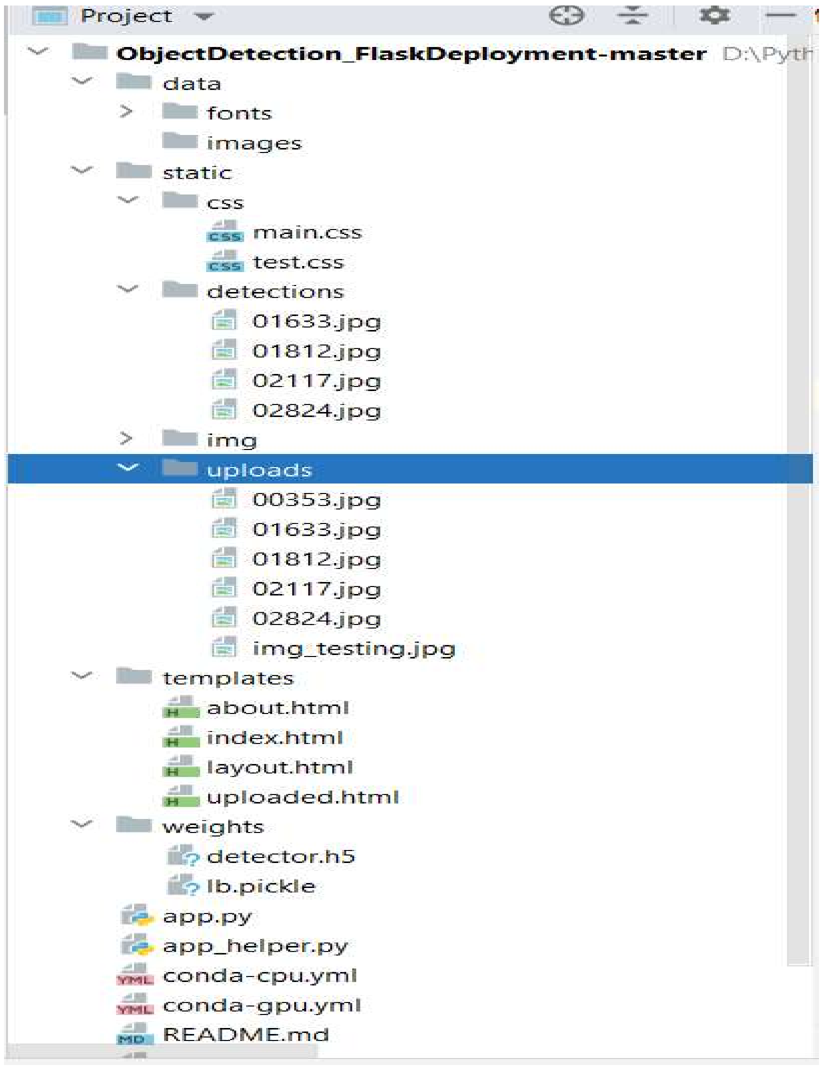
Command (GPU): *conda object-detection-gpu*

Workflow:

Webpage template: - Here, we will design a user interface where the user can submit the query in the form of an image.

GUI: - Backend code will use the saved model weights to predict class & bounding boxes and send the results back to the webpage.

GUI Folder structure:



```
Project
ObjectDetection_FlaskDeployment-master D:\Pyth
    data
        fonts
        images
    static
        css
            main.css
            test.css
        detections
            01633.jpg
            01812.jpg
            02117.jpg
            02824.jpg
        img
        uploads
            00353.jpg
            01633.jpg
            01812.jpg
            02117.jpg
            02824.jpg
            img_testing.jpg
    templates
        about.html
        index.html
        layout.html
        uploaded.html
    weights
        detector.h5
        lb.pickle
    app.py
    app_helper.py
    conda-cpu.yml
    conda-gpu.yml
    README.md
```

Explanation of input files:

“ObjectDetectionFlaskDeployment” is the master directory and following are the sub directories.

“/data” contains the frontend files such as fonts, pictures that are been used on the webpage.

“/static/css” contains CSS related files for the graphics required on the webpage.

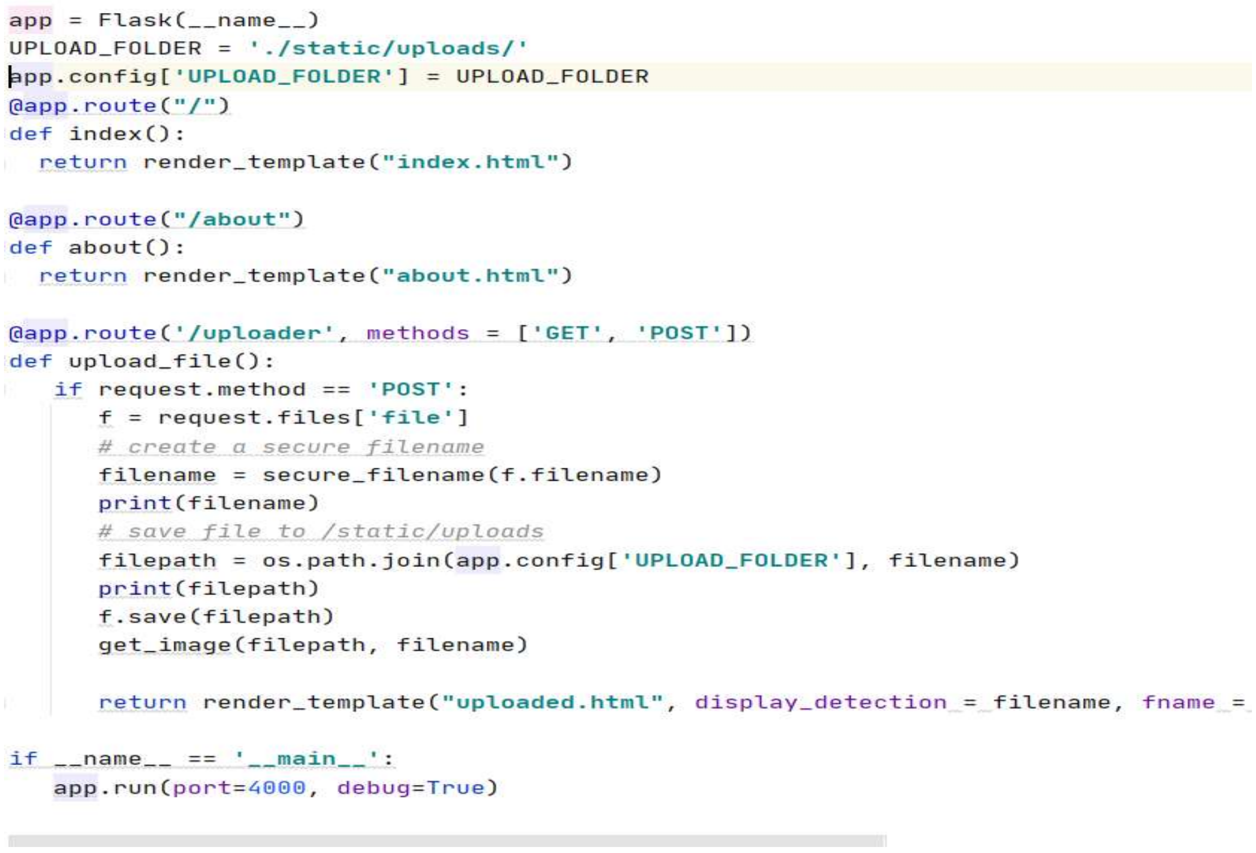
“/static/upload” contains images that the user to upload during the prediction through the web.

“/static/detection” contains images with predicted class labels and bounding boxes and these images are displayed on the web page.

“/templates” folder contains HTML files that are used in designing the user web interface.

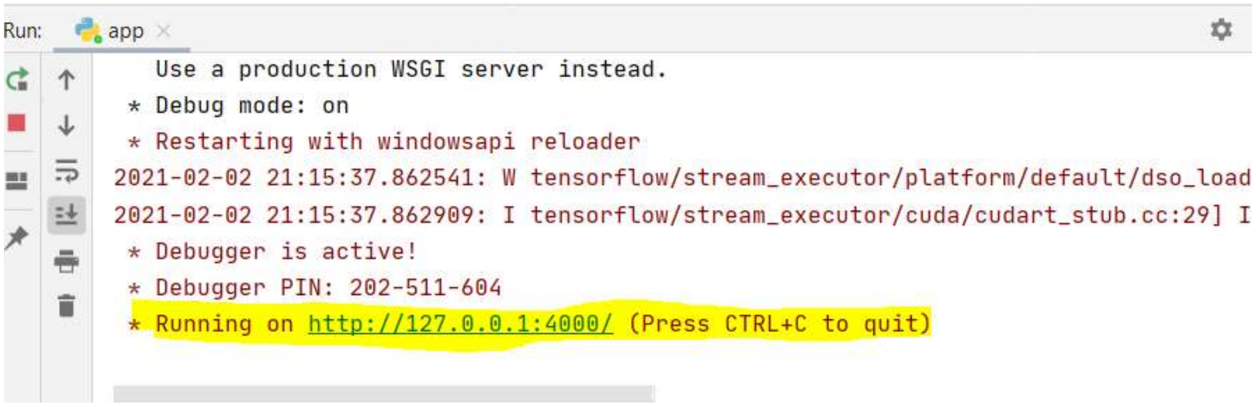
“/weights” contains model weight file -detector.h5 and labels binary pickle file- lb.pickle.

“/app.py” is the starting point where our flask starts servers running. This file contains a run method where it starts the whole application. Once this app starts, it invokes methods: GET & POST, from there it calls the function: upload_files.



```python
app = Flask(__name__)
UPLOAD_FOLDER = './static/uploads/'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
@app.route("/")
def index():
  return render_template("index.html")

@app.route("/about")
def about():
  return render_template("about.html")

@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
      f = request.files['file']
      # create a secure filename
      filename = secure_filename(f.filename)
      print(filename)
      # save file to /static/uploads
      filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
      print(filepath)
      f.save(filepath)
      get_image(filepath, filename)

      return render_template("uploaded.html", display_detection = filename, fname =

if __name__ == '__main__':
   app.run(port=4000, debug=True)
```

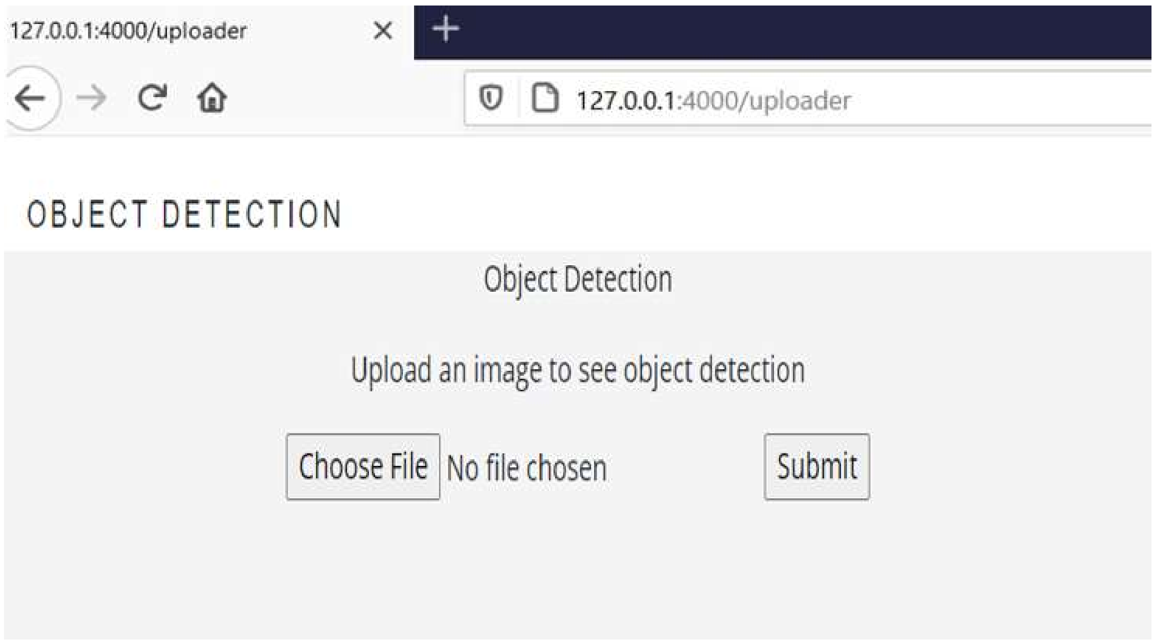
Once the above application up and running, it will provide one localhost URL as shown in the picture below.



Run: app

```
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with windowsapi reloader
2021-02-02 21:15:37.862541: W tensorflow/stream_executor/platform/default/dso_load
2021-02-02 21:15:37.862909: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] I
 * Debugger is active!
 * Debugger PIN: 202-511-604
 * Running on http://127.0.0.1:4000/ (Press CTRL+C to quit)
```

Using the above URL, one can browse and upload the car image for the predictions. URL will redirect to the below page.



Once the car image is browsed using *choose File* option and hit *submit* then internally from app.py will call app_helper.py file.

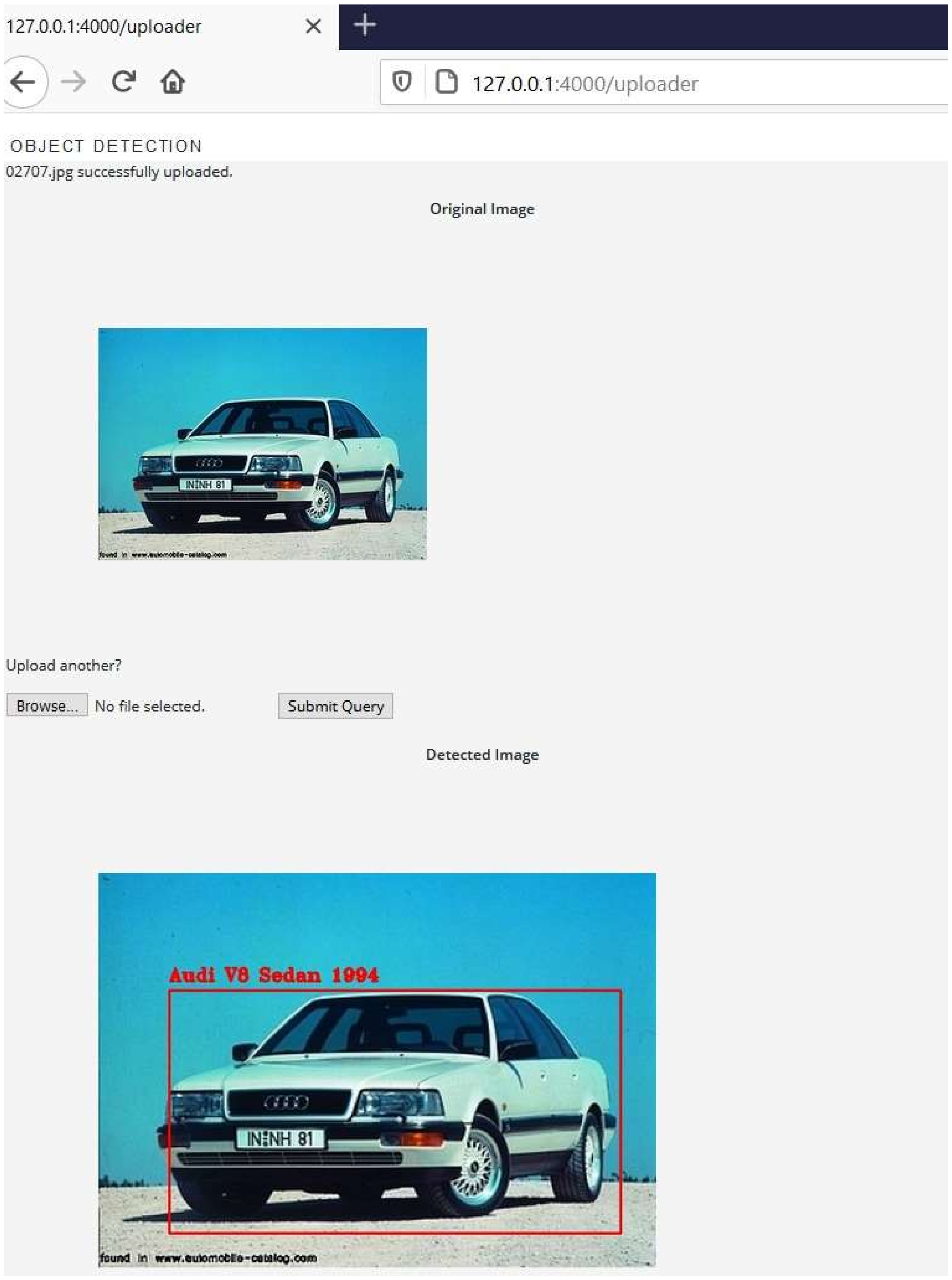
“app_helper.py” file contains a helper function that helps in predicting the image using model weights (detector.h5) and label file(lb.pickle)

Once the file is uploaded into the above GUI and hit submit, internally it uses both weight and pickle files to find a class label (car name), bounding box coordinates for object detection.

Using the above output: class labels & bonding box coordinates, using cv2 functions, we are drawing a bounding box and putting the label on top of the image.

6.2. GUI using HTML web page

After submitting the image, the GUI plots the input image and with the bounding box and car class label.



7. Implications

This Car detection model could help in the automotive and surveillance domains to trace the exact car when spotted on the public or private places.

In the automotive domain, autonomous vehicles are expected to dramatically redefine the future of transportation. However, there are still significant engineering challenges to be solved before one can fully realize the benefits of self-driving cars. The challenge for self-driving car system is to identify 3D objects, which an important step before detecting their movement. One such challenge is building models that reliably predict the movement of traffic agents, such as cars, cyclists, and pedestrians. This car detection model should be able to detect the dimensions of the cars around and maintain the distance to avoid crash.

In the surveillance domain, the most common and well-known application from the category of traffic surveillance and law enforcement is license plate recognition. Due to growing demand, other categories of vehicle classification have also been added recently: Make, model and color of the car. Other application is to recognize the cars parked on roadside, which cause unnecessary blockage and create safety issues for other road users. The surveillance system requires computer vision-based technology to facilitate and enhance the accuracy and effectiveness of detection and recognition of vehicles. This is another application where our car detection model could help in solving the problem by detecting and predicting the correct car make, model and year. And it can be further developed to extract the text on the number plates as well.

An increasing number of vehicles and insufficient data transparency make the situation worse, and this issue is becoming critical in cities with high transportation density. Hence, there are increasing numbers of applications for computer vision car detection, However, current systems have limitations, such as being affected by vehicle speeds. To solve this problem, we need to record the videos, process the clippings and generate images, process them, train the models and make predictions. To perform all these tasks in no time, it requires very highspeed computation resources.

8. Limitations

In this project, there were only few images available for each class of the cars but there are 196 such classes which makes it complex to train and predict the right class. With the limited data availability and resources, we could achieve only less than 60% accuracy in predicting the right class of the car. In the real world, there are several thousand models available and it's really a challenge for our model to train. For a few thousand images, our model was taking lot of time on GPU and it will take huge amount of time to train for the cars in the real world.

Major challenges to implement our model in the real world as follow.

#1. Dual target: object classification and localization: The first major complication of object detection is its added goal: not only do we want to classify image objects but also to determine the objects' positions, generally referred to as the object localization task.

#2. Speed for real-time detection: Object detection algorithms need to not only accurately classify and localize important objects, but they also need to be incredibly fast at prediction time to meet the real-time demands of video processing.

#3. Multiple spatial scales and aspect ratios: For many applications of object detection, items of interest may appear in a wide range of sizes and aspect ratios.

#4. Limited data: The limited amount of annotated data currently available for object detection proves to be another substantial hurdle. Object detection datasets typically contain ground truth examples for about a dozen to a hundred classes of objects, while image classification datasets can include upwards of 100,000 classes. Gathering ground truth labels along with accurate bounding boxes for object detection, however, remains incredibly tedious work.

#5. Class imbalance: It proves to be an issue for most classification problems, and object detection is no exception. Consider a typical photograph. More likely than not, the photograph contains a few main objects and the remainder of the image is filled with background.

To address these issues, we must use a multi-task loss function to penalize both misclassifications and localization errors. We need to boost the speed by using the faster algorithms like YOLO with test time of 155 frames per second (fps) which is faster than of R-CNN with 0.02 fps. We need to use several techniques to ensure detection algorithms capture objects at multiple scales and views.

9. Closing Reflections

Computer vision is the ability of artificially intelligent systems to “see” like humans, it has been a subject of increasing interest and rigorous research for decades now. As a way of emulating the human visual system, the research in the field of computer vision purports to develop machines that can automate tasks that require visual cognition.

However, the process of deciphering images, due to the significantly greater amount of multi-dimensional data that needs analysis, is much more complex than understanding other forms of binary information. This makes developing AI systems that can recognize visual data more complicated.

Deep learning which is also based on machine learning require lots of mathematical and deep learning frameworks understanding. By using dependencies such as TensorFlow, Keras, OpenCV etc., we can detect each and every object in the image by the area object in a highlighted rectangular box and identify each and every object and assign its tag to the object.

This also includes the training of different deep learning models, the evaluation of the performance by the measure of accuracy of each model, selection of the best model suited for the application and the problem that we want to solve.

This project helped us as a team to strengthen our basics and understanding the significant aspects of the implementation of a computer vision project. Understanding how things work internally is crucial in computer vision because it helps us figure out how exactly the computer analyzes and processes the data as well as appreciate the beauty behind its methodologies.

Through this Computer vision Car detection project, we acquired a lot of knowledge by referring to various blogs and competitions. This is a very interesting subject for us to learn further and gain confidence to implement our knowledge in the real-world applications like automotive surveillance and especially in the autonomous vehicle development.

10. References

Data Source: https://www.kaggle.com/jutrera/stanford-car-dataset-by-classes-folder

Image Augmentation with Bounding boxes: https://medium.com/@a.karazhay/guide-augment-images-and-multiple-bounding-boxes-for-deep-learning-in-4-steps-with-the-notebook -9b263e414dac

Training for both Regression & Classification: https://www.pyimagesearch.com/2020/10/12/multi-class-object-detection-and-bounding-box-regression-with-keras-tensorflow-and-deep-learning/

Plotting top 5 predictions: https://github.com/wengsengh/Car-Models-Classifier/blob/master/car_models_classifier.ipynb

Real world challenges: https://towardsdatascience.com/5-significant-object-detection-challenges-and-solutions-924cb09de9dd