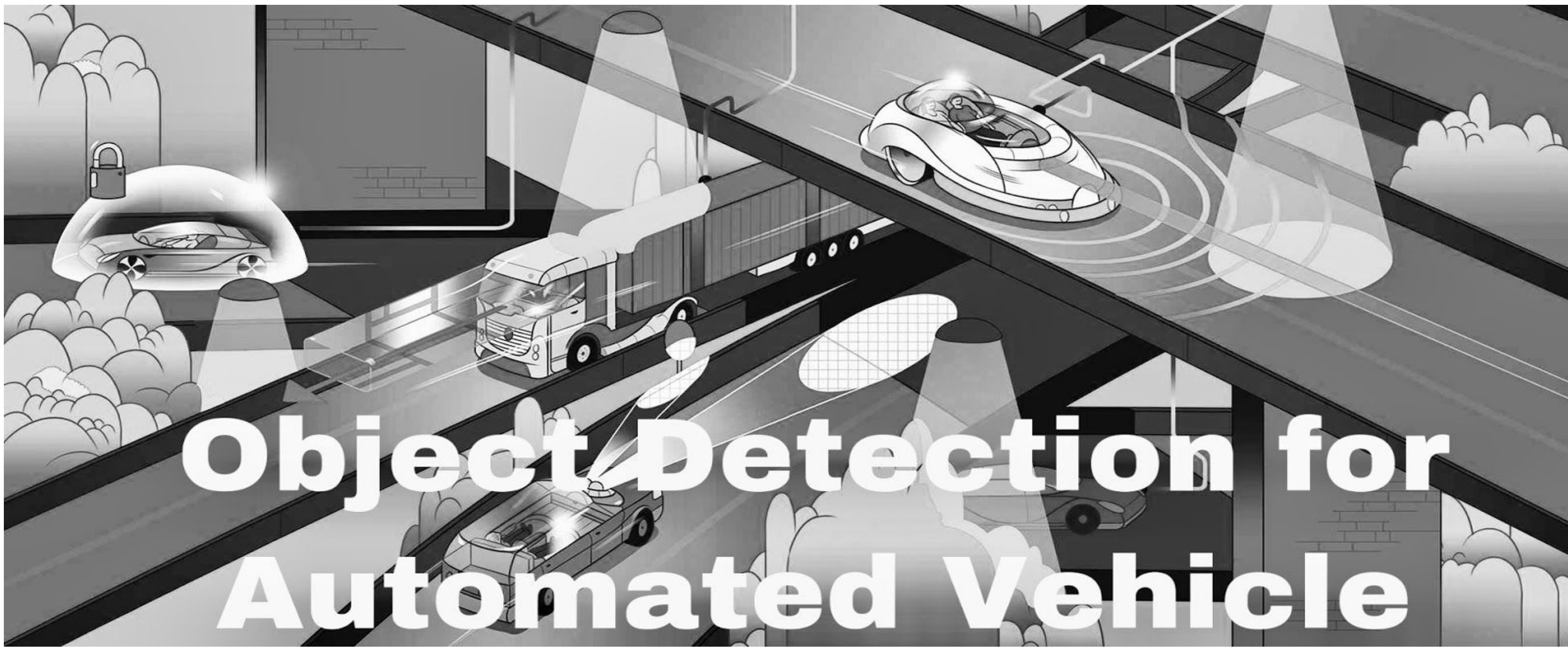


Table of Contents

1. Introduction

Automobiles rely on their drivers less than ever before and fully driverless cars represent a potentially impactful change in urban transportation. Changes in automobile technology have enabled vehicles to perform safety-critical functions in more circumstances, ranging from cruise control and lane-keeping. These changes may lead to fully-autonomous vehicles (AVs) which require no human control over safety critical functions.
An automated vehicle requires an accurate perception of its surrounding environment to operate reliably. The perception system of such vehicles transforms sensory data into semantic information that enables autonomous driving. Object detection is a fundamental function of this perception system. This project aims to build and optimize algorithms based on a large-scale dataset and focus on hard problems in this perception system i.e. object detection. Object detection is a crucial task for autonomous driving. Many important fields in autonomous driving such as prediction, planning, and motion control generally require a faithful representation of the 3D space around the vehicle.
Many automobile companies are currently working on manufacturing fully automated versions of cars. Automated vehicles offer exciting opportunities for transit providers and transit users and are expected to redefine transportation. This automation aims to demonstrate the future opportunity for a safe, green, accessible and convenient transit technology to support local travel needs. Hence, detecting the objects on the road successfully using this project can be an important step towards this development of automated vehicles.

2. Data Loading and Exploration

The data which will be used for this project work is fetched from this link. The bounding box dataset contains 11,754 frames. Each frame has an annotated bounding box label. All frames are grouped in 18 different scenes with each scene contained in its corresponding folder. Scene folder names are in the 'YYYYMMDD_hhmmss' format. They represent the date and time of the recording. Each scene is further divided into four folders as mentioned below. For this project, we will be using data from two folders namely 'camera' and 'label3D'.

- camera: input images and json info files
- lidar: input 3D point clouds
- label: annotated label images
- label3D: annotated 3D bounding boxes

Each of these folders are further divided depending on the camera from which the data was recorded. There are six cameras available in the vehicle, therefore, the following are the camera folders:

- cam_front_center
- cam_front_left
- cam_front_right
- cam_side_left
- cam_side_right
- cam_rear_center

Lastly, each of these folders contains the corresponding item for each frame. These are the filename formats for the item of a single frame:

- Input RGB image : YYMMDDDDhhmmss_camera_[frontcenter|frontleft|frontright|sideleft|sideright|rearcenter]_[ID].png
- 3D Bounding Box : YYMMDDDDhhmmss_label3D_[frontcenter|frontleft|frontright|sideleft|sideright|rearcenter]_[ID].json

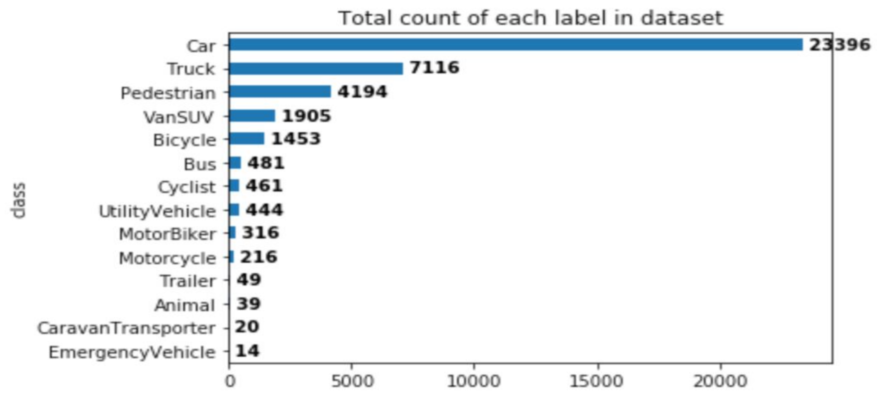
For example, a frame with ID 1617 from a scene recorded on 2018-08-07 14:50:28 from the front center camera would have the items in these locations:

- input RGB image : 20180807_145028/camera/cam_front_center/20180807145028_camera_frontcenter_000001617.png
- 3D Bounding Box: 20180807_145028/label3D/cam_front_center/20180807145028_label3D_frontcenter_000001617.json

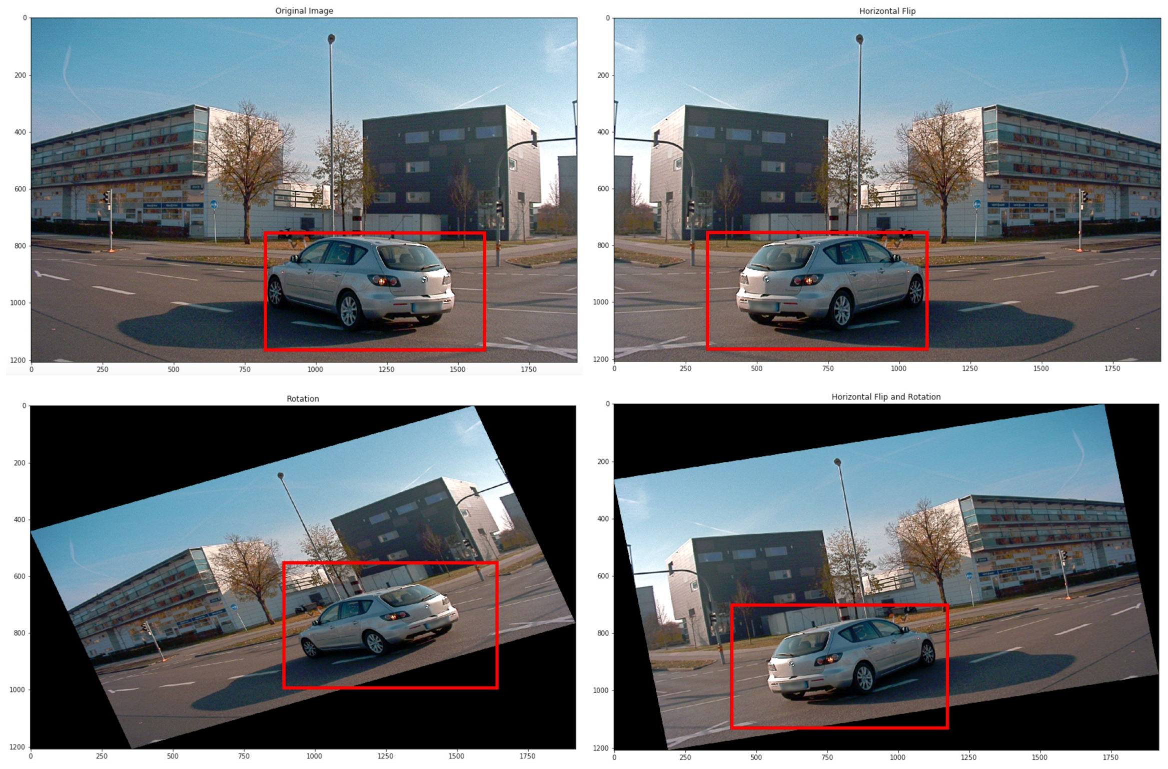
3. Data Conversion

We get the images from the dataset and these images consist of 14 different classes. The class label 'Car' has maximum occurrence in the whole dataset while the class label 'EmergencyVehicle' has least occurrence. The total count of other class labels and overall dataset exploration is done in the given IPython Notebook.

The json file in the dataset has the '2d_bbox' details which gives us the object annotations i.e. bounding boxes and it also has corresponding 'class' labels for each image. Before we start our model training, we will need a set of images to teach the network about the classes we want to recognize. So to begin with, we need to first split the data. The overall image data containing 11,754 frames is split into 8227 frames as train set, 1176 frames for test set and 2351 frames for validation set.



The above plot shows that there are not equal numbers of images for each class, hence we need to perform data augmentation. Data augmentation is a way to artificially expand the dataset and thinking of it as added noise to the dataset. Each time the neural network sees the same image, it's a bit different due to the stochastic data augmentation being applied to it. This difference can be seen as noise being added to the data sample each time, and this noise forces the neural network to learn generalised features instead of overfitting on the dataset. In the project, we will be performing, horizontal flip, rotation and sequence of both horizontal flip and rotation. Below, are the samples of the images after performing these data augmentation steps. Image 1 is the original image, image 2 is after performing horizontal flip. In image 3, rotation is performed and in image 4, both horizontal flip and rotation is performed.



In this project, we will be using TensorFlow as the Machine Learning framework. Hence, we need to convert the dataset into a TFRecord file for fine-tuning the model. TFRecord file format is a simple record-oriented binary format that TensorFlow applications use for training data. It is the default file format for TensorFlow. The TFRecord format consists of a set of shared files where each image is a serialized tf.Example message (or protobuf). Each tf.Example message contains the image as well as metadata such as label and bounding box information. The binary format of TFRecord files makes it easier to use, because for binary files we don't have to specify different directories for images and ground truth annotations. While storing data in a binary file, we have our data in one block of memory, compared to storing each image and annotation separately. Overall, by using binary files we make it easier to distribute and make the data better aligned for efficient reading.

To create a tf.Example message from the dataset, we follow the below procedure which is implemented in method build_example() written in the IPython Notebook.

- For each image, each value needs to be converted to a tf.train.Feature. And this tf.train.Feature message type can accept one of the following three types:
    - tf.train.BytesList
    - tf.train.FloatList
    - tf.train.Int64List
- Then create a map (dictionary) from the feature name string to the encoded feature value produced in the step 1.
- The map produced in step 2 is converted to a Features message.

Once, the tf.Example is created for the image dataset, then all proto messages are serialized to a binary-string using the .SerializeToString method in audi_convert method() in the notebook. We repeat the above steps for train, validation and test sets to create respective TFRecord files.

4. Model for object detection

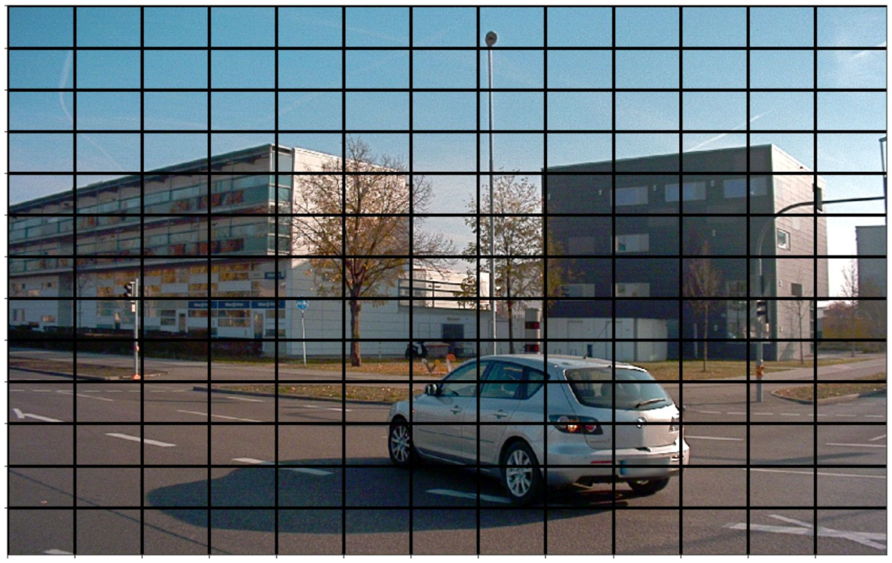
The model which we are planning to execute in this project is YOLOv3 model for object detection. You only look once (YOLO) is a state-of-the-art, real-time object detection system. It takes the entire image in a single instance and predicts the bounding box coordinates and class probabilities for these boxes. The approach involves a single deep convolutional neural network that splits the input into a grid of cells and each cell directly predicts a bounding box and object classification. The result is a large number of candidate bounding boxes that are consolidated into a final prediction by a post-processing step.

The biggest advantage of using YOLO is its superb speed. YOLO also understands generalized object representation. This is one of the best algorithms for object detection and has shown a comparatively similar performance to the R-CNN (Region-Convolutional Neural Network) algorithms. R-CNN uses regions to localize the objects within the image, i.e. it does not look at the entire image, only at the parts of the images which have a higher chance of containing an object. R-CNN extracts a bunch of regions from the given image using selective search, and then checks if any of these boxes contains an object. We need to first extract these regions, and for each region, CNN is used to extract specific features. Finally, these features are then used to detect objects. Unfortunately, R-CNN becomes rather slow due to these multiple steps involved in the process. While, YOLO runs significantly faster than other detection methods with comparable performance.

As of now, there are three main variations of the YOLO approach; they are YOLOv1, YOLOv2, and YOLOv3. The first version proposed the general architecture, whereas the second version refined the design and made use of predefined anchor boxes to improve bounding box proposal, and version three further refined the model architecture and training process. In this project, we will implement the YOLOv3 model algorithm. The code for the YOLOv3 model implementation using TensorFlow framework is present in this IPython Notebook.

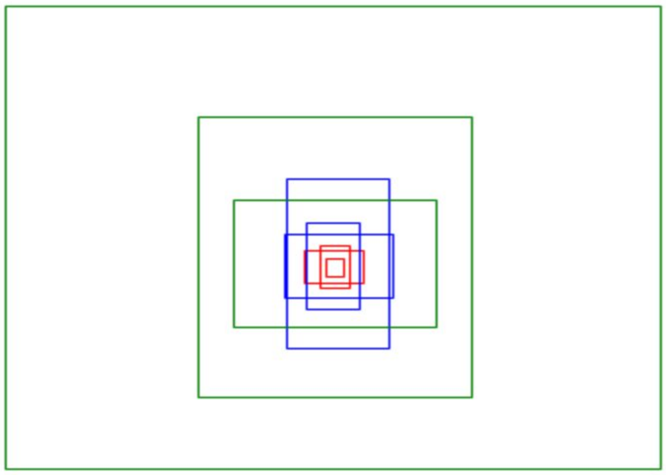
4.1. YOLOv3

As the name suggests, YOLO- You Only Look Once, it applies a single forward pass neural network to the whole image and predicts the bounding boxes and their class probabilities as well. This technique makes YOLOv3 a super-fast real-time object detection algorithm. YOLOv3 network divides an input image into [S x S] grid of cells and predicts bounding boxes as well as class probabilities for each grid. Each grid cell is responsible for predicting B bounding boxes and C class probabilities of objects whose centers fall inside the grid cell. Bounding boxes are the regions of interest (ROI) of the candidate objects. The B is associated with the number of using anchors. Each bounding box has (5 + C) attributes. The value of 5 is related to 5 bounding box attributes, those are center coordinates (b_x, b_y) and shape (b_h, b_w) of the bounding box, and one confidence score. The C is the number of classes which is 14 in this project. The confidence score reflects how confidence a box contains an object. The confidence score is in the range of 0–1. Since we have S x S grid of cells, after running one single forward pass of convolutional neural network to the whole image, YOLOv3 produces a 3D tensor with the shape of [S, S, B * (5 + C)]. We are training on the Audi dataset with C=14 and B=3. So, for the first prediction scale, after a single forward pass of CNN, the YOLOv3 outputs a tensor with the shape of [(13, 13, 3 * (5 + 14)].

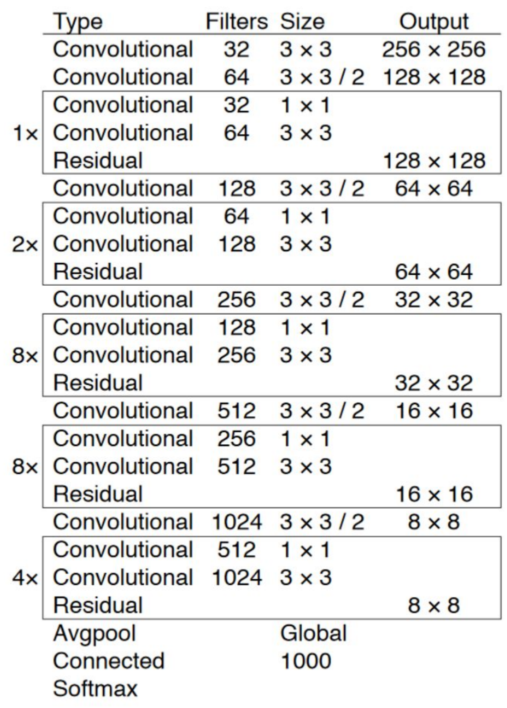


The code for the YOLOv3 model is implemented in the given IPython Notebook.

- **Anchor Boxes:** Basically, one grid cell can detect only one object whose midpoint of the object falls inside the cell. But a grid cell can also contain more than one midpoint of the objects, that means there can be multiple objects overlapping. In order to overcome this condition, YOLOv3 uses 3 different anchor boxes for every detection scale. The anchor boxes are a set of predefined bounding boxes of a certain height and width that are used to capture the scale and different aspect ratio of specific object classes that we want to detect. While there are 3 predictions across scale, so the total anchor boxes are 9, they are: (10×13), (16×30), (33×23) for the first scale, (30×61), (62×45), (59×119) for the second scale, and (116×90), (156×198), (373×326) for the third scale. These anchor boxes are shown below.



- **YOLOv3 Architecture:** YOLOv3 has 53 convolutional layers called Darknet-53 as shown below.



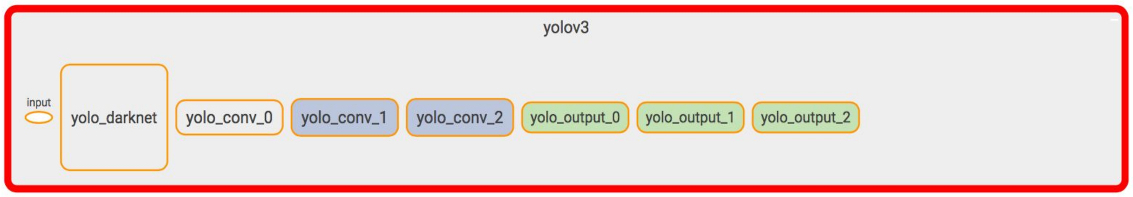
	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

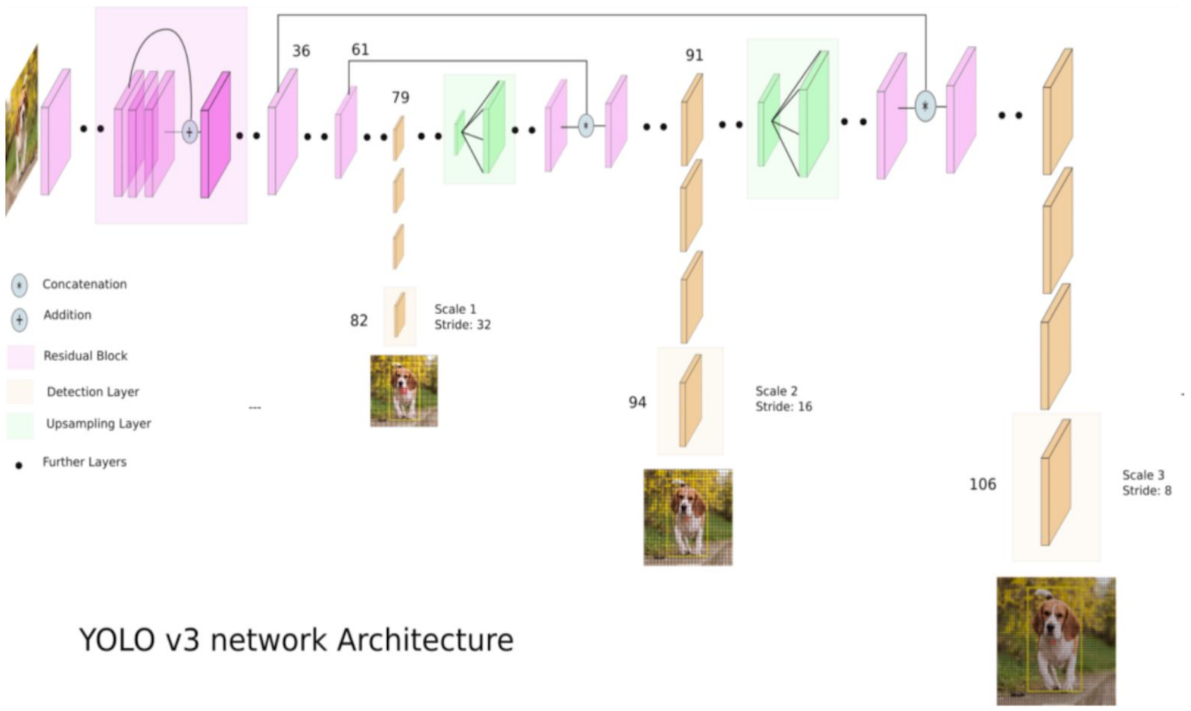
Consider layers in each rectangle as a residual block. The whole network is a chain of multiple blocks with some strides 2 Convolutional layers in between to reduce dimension. Inside the block, there is a structure of (1x1 followed by 3x3) plus a skip connection. Almost every convolutional layer in YOLOv3 has batch normalization after it.

Batch normalization helps the model train faster and reduces variance between units (and total variance as well). It is a preprocessing step of features extracted from previous layers, before feeding it to the next layers of the network. We normalize the input layer by adjusting and scaling the activations. The convolutional layer followed by a batch normalization layer uses a Leaky ReLU activation layer, otherwise, it uses the linear activation by default. ReLU(Rectified Linear Unit) is an Activation Function used in the Neural Network. Leaky ReLU is an advanced version of ReLU. Suppose if, for whatever reason, the output of a ReLU is consistently 0 (for example, if the ReLU has a large negative bias), then the gradient through it will consistently be 0. The error signal back propagated from later layers gets multiplied by this 0, so no error signal ever passes to earlier layers, the ReLU has died. Thus Leaky ReLU is used.

If the goal is to do multi-class classification as ImageNet does, an average pooling and a 1000 ways fully connected layers plus softmax activation will be added. However, in the case of object detection, we will not include this classification head. Instead, we are going to append a "detection" head to this feature extractor. And since YOLOv3 is designed to be a multi-scaled detector, we also need features from multiple scales. Therefore, features from the last three residual blocks are all used in the later detection.

The YOLOv3 makes detection in 3 different scales in order to accommodate different objects size by using strides of 32, 16 and 8. This means, when we feed an input image of size 416 x 416, YOLOv3 will make detection on the scale of 13 x 13, 26 x 26, and 52 x 52.





YOLO v3 network Architecture

For the first scale, YOLOv3 down samples the input image into 13 x 13 and makes a prediction at the 82nd layer. The 1st detection scale yields a 3D tensor of size 13 x 13 x 57. After that, YOLOv3 takes the feature map from layer 79 and applies one convolutional layer before upsampling it by a factor of 2 to have a size of 26 x 26. This upsampled feature map is then concatenated with the feature map from layer 61. The concatenated feature map is then subjected to a few more convolutional layers until the 2nd detection scale is performed at layer 94. The second prediction scale produces a 3D tensor of size 26 x 26 x 57. The same design is again performed one more time to predict the 3rd scale. The feature map from layer 91 is added one convolutional layer and is then concatenated with a feature map from layer 36. The final prediction layer is done at layer 106 yielding a 3D tensor of size 52 x 52 x 57.

- **Bounding box prediction:** For each bounding box, YOLOv3 predicts 4 coordinates, t_x, t_y, t_w, t_h. The t_x and t_y are the bounding box's center coordinate relative to the grid cell whose center falls inside, and the t_w and t_h are the bounding box's shape, width and height, respectively. The final output of the bounding box predictions need to be refined based on this formula where p_w and p_h are the anchor's width and height, respectively.

$$b_x = sigmoid(t_x) + C_x$$
$$b_y = sigmoid(t_y) + C_y$$
$$b_w = exp(t_w) * p_w$$
$$b_h = exp(t_h) * p_h$$

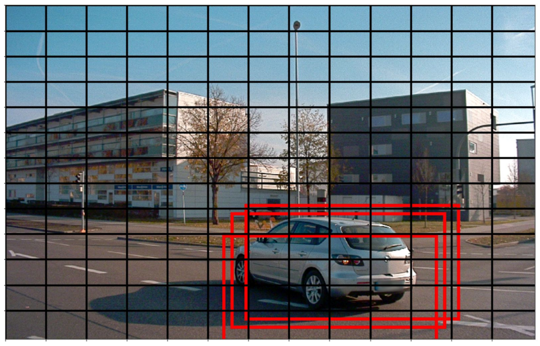
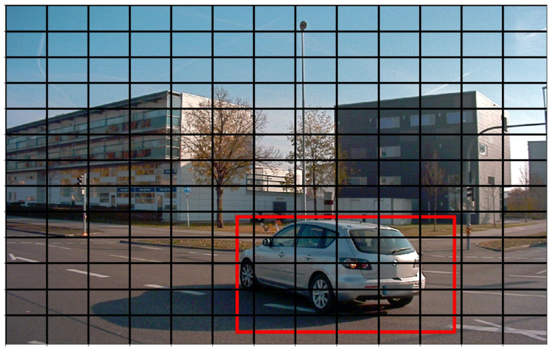
  The YOLOv3's algorithm returns bounding boxes in the form of (b_x, b_y, b_w, b_h). The b_x and b_y are the center coordinates of the boxes and b_w and b_h are the box shape (width and height). To draw boxes, we need the top-left coordinate (x1, y1) and the box shape (width and height). To get this we can convert them using this simple relation:

$$x_1 = b_x - (b_w/2)$$
$$y_1 = b_y - (b_h/2)$$

- **Non-Maximum Suppression:** After a single forward pass, YOLOv3 network suggests multiple bounding boxes. Here, it is required to decide which one of these bounding boxes is the right one. Fortunately, to overcome this problem, a method called non-maximum suppression (NMS) is applied. Basically, what NMS does is to clean up these detections. The first step of NMS is to suppress all the prediction boxes where the confidence score is under a certain threshold value. We are setting the confidence threshold to 0.5, so every bounding box where the confidence score is less than or equal to 0.5 will be discarded.

  Yet, this method is still not sufficient to choose the proper bounding boxes, because not all unnecessary bounding boxes can be eliminated by this step, so then the second step of NMS is applied. The rest of the higher confidence scores are sorted from the highest to the lowest one, then highlight the bounding box with the highest score as the proper bounding box, and after that find all the other bounding boxes that have a high IOU (intersection over union) with this highlighted box. We have set the IOU threshold to 0.5, so every bounding box having IOU greater than 0.5 must be removed because it has a high IOU that corresponds to the same highlighted object. This method allows us to output only one proper bounding box for a detected object. This process is repeated for the remaining bounding boxes and always the bounding box with the highest score is selected as the appropriate bounding box and overlapping boxes with high IOU is removed. The same step is followed until all bounding boxes are selected properly.

Before non-maximum suppression **After non-maximum suppression**

- **Loss Function:** YOLOv3 predicts multiple bounding boxes per grid cell. To compute the loss for the true positive, we only want one of them to be responsible for the object. For this purpose, we select the one with the highest IoU (intersection over union) with the ground truth. This strategy leads to specialization among the bounding box predictions. Each prediction gets better at predicting certain sizes and aspect ratios. YOLOv3 uses sum-squared error between the predictions and the ground truth to calculate loss.

  The loss function comprises of:

  - Classification loss: If an object is detected, the classification loss at each cell is the squared error of the class conditional probabilities for each class
  - Localization loss: The localization loss measures the errors between the predicted boundary box locations and sizes and their corresponding ground truth. We only count the box responsible for detecting the object.
  - Confidence loss: The confidence loss measures the objectness of the box.

  The final loss adds localization, confidence and classification losses together.

4.2. Weight Conversion

We defined the detector's architecture. To use it, we have to either train it on our own dataset or use pretrained weights. Weights pretrained on COCO dataset are available for public use. We can download it using this link.

The structure of this binary file is such that the first 3 int32 values are header information which contains major version number, minor version number, subversion number, followed by int64 value for number of images seen by the network during training. After that, there are 62,001,757 float32 values which are weights of each convolutional and batch normalization layer. It is important to remember that they are saved in row-major format, which is opposite to the format used by Tensorflow (column-major).

In YOLOv3, there are convolutional layers which are immediately followed by batch normalization. So, the weights are applied differently for convolutional layers based on whether it is followed by batch normalization layer or not. Most of the convolution layers are immediately followed by the batch normalization layer. In this case, we need to read first read 4* num_filters weights of batch norm layer: gamma, beta, moving mean and moving variance, then {kernel_size[0] * kernel_size[1] * num_filters * input_channels} weights of convolutional layer. In the opposite case, when the convolutional layer is not followed by the batch normalization layer, instead of reading batch normalization params, we need to read num_filters bias weights. By following the same, we restore the weights of the model. This weight conversion code is implemented in the given IPython Notebook.

5. Training Model

Training a model to detect objects from scratch would take thousands of training images for each class and hours or days of training time. To speed this up, we will make use of transfer learning. Transfer learning is a process where we take the weights of a model that has already been trained on lots of data to perform a similar task, and then train the model on our own data, fine tuning the layers from the pre-trained model.

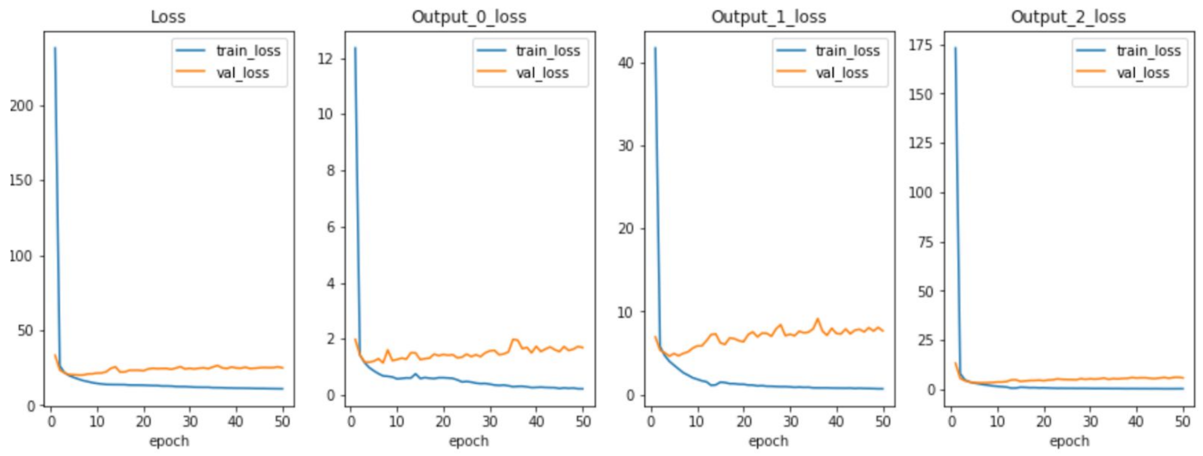
There are many models that have been trained to recognize a wide variety of objects in images. In this project, we will be using Darknet weights which are pre-trained on the COCO dataset as mentioned in the 'Weight Conversion' section. We will use the checkpoints from these trained models and then apply them to our custom object detection task. The training of the model is implemented in the given IPython Notebook.

First we set the parameters for training such as epoch, batch size, learning rate, number of classes we need to identify along with the number of classes for the pre-trained model weights which we are using. We also need to pass the created tensorflow records as training and validation sets. Once the parameters are set, we need to create the object of the YOLOv3 model we implemented in section '4.1 YOLOv3' with the weight checkpoint we converted and saved as per section '4.2 Weight Conversion'. Then we initialize the optimizer which is used for improving speed and performance for training the model. Optimizers methods are used to change the attributes of the neural network such as weights and learning rate in order to reduce the losses. Optimizers help to get results faster. Here, we will be using Adaptive Moment Estimation (Adam) optimizer. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. Then, we initialize the loss object which is our metrics to be monitored while training the model. For compiling the model, we pass parameters such as optimizer object, loss object and also set run_eagerly field to 'True'. TensorFlow's eager execution is an imperative programming environment that evaluates operations immediately, without building graphs i.e. operations return concrete values instead of constructing a computational graph to run later. This makes it easy to get started with TensorFlow and debug models.

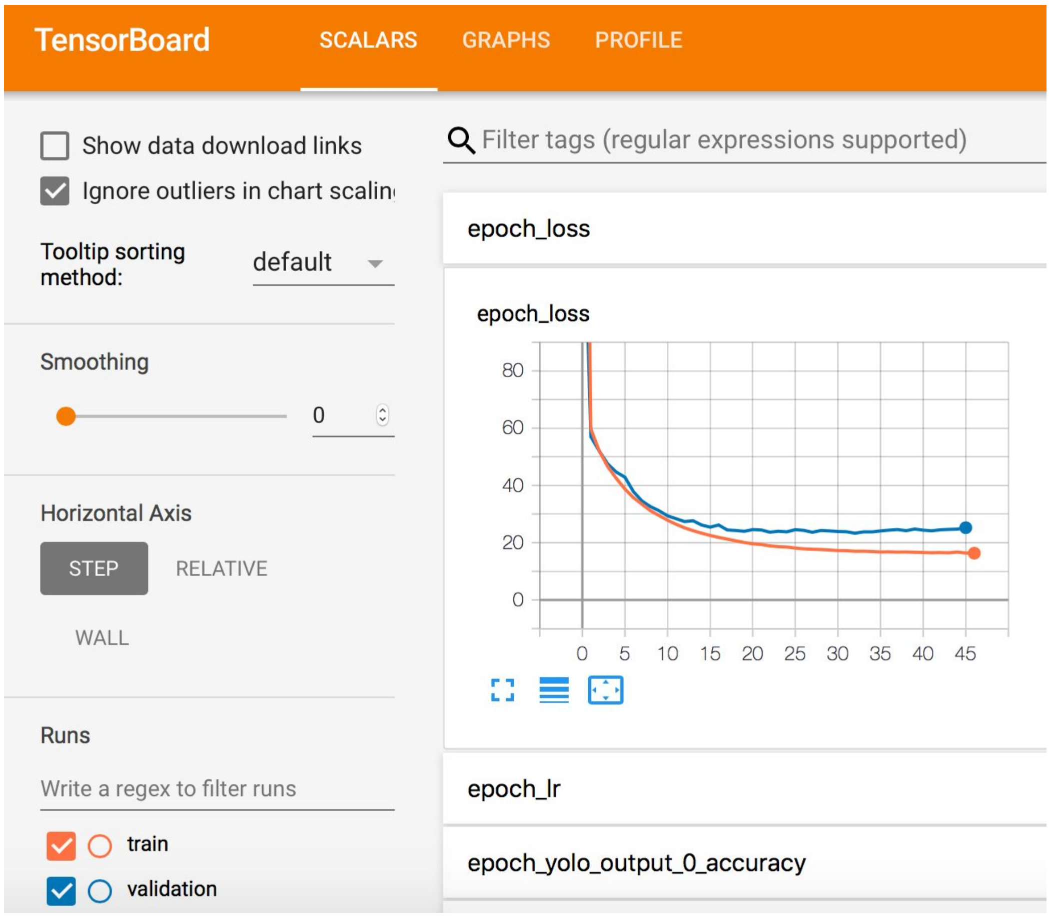
A problem with training neural networks is in the choice of the number of training epochs to use. Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset. Early stopping requires that a validation dataset is evaluated during training. So, we specify the validation dataset to the fit() method when training the model. The model is evaluated on the validation dataset at the end of each training epoch. Along with the validation dataset, we obviously pass training dataset as well to fit() method. We also need to instantiate Callbacks to pass to this fit() method.

Callbacks provide a way to execute code and interact with the training model process automatically. We can pass a list of callbacks to the callback argument when fitting the model. Here, we are using 'ReduceLROnPlateau' callback which is set to monitor loss of the model. The loss function chosen to be optimized for the model is calculated at the end of each epoch. To callbacks, this is made available via the name 'loss' and the loss on the validation dataset will be made available via the name 'val_loss'. We have also added an 'EarlyStopping' callback. This callback allows us to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process. Training will stop when the chosen performance measure stops improving. To discover the training epoch on which training was stopped, the 'verbose' argument can be set to 1. Once stopped, the callback will print the epoch number. Often, the first sign of no further improvement may not be the best time to stop training. This is because the model can get slightly worse before getting much better. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the 'patience' argument. We are setting this 'patience' value to 3. The EarlyStopping callback will stop training once triggered, but the model at the end of training may not be the model with best performance on the validation dataset. An additional callback is required that will save the best model observed during training for later use. This is the ModelCheckpoint callback. We are using this callback to save weights after each epoch completion. Lastly, we have added a TensorBoard callback to enable visualizations for TensorBoard. TensorBoard is a visualization tool provided with TensorFlow.

During the first training process, epoch is set to 50, batch size = 32 and learning rate = 0.001. The plots shown below are the plots for the loss value at each epoch. We can see that the validation loss starts to increase after a few epoch executions. This shows that the model is overfitting. Since with smaller batch size there are more weights updates, overfitting can be observed faster than with the larger batch size.



For the next training process, epoch is set to 50, batch size = 64 and learning rate = 0.01. Below is the plot from TensorBoard for the loss of the model. Here, the training loss seems to vary along with the validation loss in a similar pattern.



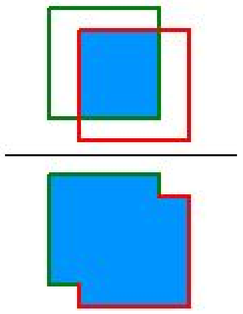
6. Testing Model

There are different metrics which are used for testing the object detection models. Here, we will be using Intersection Over Union (IOU). IOU is a measure that evaluates the overlap between two bounding boxes. It requires a ground truth bounding box B_{gt} and a predicted bounding box B_p. By applying the IOU we can tell if a detection is valid (True Positive) or not (False Positive). IOU is given by the overlapping area between the predicted bounding box and the ground truth bounding box divided by the area of union between them:

$$\text{IOU} = \frac{\text{area}\,(B_p \cap B_{gt})}{\text{area}\,(B_p \cup B_{gt})}$$

The image below illustrates the IOU between a ground truth bounding box (in green) and a detected bounding box (in red).

$$IOU = \frac{\text{area of overlap}}{\text{area of union}} =$$



- threshold: depending on the metric, it is usually pre-set to 50%, 75% or 95%.
- True Positive (TP): A correct detection. Detection with IOU ≥ threshold
- False Positive (FP): A wrong detection. Detection with IOU < threshold
- False Negative (FN): A ground truth not detected.
- True Negative (TN): It would represent a corrected misdetection. In the object detection task there are many possible bounding boxes that should not be detected within an image. Thus, TN would be all possible bounding boxes that were correctly not detected. That's why it is not used by the metrics.

The evaluation class and its related methods are implemented in this IPython Notebook. Here, the aim is to plot the Precision-Recall curve for each object class. Precision is the ability of a model to identify only the relevant objects. It is the percentage of correct positive predictions and is given by:

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{\text{all detections}}$$

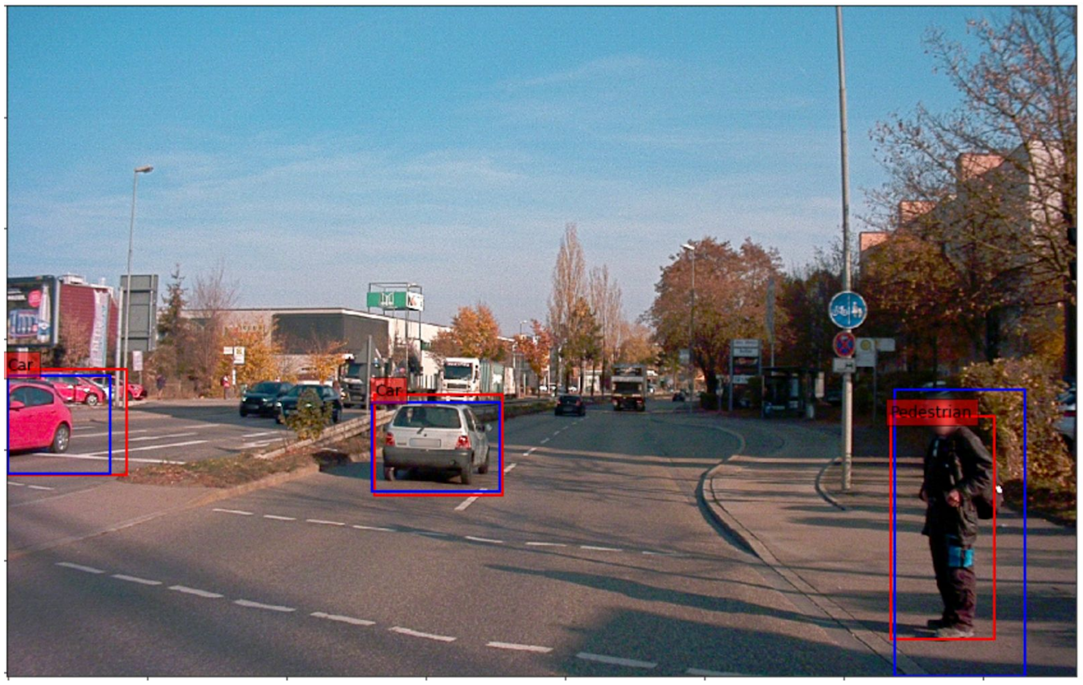
Recall is the ability of a model to find all the relevant cases (all ground truth bounding boxes). It is the percentage of true positive detected among all relevant ground truths and is given by:

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{\text{all ground truths}}$$

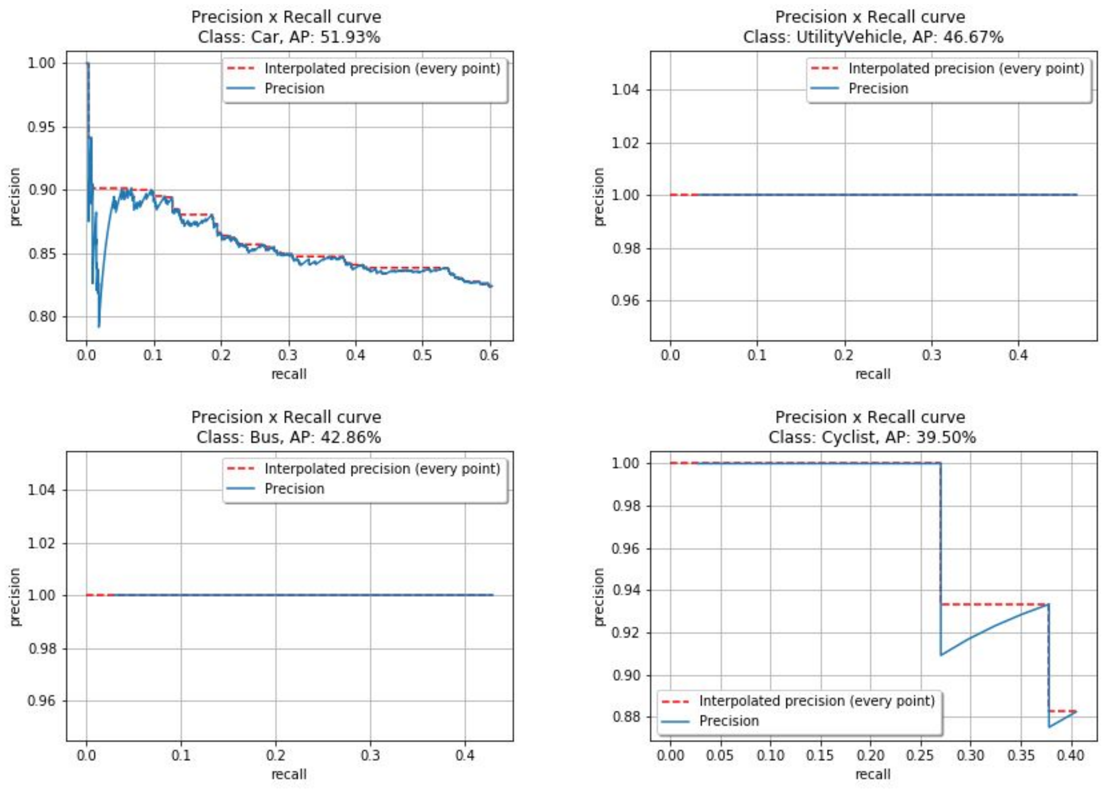
An object detector of a particular class is considered good if its precision stays high as recall increases, which means that if we vary the confidence threshold, the precision and recall will still be high. Another way to identify a good object detector is to look for a detector that can identify only relevant objects (0 False Positives = high precision), finding all ground truth objects (0 False Negatives = high recall). A poor object detector needs to increase the number of detected objects (increasing False Positives = lower precision) in order to retrieve all ground truth objects (high recall). That's why the Precision x Recall curve usually starts with high precision values, decreasing as recall increases.

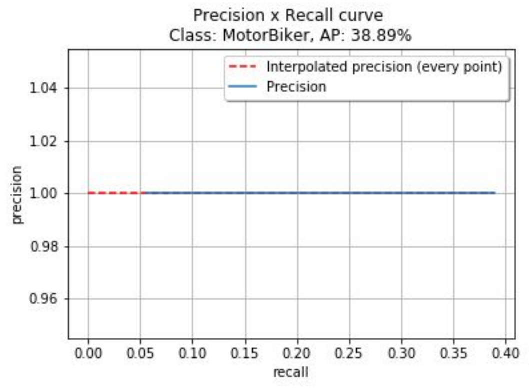
Another way to compare the performance of object detectors is to calculate the area under the curve (AUC) of the Precision x Recall curve. As AP curves are often zigzag curves going up and down, comparing different curves in the same plot usually is not an easy task - because the curves tend to cross each other much frequently. Hence, Average Precision (AP), a numerical metric, can help us compare different detectors. In practice AP is the precision averaged across all recall values between 0 and 1.

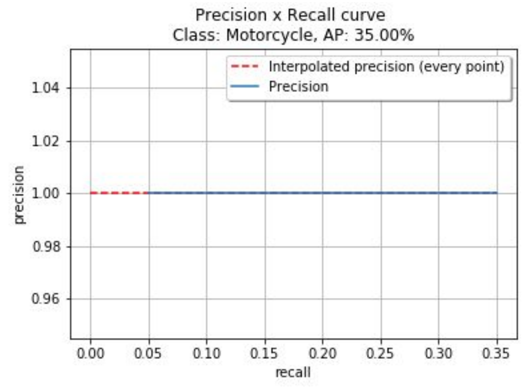
Hence, along with plotting Precision X Recall, we will be calculating the average precision value for each class. We first need to create the ground truth file and detection file. These files should contain details about bounding boxes of different classes for each image such as class, left(x1), top(y1), right(x2) and bottom(y2). For the detection of objects we need to load the YOLOv3 model with the best checkpoint weight which we got from the training stage. Then this model is executed to detect the objects in the test set images. The creation of both of this ground truth and detection file along with the final object detection testing is implemented in this IPython Notebook. Below diagram shows the ground truth bounding box in blue and detection bounding box in red.

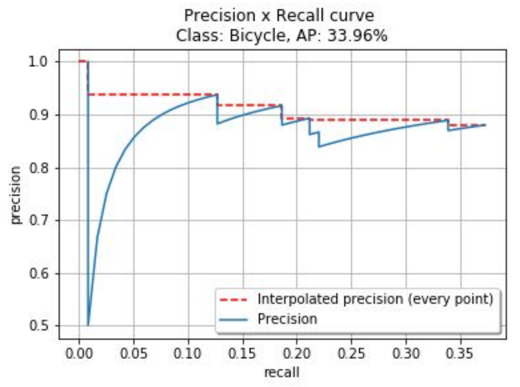


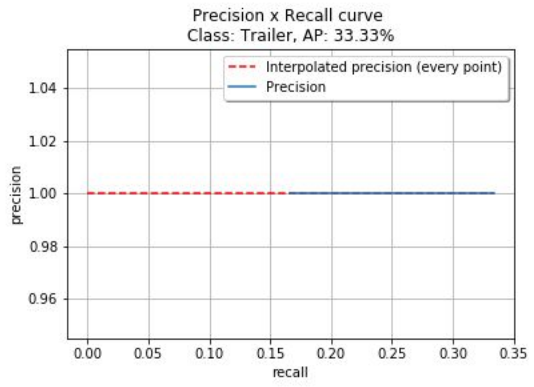
Once we have both ground truth and detection files, we can pass the data to calculate the accuracy precision and plot the Precision X Recall plot for each class detected. Below are the plots for the class which are detected by the trained model. And the mean average precision (mAP) is around 35.95%.

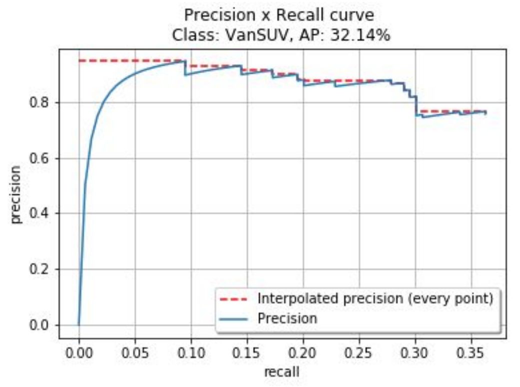


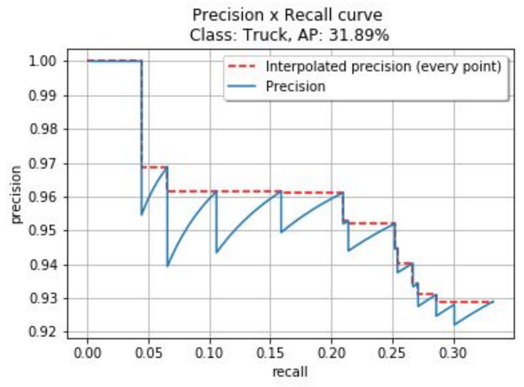


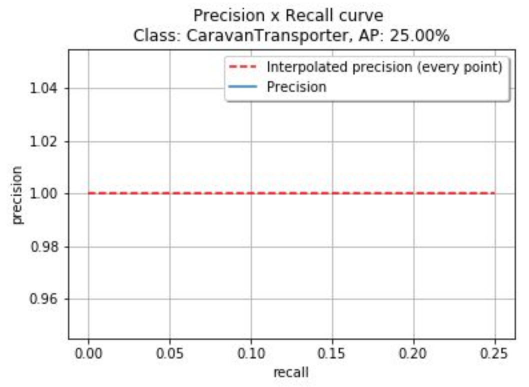


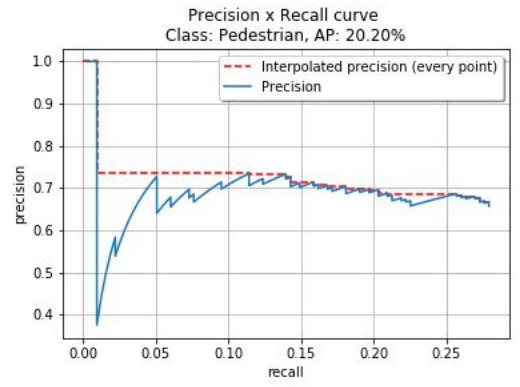




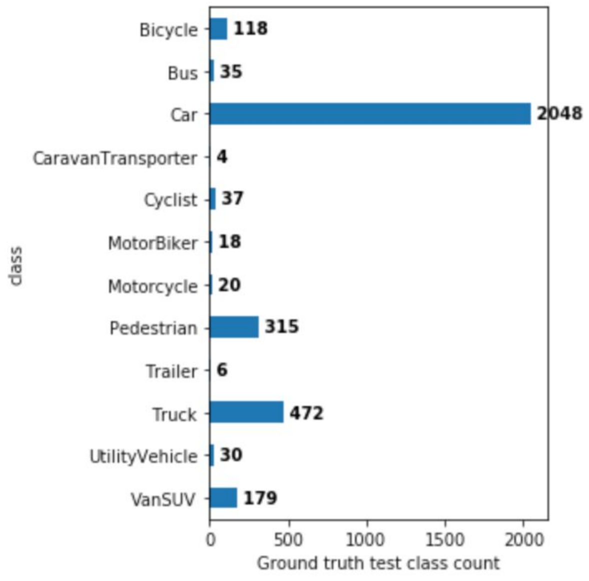


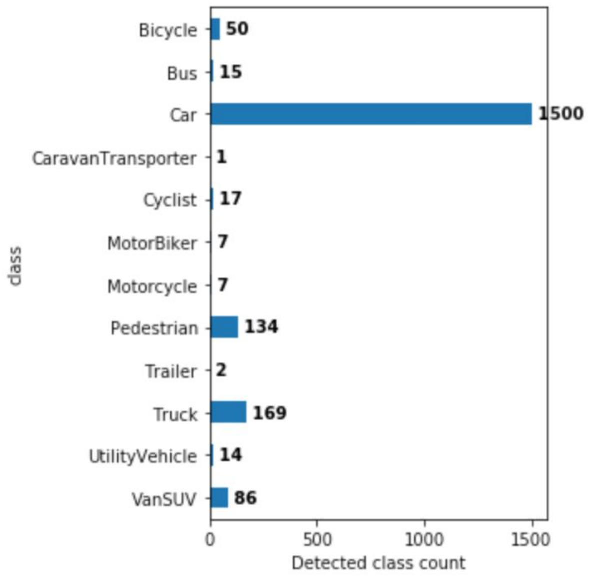




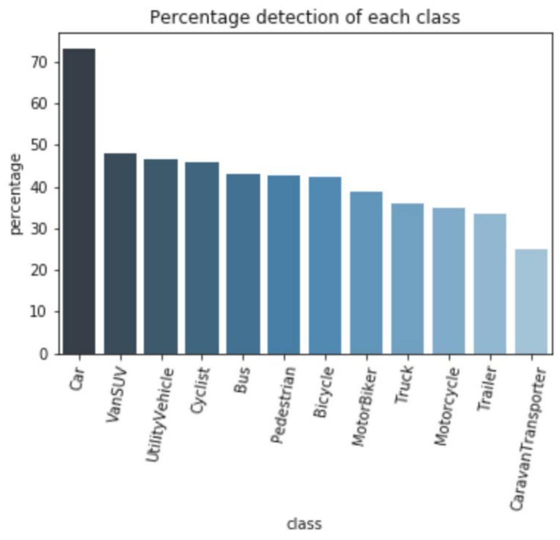


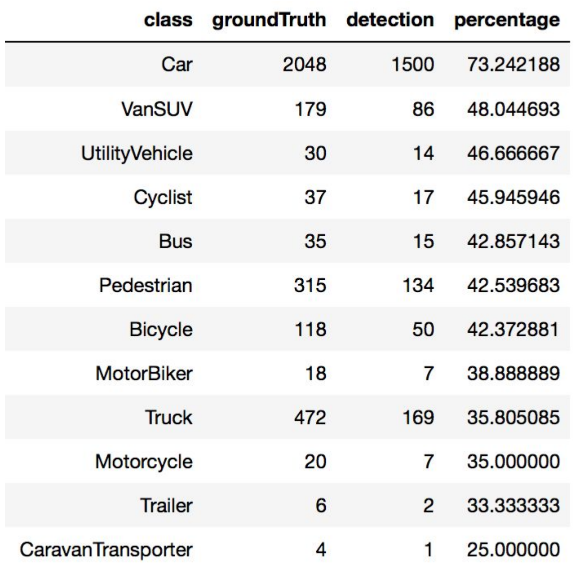
From the diagram below, we can see the total count of each class present in the ground truth file in the test set along with the total count of each class detected for this test set.





The class label ‘Car’ is detected in most of the images while class labels ‘Caravan Transporter’ and ‘Pedestrian’ have the least percentage of detection by the model.





class	groundTruth	detection	percentage
Car	2048	1500	73.242188
VanSUV	179	86	48.044693
UtilityVehicle	30	14	46.666667
Cyclist	37	17	45.945946
Bus	35	15	42.857143
Pedestrian	315	134	42.539683
Bicycle	118	50	42.372881
MotorBiker	18	7	38.888889
Truck	472	169	35.805085
Motorcycle	20	7	35.000000
Trailer	6	2	33.333333
CaravanTransporter	4	1	25.000000

From the above plot, we see that the actual detection percentage of class ‘Car’ is around 73% while the accuracy for this class is around 51%. This is because the IOU value for the detected and actual boxes must be less than the set threshold.

Based on all the observations post testing, we can infer that the model further needs more training probably by increasing the number of images for classes which have less accuracy. But this overall project gives the idea for training the object detection model using YOLOv3 architecture and TensorFlow 2.0. Also, we learnt how to test and check the performance of object detection model.

7. Future Work

- As we concluded above, we can further improve this model by training with a larger dataset. Also, we can train enough to have lesser training and validation loss.
- We can perform various other data augmentation processes such as scaling, shearing, translation, etc.
- In the Audi dataset, we also have 3D annotations and LiDAR (Light Detection and Ranging) sensor details. This project can further be enhanced by incorporating these details from the dataset for autonomous vehicles to have a better sense of surrounding. We can enhance the current project to perform 3D object detection using these additional details.