Title: Image tagging and road object detection

Group 5

Guide	**Prof. Anoop M.Namboodiri**
Mentor	**Sangeeth Reddy**
Team Members	- **Richard Deepak Vasanth Raj** - **Rahul Juluru** - **Sridhar Deshpande** - **Kshama Pandey**

Contents

1 Problem Statement

Object detection is a method of localizing and classifying an object in an image to understand the image entirely. It forms the basis of many other downstream computer vision tasks, for example, instance segmentation, image captioning, object tracking, and more. Specific object detection applications include pedestrian detection, animal detection, vehicle detection, people counting, face detection, text detection, pose detection, or number-plate recognition.

Automated driving and vehicle safety systems also need object detection. Visual data plays an essential role in enabling advanced driver-assistance systems in autonomous vehicles and improving road safety.

The object detection methods make bounding boxes around the detected objects and the predicted class label and confidence score associated with each bounding box. Object detection and tracking are challenging tasks and play an essential role in many visual-based applications. In this project, we are trying to analyze AI solutions to perform object detection and label different elements appropriately.

The main objective of the project is to build an application to detect multiple objects, tag and tracking in a video.

2 Literature review

2.1 Object Detection

The pipeline of traditional object detection models can be mainly divided into three stages: **informative region selection**, **feature extraction** and **classification**. As Referred in [1] each of these have its own challenges which mainly affect its performance both in terms of computation and true detection.

2.1.1 Convolution Neural Network

CNN is the most representative model of deep learning. The expressivity and robust training algorithms allow it to learn informative object representations without the need to design features manually.

The main advantages of CNN compared to traditional approaches are:

- Hierarchical feature representation
- Deeper architecture provides an exponentially increased expressive capability.
- Low dimensional data is recast as high-dimensional data transforms problems and is solved from a different viewpoint.

Generic object detection aims at locating and classifying existing objects in any one image, and labeling them with rectangular bounding boxes to show the confidences of existence. The framework can be broadly categorized into two types:

- *Traditional Object detection pipeline:* generating region proposals at first and then classifying each proposal into different object categories (two step process).
- *Regression or classification pipeline:* unified framework to achieve categories and locations.

2.1.2 R-CNN

By using a selective search algorithm to extract 2k region proposals from an image The selective search method relies on simple bottom-up grouping and saliency cues to provide more accurate candidate boxes of arbitrary sizes quickly and to reduce the searching space in object detection. Each region proposal is warped or cropped into a fixed resolution and the CNN module in [6] is utilized to extract a 4096-dimensional feature as the final representation. These are fed into an SVM to classify the presence of the object within that candidate region proposal. The algorithm also predicts four values which are offset values to increase the precision of the bounding box.

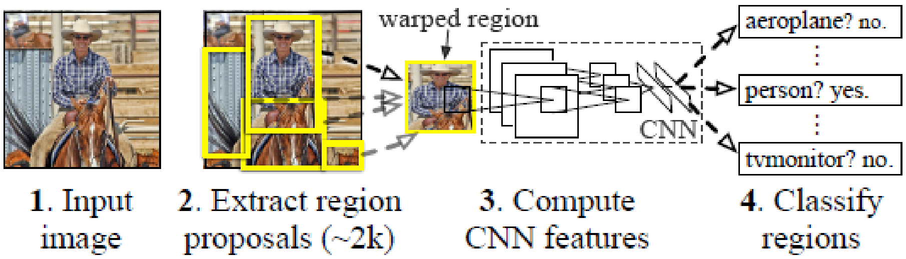


Figure 2.1 R-CNN: Regions with CNN features

2.1.3 SPP (Spatial Pyramid Pooling) Net

The fixed size constraint of a CNN network is not because of the convolution layer but because of the Fully Connected (FC) layer. SPP layers do not just apply one pooling operation, it applies a different output sized pooling operation and combines the results before sending them to the next layer. The SPPNet made the model agnostic of input image size, and drastically improved the bounding box prediction speed as compared to the R-CNN.

Though the SPP net made significant improvements in accuracy and efficiency, It still has the same multi stage pipeline of R-CNN. Storage space constraint,

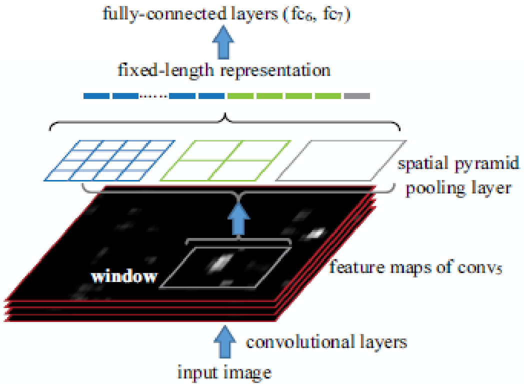

fully-connected layers (fc6, fc7)
fixed-length representation
spatial pyramid pooling layer
feature maps of conv5
window
convolutional layers
input image

Figure 2.2 Spatial Pyramid Pooling

2.1.4 Fast R-CNN

Similar to SPP-net, the whole image is processed with convolution layers to produce feature maps. Then, a fixed-length feature vector is extracted from each region proposal with a region of interest (RoI) pooling layer. The RoI pooling layer is a special case of the SPP layer, which has only one pyramid level. Each feature vector is then fed into a sequence of FC layers before finally branching into two sibling output layers - classifying and encodes refined bounding box positions with four real-valued numbers. The RoI layer is simply the special case of the spatial pyramid pooling layer with one pyramid level. The multi-tasks loss L is defined as below to jointly train classification and bounding-box regression.

2.1.5 Faster R-CNN

Region proposal computation is also a bottleneck, selective search used in all the above algorithms is a slow and time-consuming process affecting the performance of the network. A separate network, Region Proposal Network(RPN) is used in faster R-CNN to learn the region proposals. The RPN shares the network with the object detection. With the proposal of Faster R-CNN, region proposal based CNN architectures for object detection can really be trained in an end-to-end way.

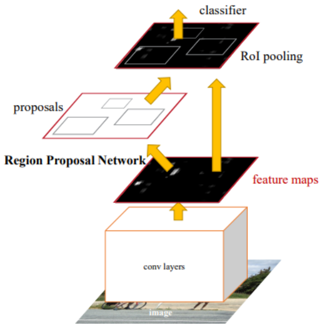


Figure 2.3 Region Proposal Network

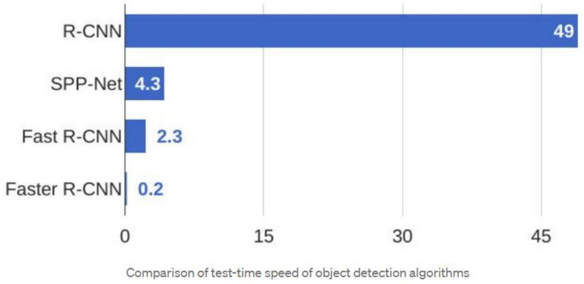


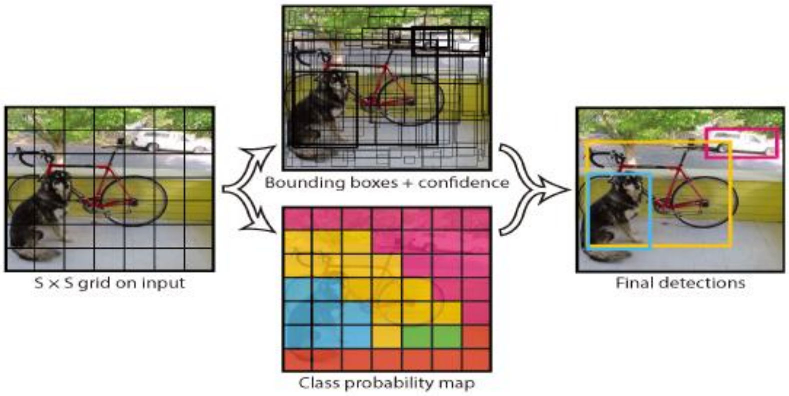
Figure 2.4 R-CNN Test-Time Speed

As can be seen from above, Faster R-CNN is much faster than its predecessors. Therefore, it can even be used for real-time object detection.

2.1.6 YOLO

Even in the Faster R-CNN being a region proposal-based framework requires an alternate training to obtain shared convolutional parameters between RPN and detection networks. In YOLO a single convolutional network predicts the bounding boxes and the class probabilities for these boxes.

YOLO divides the input image into an SXS grid and each grid cell is responsible for predicting the object centered in that grid cell. Each grid cell predicts B bounding boxes and their corresponding confidence scores, which indicates how likely there exist objects. At the same time, regardless of the number of boxes, C conditional class probabilities should also be predicted in each grid cell.



There are multiple versions of Yolo, some of which are described below:

2.1.6.1 YOLO V1[2015]

The YOLOv1 processed images in real-time at 45 frames per second, while a smaller version – Fast YOLO – reached 155 frames per second and still achieved double the mAP of other real-time detectors.

2.1.6.2 YOLO v5[2020]

most popular object detection algorithm. It is just the PyTorch extension of YOLOv3. Its speed is similar to yolo v4.

2.1.6.3 YOLO v7[2022]

Latest update keeping in mind the amount of memory it takes to keep layers in memory along with the distance that it takes a gradient to back-propagate through the layers. The final layer aggregation they choose is E-ELAN an extend version of the ELAN computational block.

2.1.6.4 YOLO V8[2023]

YOLOv8 is a cutting-edge, state-of-the-art (SOTA) model that builds upon the success of previous YOLO versions and introduces new features and improvements to further boost performance and flexibility. YOLOv8 is designed to be fast, accurate, and easy to use, making it an excellent choice for a wide range of object detection, image segmentation and image classification tasks.

Region proposal-based methods, such as Faster R-CNN and R-FCN, perform better than regression/ classification-based approaches, namely YOLO due to the fact that quite a lot of localization errors are produced by regression/classification-based approaches. But regression-based approaches are much faster which is a necessity for running object detection on videos.

2.2 Object Tracking

Once the objects are identified and bounded in the image, we need to track them in the subsequent video images. We perform tracking instead of detection every frame due to the following reasons.

- Tracking is faster than Detection
- Tracking can help when detection fails and it also helps in handling some level of occlusion.
- Tracking preserves identity: (Intra class association)

Common problems with Multiple Object Tracking (MOT) are

1. Long time occlusions
2. Initialization and termination of tracks
3. Similar objects
4. Interacting among multiple objects

We propose to run the object detection every N frame to correct the tracking errors.

OpenCV has different tracker types and let us see the different options they are providing.

2.2.1 Boosting

This method is based on the online version of the AdaBoost algorithm - the algorithm increases the weights of incorrectly classified objects, which allows a weak classifier to “focus” on their detection.

2.2.2 Multiple Instance Learning (MIL)

This algorithm has the same approach as BOOSTING, however, instead of guessing where the tracked object is in the next frame, an approach is used in which several potentially positive objects, called a “bag”, are selected around a positive definite object. A positive “bag” contains at least one positive result.

2.2.3 Kernelized correlation Filter (KCF)

Is a combination of two algorithms: BOOSTING and MIL. The concept of the method is that a set of images from a “bag” obtained by the MIL method has many overlapping areas. Correlation filtering applied to these areas makes it possible to track the movement of an object with high accuracy and to predict its further position.

2.2.4 Tracking learning detection (TLD)

This method allows you to decompose the task of tracking an object into three processes: tracking, learning and detecting. The tracker (based on the Median Flow tracker) tracks the object, while the detector localizes external signs and corrects the tracker if necessary. The learning part evaluates detection errors and prevents them in the future by recognizing missed or false detections.

2.2.5 Median Flow

This algorithm is based on the Lucas-Kanade method. The algorithm tracks the movement of the object in the forward and backward directions in time and estimates the error of these trajectories, which allows the tracker to predict the further position of the object in real-time.

2.2.6 Generic Object Tracking Using Regression Networks (GOTURN)

This algorithm is an “offline” tracker since it basically contains a deep convolutional neural network. Two images are fed into the network: “previous” and “current”. In the “previous” image, the position of the object is known, while in the “current” image, the position of the object must be predicted. Thus, both images are passed through a convolutional neural network, the output of which is a set of 4 points representing the coordinates of the predicted bounding box containing the object. Since the algorithm is based on the use of a neural network, the user needs to download and specify the model and weight files for further tracking of the object.

2.2.7 Minimum Output Sum of Squared Errors (MOSSE)

This algorithm is based on the calculation of adaptive correlations in Fourier space. The filter minimizes the sum of squared errors between the actual correlation output and the predicted correlation output. This tracker is robust to changes in lighting, scale, pose, and non-rigid deformations of the object.

2.2.8 Channel and Spatial Reliability Tracker (CSRT)

This algorithm uses spatial reliability maps for adjusting the filter support to the part of the selected region from the frame for tracking, which gives an ability to increase the search area and track non-rectangular objects. Reliability indices reflect the quality of the studied filters by channel and are used as weights for localization. Thus, using HoGs and Color names as feature sets, the algorithm performs relatively well.

For the evaluation we want to use Multiple Object Tracking Accuracy (MOTA) and Multiple Object Tracking Precision (MOTP) metrics.

3 Methodology & Design Considerations

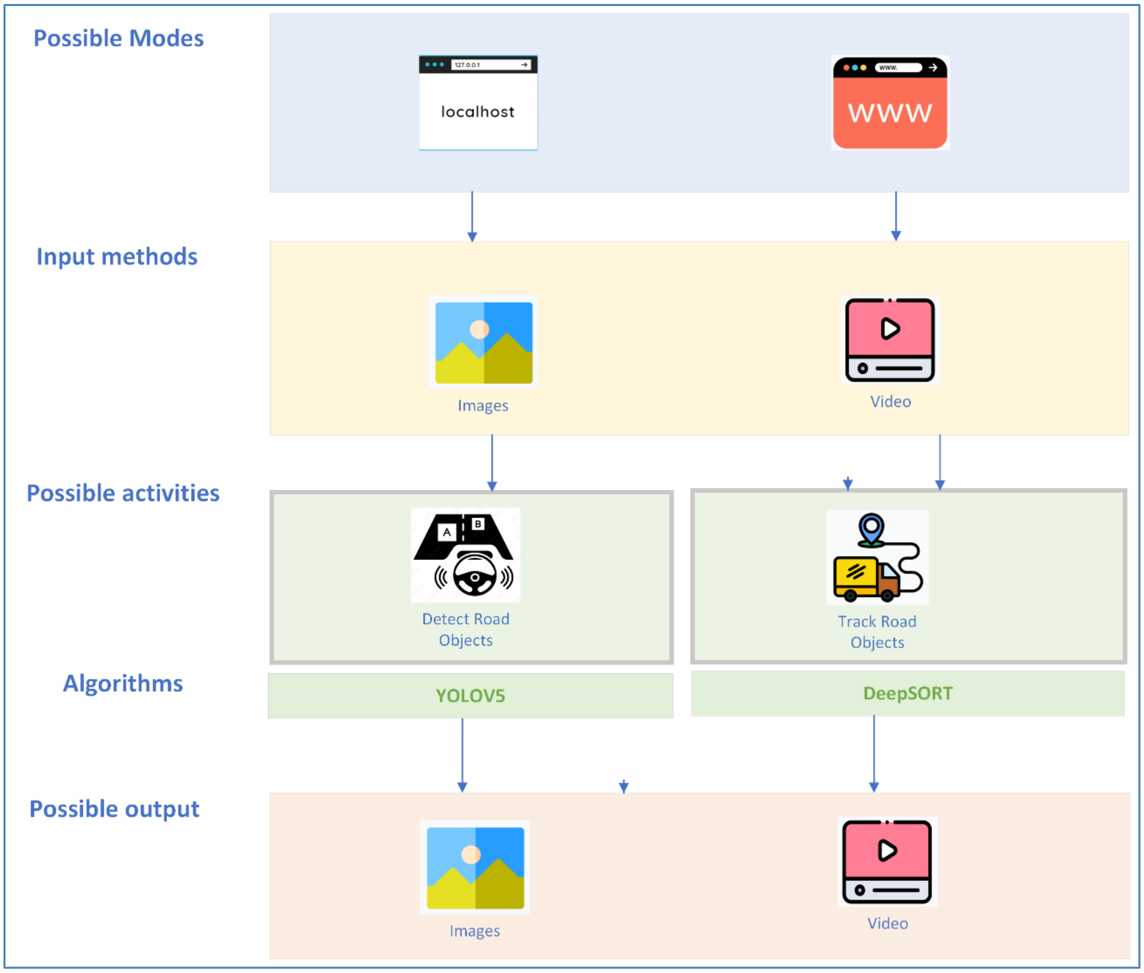


Figure 3.1 Methodology and Design Considerations

In order to implement object detection and tracking for road objects, we have to consider several possibilities of the following:

3.1 Possible modes

The invocation of algorithm is dependent upon the intent form where consumer wanted to interact with and the possibilities can be either as an end point REST API , through terminal command line , from browsers as web application or as desktop application.

3.2 Input Methods

The mode of input for detection also plays a critical role as the approach of running the solution becomes tricky and the level of complexity necessary to sustain the results becomes a matter of concern quickly. As the first step , we can start with providing image or screenshot or video for detecting and tracking road scene objects and increase scope for webcam or screen grab and perform the necessary operations.

3.3 Possible activities

As we are trying to scope the project to detect objects like car , bike , bicycle , pedestrian etc. on road scene , we are restricting our activities to two which are detecting objects (focus on the desired object and mark it with confidence score) and tracking objects (assigning a class and an unique ID which should hold its identity until we complete analyzing the input).

3.4 Possible output

Depending on the restrictions we obtain from output we receive from the algorithms we are going to choose during implementation, we restricted the format of outputs to be image or video for a given input of image/screenshot and video respectively.

4 Outcome

4.1 Outcome in stages

The goal of creating a web application which is capable of performing object detection and tracking can be achieved by the high-level stages mentioned below.

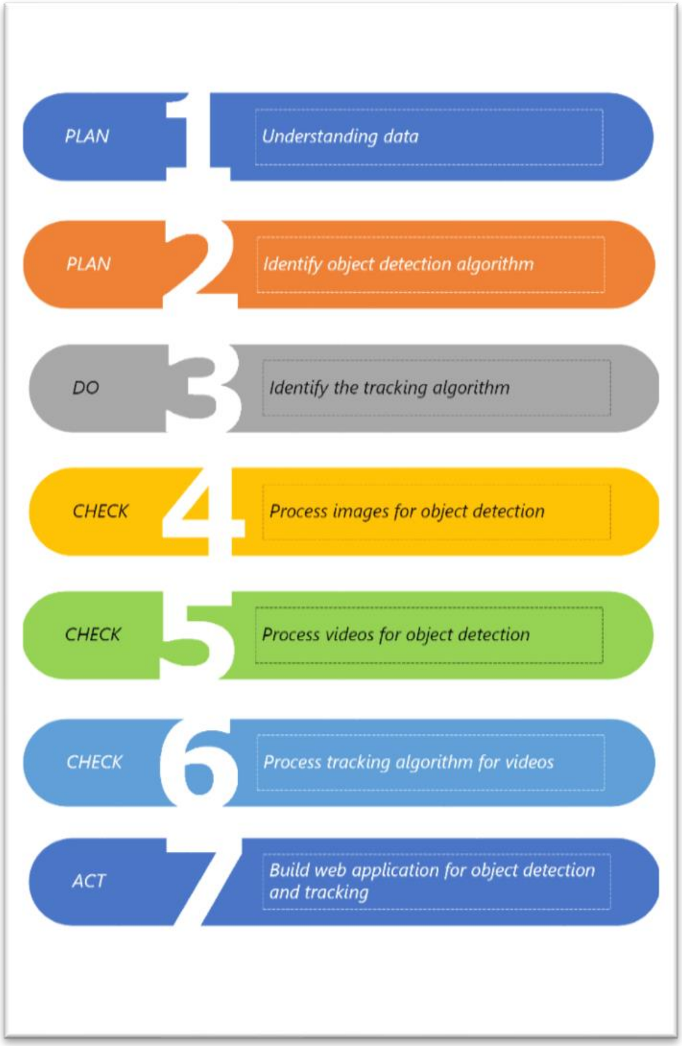

PLAN
1
Understanding data
PLAN
2
Identify object detection algorithm
DO
3
Identify the tracking algorithm
CHECK
4
Process images for object detection
CHECK
5
Process videos for object detection
CHECK
6
Process tracking algorithm for videos
ACT
7
Build web application for object detection and tracking

Figure 4.1 Stage wise outcome

4.1.1 Understanding data

a. Understand the essence of BDD100K dataset which can facilitate algorithmic study on large-scale diverse visual data and multiple tasks and the data contains images and videos. We have understood that there are 12 classes upon which we are going to train the model.

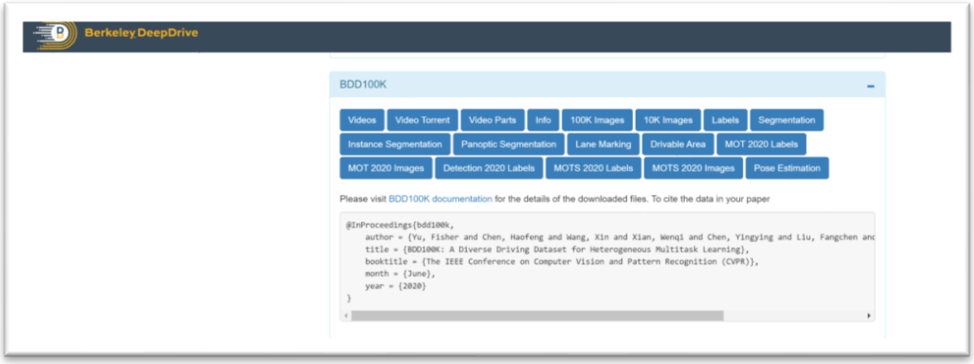
b. BDD100K dataset is unique in its choice of data it can provide because of the following reasons:

   i. **Large-Scale**: 100K driving videos collected from more than 50K rides. Each video is 40-second long and 30fps. More than 100 million frames in total.

   ii. **Diverse:** Diverse scene types including city streets, residential areas, and highways, and diverse weather conditions at different times of the day.

   iii. **Multi-task:** Lane detection, object detection, semantic segmentation, instance segmentation, panoptic segmentation, multi-object tracking, segmentation tracking and more.

c. Download the BDD100K data from the **link** after registering as and end user. The challenge is the bandwidth that website supports to get few videos and images with labels data that is essentials for our training.



Please visit BDD100K documentation for the details of the downloaded files. To cite the data in your paper

```
@InProceedings{bdd100k,
    author = {Yu, Fisher and Chen, Haofeng and Wang, Xin and Xian, Wenqi and Chen, Yingying and Liu, Fangchen anc
    title = {BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning},
    booktitle = {The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)},
    month = {June},
    year = {2020}
}
```

Figure 4.2 Berkeley DeepDrive BDD100K website

d. Create a dedicated Google drive account and upload all the necessary data by taking up additional space and provide access to all developers responsible for project delivery.

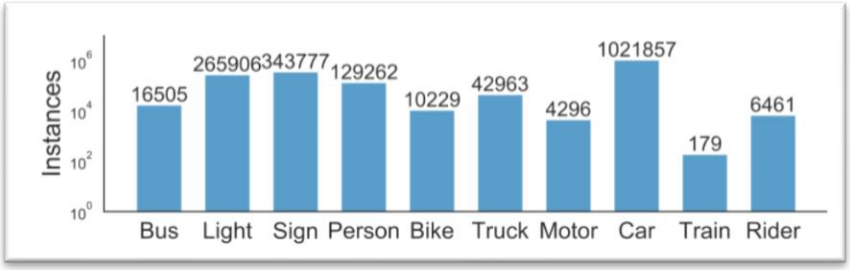


Figure 4.3 Classes defined in BDD100K Dataset

After downloading the BDD100K data, we can observe that the training and validation folders contain their videos or images respectively. There are labels for each of the image to indicate the possible classes available at each image or in video frame with required co-ordinates in JSON format.

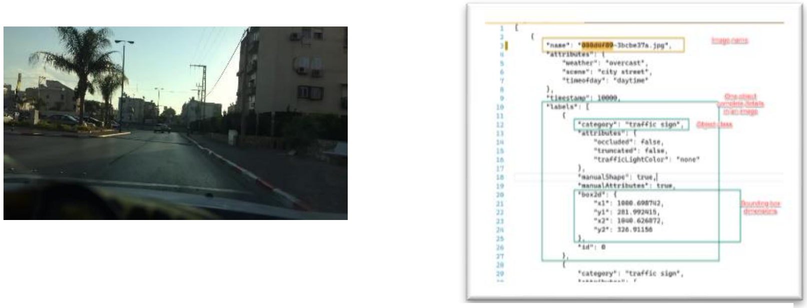

```
[
    {
        "name": "00a0f008-3bcbe37a.jpg",
        "attributes": {
            "weather": "overcast",
            "scene": "city street",
            "timeofday": "daytime"
        },
        "timestamp": 10000,
        "labels": [
            {
                "category": "traffic sign",
                "attributes": {
                    "occluded": false,
                    "truncated": false,
                    "trafficLightColor": "none"
                },
                "manualShape": true,
                "manualAttributes": true,
                "box2d": {
                    "x1": 1000.698742,
                    "y1": 281.992415,
                    "x2": 1040.626872,
                    "y2": 326.91156
                },
                "id": 0
            },
            {
                "category": "traffic sign",
```

Figure 4.4 Objects in an image and their representations in JSON format

4.1.2 Identify object detection

Upload all the data that we have obtained into Google Drive and we are going to consider Google Colab Pro for obtaining the desired model which can be used as base for object detection and tracking.

After much deliberation and thorough analysis from the algorithms we have identified during literature review, we have realized that YOLOV5 might be a great fit considering the help documents and community support it provides comparatively with other YOLO versions.

There are several stages in understanding the logic behind the YOLOV5 and its being provided by Ultralytics which was trained with COCO128 dataset. Here are the steps that its follows behind the scene to detect an object within a frame or image:

4.1.2.1 Divide the image into cells

The first step within YOLOV5 algorithm is to divide the image to cells with an S X S grid where S is number of rows and columns.

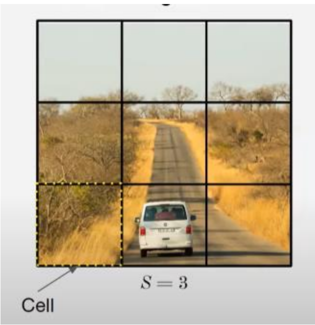


Figure 4.5 Divide the image to cells

4.1.2.2 Prediction of bounding boxes for each cell

A cell is responsible for detecting an object if the object's bounding box falls within the cell. Notice that each cell has 2 blue dots.

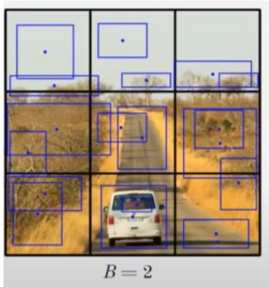


Figure 4.6 Predicting bounding boxes within each cell

4.1.2.3 Return bounding boxes above confidence threshold

All other bounding boxes have a confidence probability less than the threshold(say 0.90) so they are suppressed.

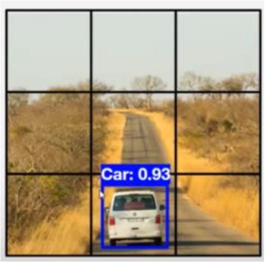


Figure 4.7 Selection of bounding box which is above confidence threshold

In order to understand more about the bounding boxes calculation, let's use a simple example where there are 3 X 3 cells(S = 3) , each cell predicts 1 bounding box (B=1) , and objects are either dog = 1 or human = 2 .For each cell, the CNN predicts a vector "y" as shown below in diagram which contains the following:

- Probability the bounding box contains an object
- Coordinates of the bounding box's center

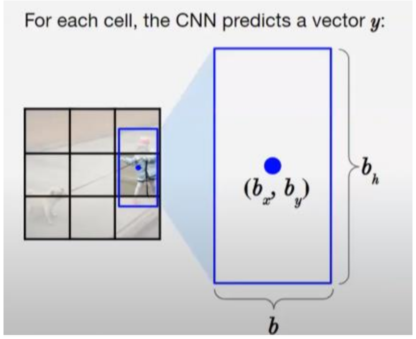
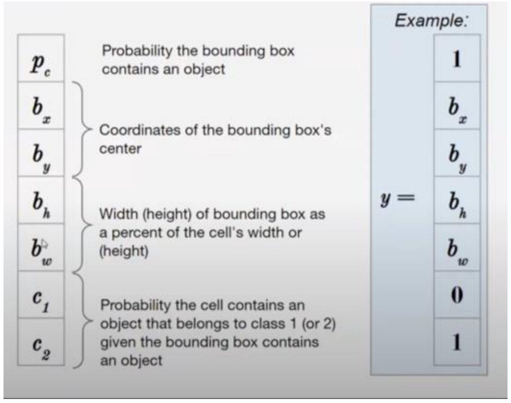

For each cell, the CNN predicts a vector y:

Figure 4.8 Co-ordinates of bounding box

- Width(height) of bounding box as a percent of the cell's width or (height)
- Probability the cell contains an object that belongs to class 1(or 2) given the bounding box contains an object


p_c Probability the bounding box contains an object

b_x, b_y Coordinates of the bounding box's center

b_h, b_w Width (height) of bounding box as a percent of the cell's width or (height)

c_1, c_2 Probability the cell contains an object that belongs to class 1 (or 2) given the bounding box contains an object

Example:

$$y = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \end{bmatrix}$$

Figure 4.9 Probability of object with co-ordinates

4.1.2.4 Union over intersection (UoI)

Once the objects within all cells are identified, we are going to calculate the Union over Intersection (UoI) which measures the overlap between tow bounding boxes. During training, we calculate the UoI between a predicted bounding box and the ground truth(the prelabeled bounding box we main to match).

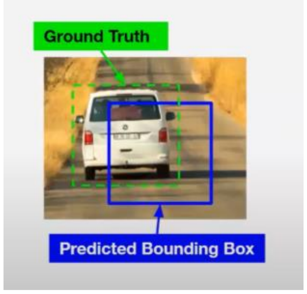
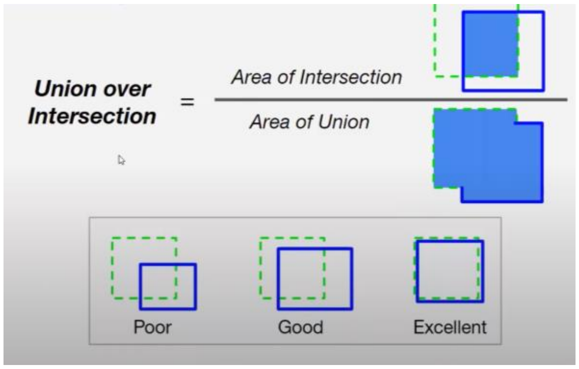


Figure 4.10 Ground truth - Bounding box

Union over intersection = Area of intersection / Area of union

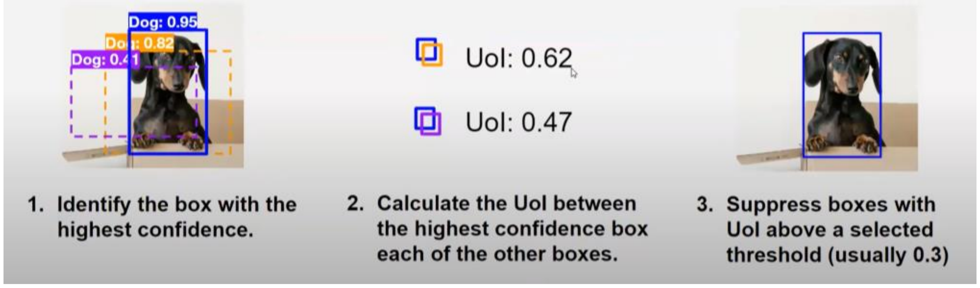


Union over Intersection = *Area of Intersection* / *Area of Union*

Figure 4.11 Pictorial representation of UoI

4.1.2.5 Double Counting Object (Non-max suppression)

Sometimes the same object will be detected multiple times. Non-max suppression solves multiple counting by removing the box with the lower confidence probability when the UoI between the two boxes with the same label is above some threshold.



1. Identify the box with the highest confidence.
2. Calculate the UoI between the highest confidence box each of the other boxes.
3. Suppress boxes with UoI above a selected threshold (usually 0.3)

Figure 4.12 Non-max suppression

4.1.3 Identifying the tracking algorithm

DeepSORT is a computer vision tracking algorithm for tracking objects while assigning an ID to each object. DeepSORT is an extension of the SORT (Simple Online Realtime Tracking) algorithm. DeepSORT introduces deep learning into the SORT algorithm by adding an appearance descriptor to reduce identity switches , hence making tracking more efficient. To understand DeepSORT, First Let's see how the SORT algorithm works.

SORT is an approach to Object tracking where rudimentary approaches like Kalman filters and Hungarian algorithms are used to track objects and claim to be better than many online trackers. SORT is made of 4 key components which are as follows:

4.1.3.1 Detection

This is the first step in the tracking module. In this step, an object detector detects the objects in the frame that are to be tracked. These detections are then passed on to the next step. Detectors like FrRCNN, YOLO, and more are most frequently used.

4.1.3.2 Estimation

In this step, we propagate the detections from the current frame to the next which is estimating the position of the target in the next frame using a constant velocity model. When a detection is associated with a target, the detected bounding box is used to update the target state where the velocity components are optimally solved via the Kalman filter framework

4.1.3.3 Data association

We now have the target bounding box and the detected bounding box. So, a cost matrix is computed as the intersection-over-union (IOU) distance between each detection and all predicted bounding boxes from the existing targets. The assignment is solved optimally using the Hungarian algorithm. If the IOU of detection and target is less than a certain threshold value called IOUmin then that assignment is rejected. This technique solves the occlusion problem and helps maintain the IDs.

4.1.3.4 Creation and Deletion of Track Identities

This module is responsible for the creation and deletion of IDs. Unique identities are created and destroyed according to the IOU_{min}. If the overlap of detection and target is less than IOU_{min} then it signifies the untracked object. Tracks are terminated if they are not detected for T_{Lost} frames, you can specify what the amount of frame should be for T_{Lost}. Should an object reappear, tracking will implicitly resume under a new identity.

4.1.4 Process images & videos for object detection

There are several steps to perform to come up with best model (weights) which can be used for object detection to any new input after training BDD100K dataset with required customizations as follows:

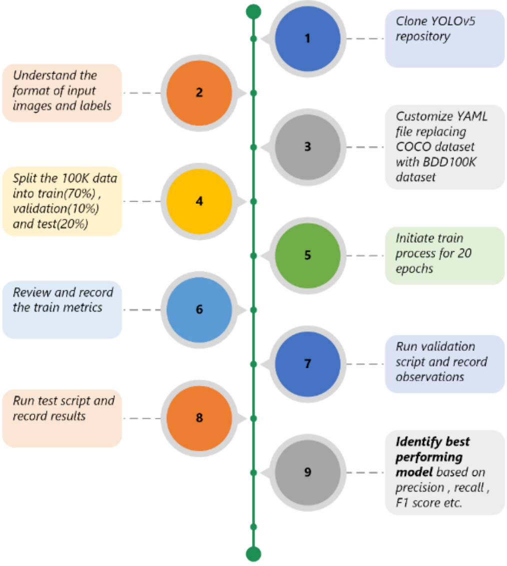
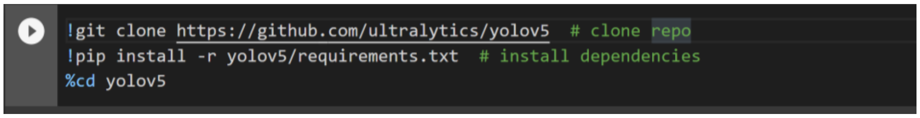

1
Clone YOLOv5 repository
2
Understand the format of input images and labels
3
Customize YAML file replacing COCO dataset with BDD100K dataset
4
Split the 100K data into train(70%) , validation(10%) and test(20%)
5
Initiate train process for 20 epochs
6
Review and record the train metrics
7
Run validation script and record observations
8
Run test script and record results
9
Identify best performing model based on precision , recall , F1 score etc.

Figure 4.13 Implementation of detection algorithm

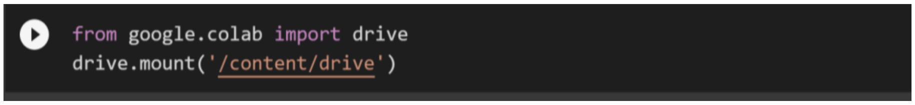
4.1.4.1 Clone YOLOv5 repository

For training the BDD100K dataset, we are going to use Google Colab Pro+ . The first step for using YOLOv5 is to create a clone of YOLOv5 repository and understand the folder structure that we obtain. Additionally, map the google drive correspondingly to use the images we have downloaded in previous steps for training purposes.



```
!git clone https://github.com/ultralytics/yolov5  # clone repo
!pip install -r yolov5/requirements.txt  # install dependencies
%cd yolov5
```

Figure 4.14 Clone the repository

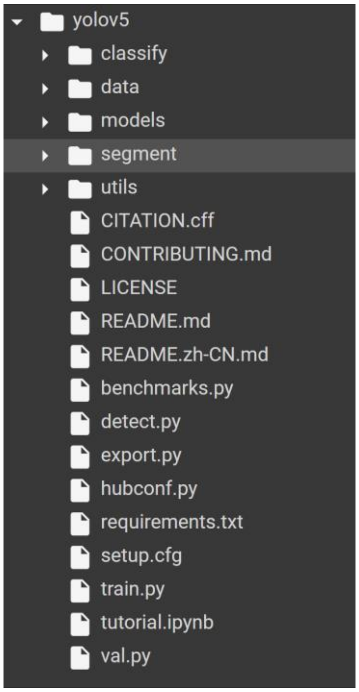


```
from google.colab import drive
drive.mount('/content/drive')
```

Figure 4.15 Mount the Google drive to load the content

4.1.4.2 Understand the format of input images and labels

Understand the folder structure of repository and check the places where we need to customize the settings to train BDD100K dataset as YOLOv5 is pretrained on COCO128 dataset. The images provided are in JPEG format with resolution 1280 X 720. Labels are provided in JSON files and we can extract it at the same level as yolov5 for training purposes.

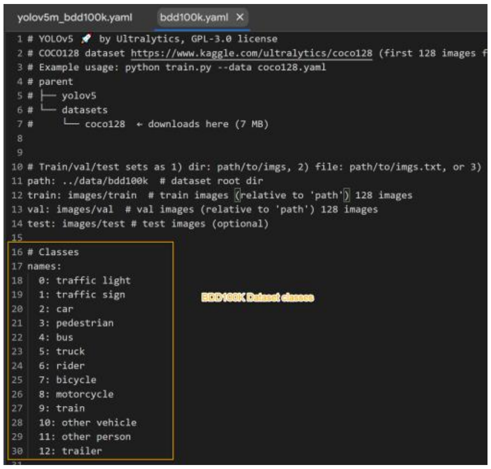


```
yolov5
    classify
    data
    models
    segment
    utils
    CITATION.cff
    CONTRIBUTING.md
    LICENSE
    README.md
    README.zh-CN.md
    benchmarks.py
    detect.py
    export.py
    hubconf.py
    requirements.txt
    setup.cfg
    train.py
    tutorial.ipynb
    val.py
```

Figure 4.16 Folder structure after cloning

4.1.4.3 Customize YAML file replacing with BDD100K related datasets

Cleanup the existing YAML file to accommodate the number of classes with their dictionary value for each class and place it in appropriate folder structure.

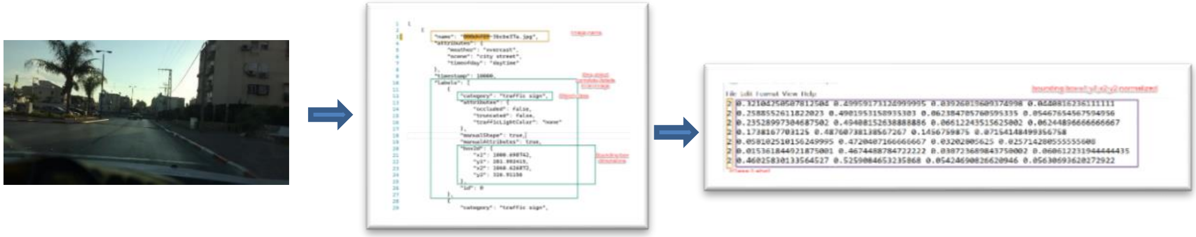


```
# YOLOv5 🚀 by Ultralytics, GPL-3.0 license
# COCO128 dataset https://www.kaggle.com/ultralytics/coco128 (first 128 images f
# Example usage: python train.py --data coco128.yaml
# parent
# ├── yolov5
# └── datasets
#     └── coco128  ← downloads here (7 MB)


# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3)
path: ../data/bdd100k  # dataset root dir
train: images/train  # train images (relative to 'path') 128 images
val: images/val  # val images (relative to 'path') 128 images
test: images/test # test images (optional)

# Classes
names:
  0: traffic light
  1: traffic sign
  2: car
  3: pedestrian
  4: bus
  5: truck
  6: rider
  7: bicycle
  8: motorcycle
  9: train
  10: other vehicle
  11: other person
  12: trailer
```

Figure 4.17 Customized BDD100K YAML file

4.1.4.4 Split the BDD100K data into train, validation and test data

Before we start splitting the BDD100K, there is prerequisite task to convert JSON format of label files we got from BDD website to text format that is supported by YOLOv5 as shown in the figure.

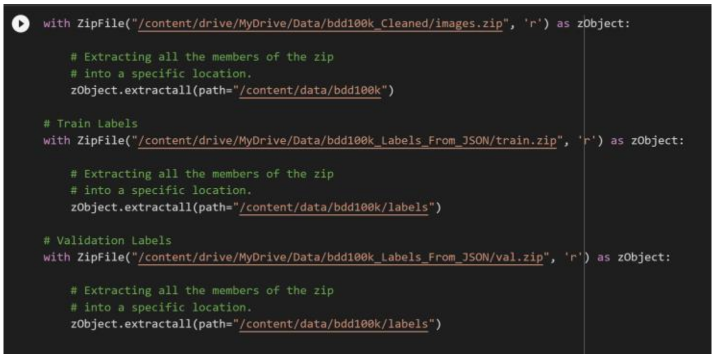


```
{
    {
        "name": "[illegible]-3bcbe77a.jpg",
        "attributes": {
            "weather": "overcast",
            "scene": "city street",
            "timeofday": "daytime"
        },
        "timestamp": 10000,
        "labels": [
            {
                "category": "traffic sign",
                "attributes": {
                    "occluded": false,
                    "truncated": false,
                    "trafficLightColor": "none"
                },
                "manualShape": true,
                "manualAttributes": true,
                "box2d": {
                    "x1": 1000.698742,
                    "y1": 281.992415,
                    "x2": 1040.626872,
                    "y2": 326.91158
                },
                "id": 0
            },
            {
                "category": "traffic sign",
```

```
2 0.32104250507812504 0.29959173124999995 0.03926019600374998 0.04408162361111111
2 0.2588552611822023 0.4901953158935303 0.06238470576059533 0.05467654567594956
2 0.23528997304687502 0.49408152638888886 0.06612243515625002 0.06244896666666667
2 0.1738167703125 0.48760738138567267 0.1456759875 0.07154148499356758
2 0.05810251015624999 0.47204071666666667 0.03202805625 0.02571428055555600
2 0.01536184492187500 0.46744887847222222 0.03872368984375000 0.06061223194444435
2 0.46025830133564527 0.52590846532358 0.05424690826620946 0.05630693620272922
```

Figure 4.18 Converting label JSON to text format

After conversion, upload generated files in Google Drive



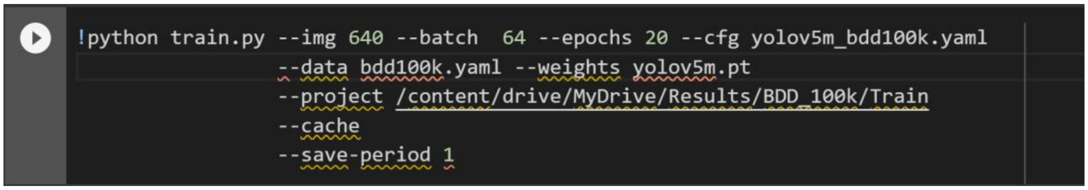
```
with ZipFile("/content/drive/MyDrive/Data/bdd100k_Cleaned/images.zip", 'r') as zObject:

    # Extracting all the members of the zip
    # into a specific location.
    zObject.extractall(path="/content/data/bdd100k")

# Train Labels
with ZipFile("/content/drive/MyDrive/Data/bdd100k_Labels_From_JSON/train.zip", 'r') as zObject:

    # Extracting all the members of the zip
    # into a specific location.
    zObject.extractall(path="/content/data/bdd100k/labels")

# Validation Labels
with ZipFile("/content/drive/MyDrive/Data/bdd100k_Labels_From_JSON/val.zip", 'r') as zObject:

    # Extracting all the members of the zip
    # into a specific location.
    zObject.extractall(path="/content/data/bdd100k/labels")
```

Figure 4.19 Loading of train, validation and test images with labels

Reformat the images according to train, validation and test folders with their labels correspondingly into the colab. Fortunately, BDD provides a same name for the image and their corresponding label file as well which eases development for mapping image and their label files.

4.1.4.5 Initiate train process for 20 epochs

Run detect with appropriate parameters essential to store the best model weights for 20 epochs to validate if its working as per expectations or not. If needed, adjust the hyperparameters essential during the train run.



```
!python train.py --img 640 --batch  64 --epochs 20 --cfg yolov5m_bdd100k.yaml
                 --data bdd100k.yaml --weights yolov5m.pt
                 --project /content/drive/MyDrive/Results/BDD_100k/Train
                 --cache
                 --save-period 1
```

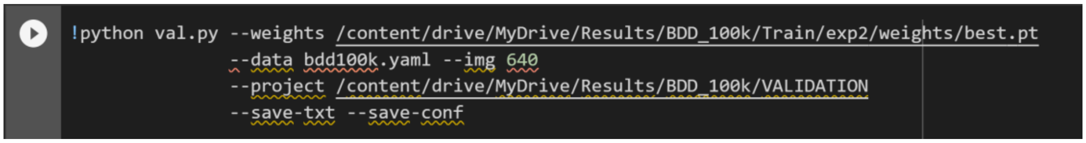
Figure 4.20 Train the BDD100K data for 20 epochs

4.1.4.6 Review and record train metrics

Record all the metrics and review if accuracy, precision and recall are improving for each epoch. The generated results are provided as part of actual output section in 4.2.

4.1.4.7 Run validation script and record observations

After completion of 200 epochs, it’s time to run validation set and check the accuracy for the new data as this is not yet trained for those images. Observe the performance metrics as we did previously and retrain if the results are not promising. This is a tedious process but surely demands a lot of attention as this is the crux of entire project efficiency.

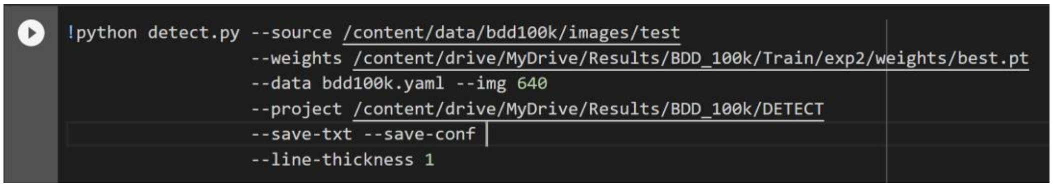


```
!python val.py --weights /content/drive/MyDrive/Results/BDD_100k/Train/exp2/weights/best.pt
               --data bdd100k.yaml --img 640
               --project /content/drive/MyDrive/Results/BDD_100k/VALIDATION
               --save-txt --save-conf
```

Figure 4.21 Validation run for 10% of BDD100K data

4.1.4.8 Run test script on images and record results.

Once we are good with validation and train phases, we need to head towards running the solution against test data which accounts for 20% of test data. This is the final litmus test to see how efficiently we have trained the algorithm. Record all the results generated for the run for final observation before we start utilizing for detection.

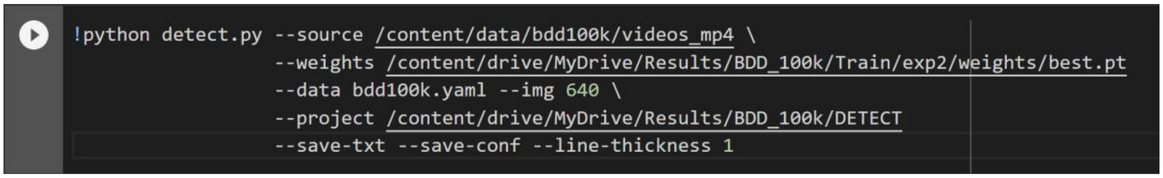


```
!python detect.py --source /content/data/bdd100k/images/test
                  --weights /content/drive/MyDrive/Results/BDD_100k/Train/exp2/weights/best.pt
                  --data bdd100k.yaml --img 640
                  --project /content/drive/MyDrive/Results/BDD_100k/DETECT
                  --save-txt --save-conf
                  --line-thickness 1
```

Figure 4.22 Detection test with weights generated for training and validation phases

4.1.4.9 Run test script on videos and record results

Once we are good with validation and train phases, we need to head towards running the solution against test data which accounts for 20% of test data. This is the final litmus test to see how efficiently we have trained the algorithm. Record all the results generated for the run for final observation before we start utilizing for detection.

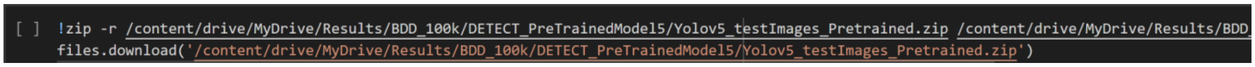


```
!python detect.py --source /content/data/bdd100k/videos_mp4 \
                  --weights /content/drive/MyDrive/Results/BDD_100k/Train/exp2/weights/best.pt
                  --data bdd100k.yaml --img 640 \
                  --project /content/drive/MyDrive/Results/BDD_100k/DETECT
                  --save-txt --save-conf --line-thickness 1
```

Figure 4.23 Detection test with weights generated for training and validation phases

4.1.4.10 Record best performing model

Once we are good with test results as well, download the model weights and store in Google Drive and also in archive.org which can be used for web application to download it directly from the web thereby reducing the project space.



```
[ ]  !zip -r /content/drive/MyDrive/Results/BDD_100k/DETECT_PreTrainedModel5/Yolov5_testImages_Pretrained.zip /content/drive/MyDrive/Results/BDD_
     files.download('/content/drive/MyDrive/Results/BDD_100k/DETECT_PreTrainedModel5/Yolov5_testImages_Pretrained.zip')
```

Figure 4.24 Example code to download weights file

4.1.5 Process tracking algorithm for videos

After detection algorithm is implemented, we must integrate the tracking algorithm which is DeepSORT and here are the steps to perform.

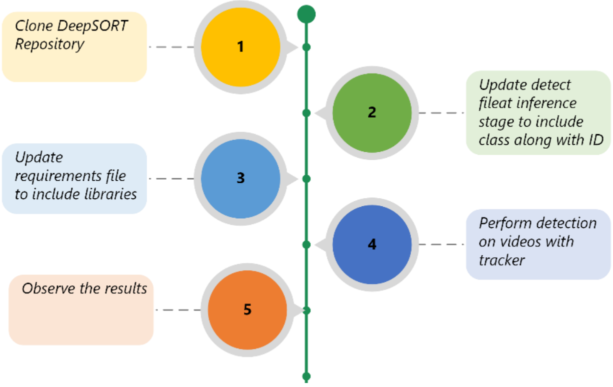
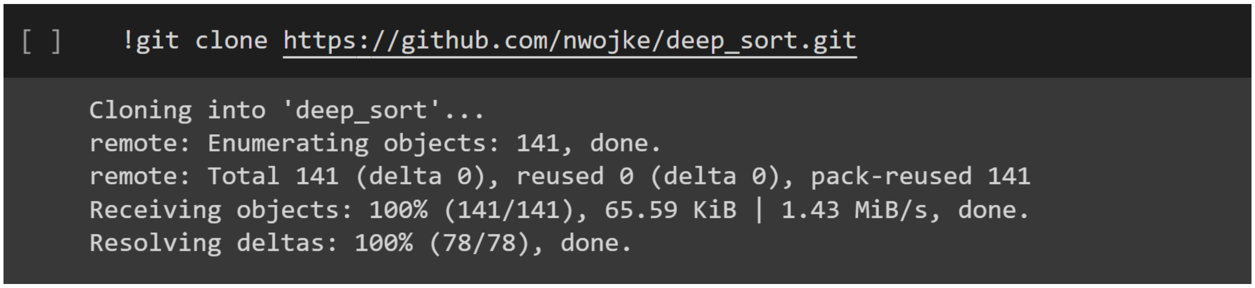

Clone DeepSORT Repository
1
2
Update detect fileat inference stage to include class along with ID
Update requirements file to include libraries
3
4
Perform detection on videos with tracker
Observe the results
5

Figure 4.25 Tracking algorithm integration

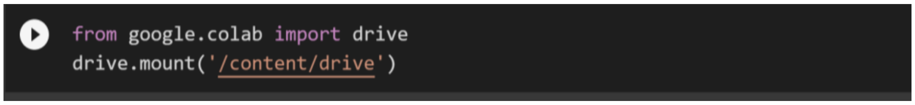
4.1.5.1 Clone Deep SORT repository

Clone the DeepSORT repository into Google Colab.



```
[ ]   !git clone https://github.com/nwojke/deep_sort.git

      Cloning into 'deep_sort'...
      remote: Enumerating objects: 141, done.
      remote: Total 141 (delta 0), reused 0 (delta 0), pack-reused 141
      Receiving objects: 100% (141/141), 65.59 KiB | 1.43 MiB/s, done.
      Resolving deltas: 100% (78/78), done.
```

Figure 4.26 Clone the repository

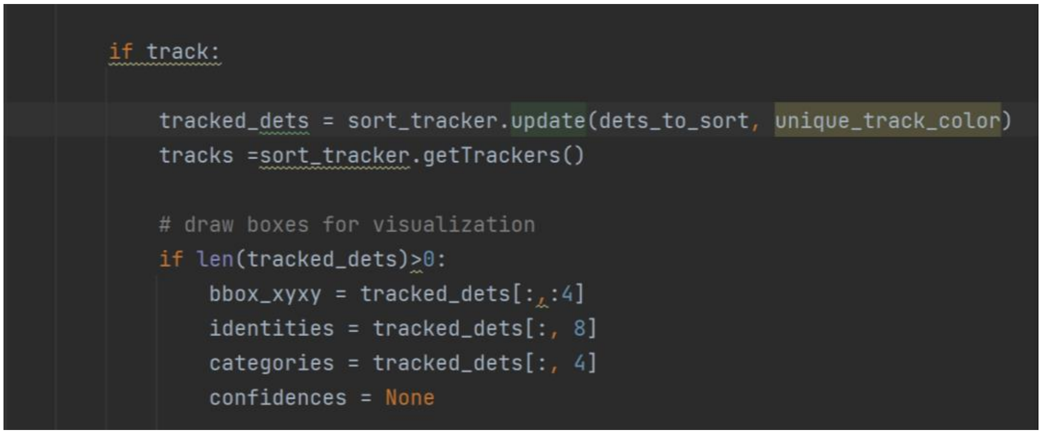


```
from google.colab import drive
drive.mount('/content/drive')
```

Figure 4.27 Mount the Google drive to load the content

4.1.5.2 Update detect file at inference stage to include class along with ID

Understand the detect file and sort file responsible to track the required classes within video. Include logic to have labels along with ID for observation purposes.

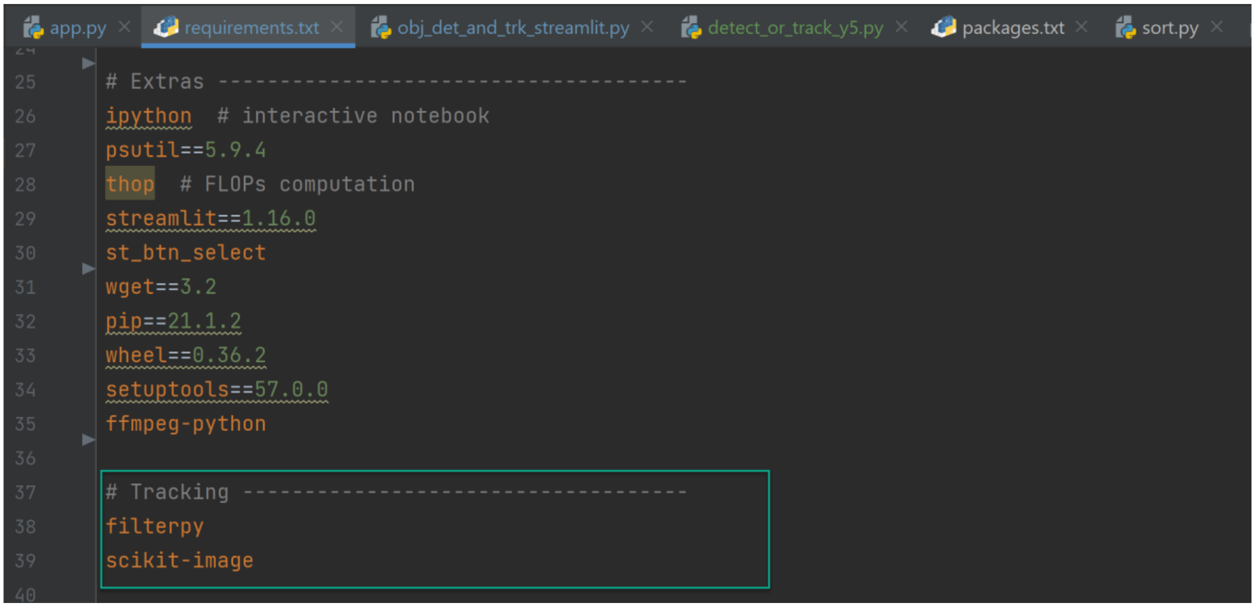


```
if track:

    tracked_dets = sort_tracker.update(dets_to_sort, unique_track_color)
    tracks =sort_tracker.getTrackers()

    # draw boxes for visualization
    if len(tracked_dets)>0:
        bbox_xyxy = tracked_dets[:,:4]
        identities = tracked_dets[:, 8]
        categories = tracked_dets[:, 4]
        confidences = None
```

Figure 4.28 Folder structure after cloning

4.1.5.3 Update requirements file to include libraries.

Update the requirement.txt file to include libraries needed for object tracking.



```
# Extras ------------------------------------
ipython  # interactive notebook
psutil==5.9.4
thop  # FLOPs computation
streamlit==1.16.0
st_btn_select
wget==3.2
pip==21.1.2
wheel==0.36.2
setuptools==57.0.0
ffmpeg-python

# Tracking -----------------------------------
filterpy
scikit-image
```

Figure 4.29 Updating tracking algorithm

4.1.5.4 Observe the results

Once we invoke detection algorithm , understand the different objects being recorded and some of the snapshots are provided in actual outcome 4.2.9

4.1.6 Build web application for object detection and tracking

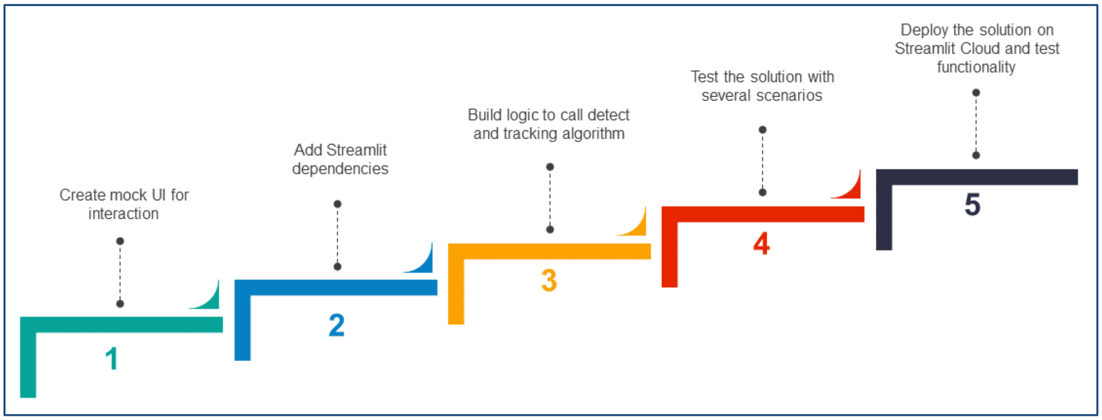

Create mock UI for interaction
1
Add Streamlit dependencies
2
Build logic to call detect and tracking algorithm
3
Test the solution with several scenarios
4
Deploy the solution on Streamlit Cloud and test functionality
5

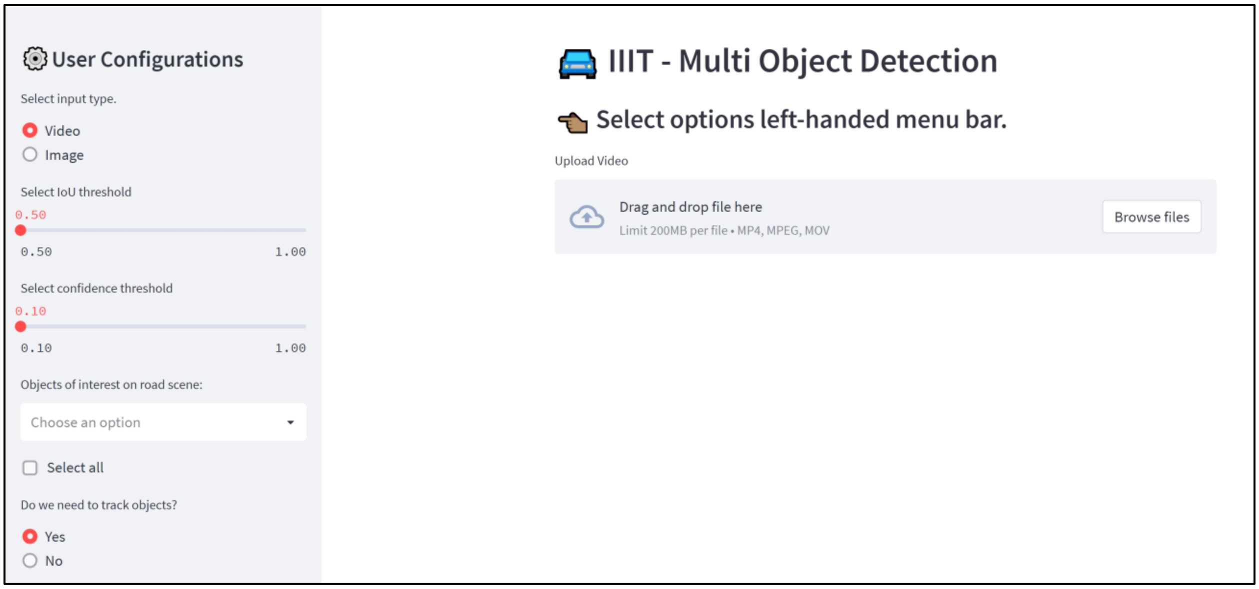
Figure 4.30 Building Streamlit application

4.1.6.1 Create mock UI for interaction

Understand the different parameters required for object detection and tracking algorithm. Here are few inputs we might think of:

Element	Type	Need for the element
Input Choice	Radio button (Video / Image)	Providing an option to perform required operation either on video or image
Select IoU threshold	Slider (values between 0.5 and 1.0)	As this is one of the important hyper parameter which determine the required object, we can provide threshold value for IoU.
Select confidence threshold	Slider (values between 0.1 and 1.0)	Confidence threshold is the maximum value that we can tolerate to determine if the object is of expected class after non-max suppression
Objects of interest on road scene*	Multi Select	If we need to specifically observe the objects of our interest, we can select the required list of objects . Possible values are 'traffic light', 'traffic sign', 'car', 'pedestrian', 'bus', 'truck', 'rider', 'bicycle', 'motorcycle', 'train', 'trailer'
Do we need to track objects? *	Radio Button with Yes /No	If we select tracking as Yes, then detection + DeepSORT logic will be invoked. If No, only YOLOv5 logic will be invoked.
Upload Image	File Uploader	If we choose the input choice as image , then we need to upload the image with supported formats like PNG,JPEG,JPG.
Upload Video*	File Uploader	If we choose the input choice as video , then we need to upload the image with supported formats like MP4,MOV,MPEG.

*Note: Options with * indicate that they are available for input choice = Video*



User Configurations

Select input type.

● Video
○ Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

Objects of interest on road scene:

Choose an option

□ Select all

Do we need to track objects?

● Yes
○ No

IIIT - Multi Object Detection

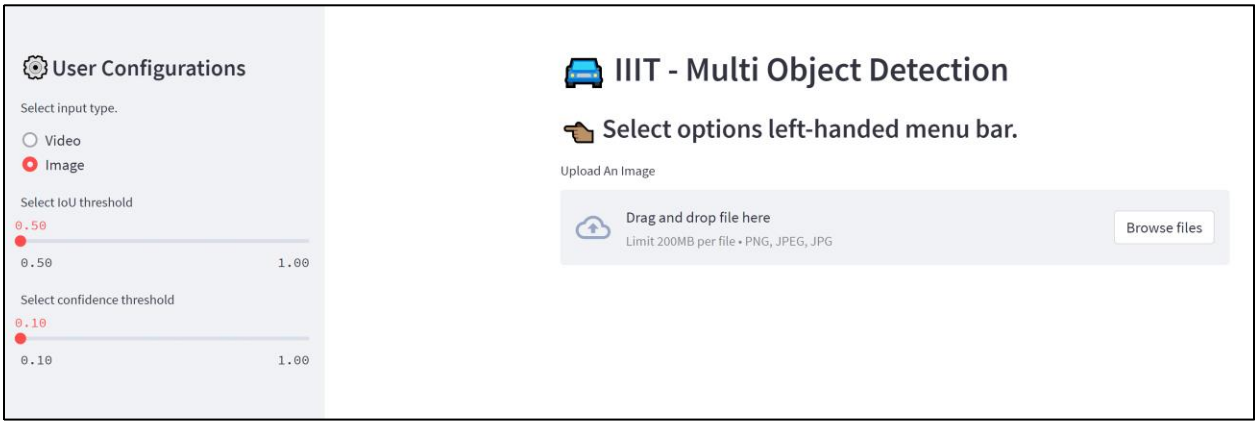
Select options left-handed menu bar.

Upload Video

Drag and drop file here

Limit 200MB per file • MP4, MPEG, MOV

Figure 4.31 Mock UI for video input



User Configurations

Select input type.

○ Video
● Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

IIIT - Multi Object Detection

Select options left-handed menu bar.

Upload An Image

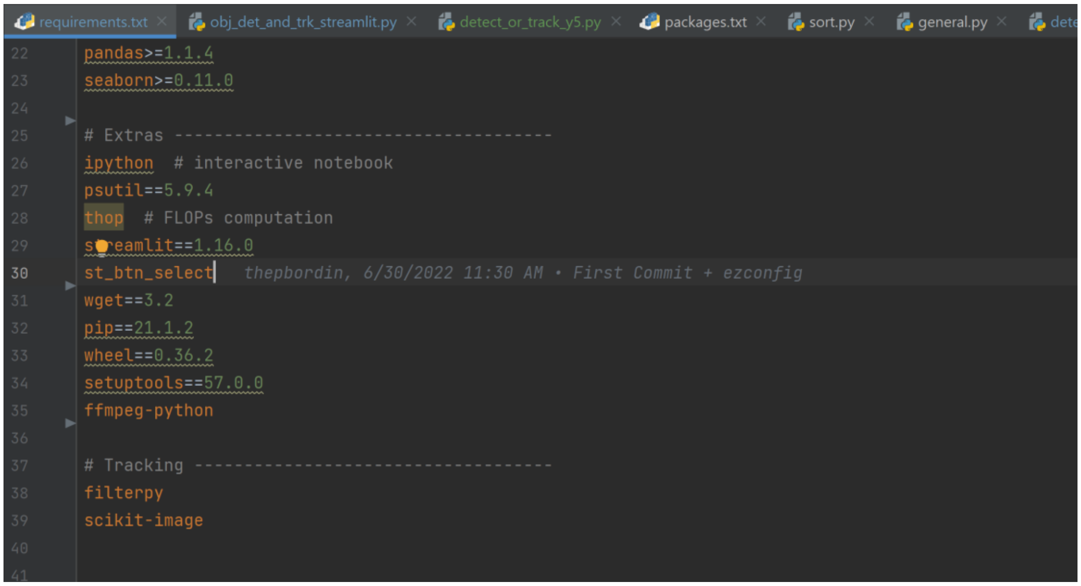
Drag and drop file here

Limit 200MB per file • PNG, JPEG, JPG

Figure 4.32 Mock UI for image input

4.1.6.2 Add streamlit dependencies

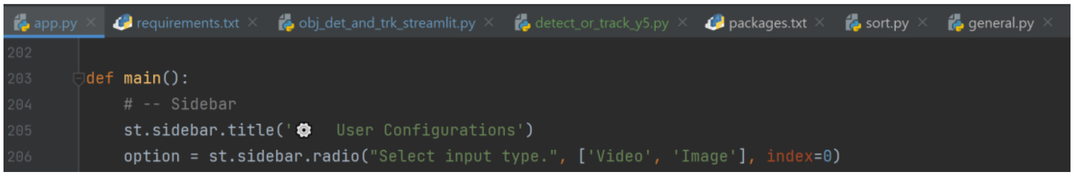
Add all the streamlit required dependencies in requirements.txt file and start running the essential commands in Python console to setup the essential environment.


requirements.txt
obj_det_and_trk_streamlit.py
detect_or_track_y5.py
packages.txt
sort.py
general.py
dete
thepbordin, 6/30/2022 11:30 AM • First Commit + ezconfig

```
pandas>=1.1.4
seaborn>=0.11.0

# Extras --------------------------------------
ipython  # interactive notebook
psutil==5.9.4
thop  # FLOPs computation
streamlit==1.16.0
st_btn_select
wget==3.2
pip==21.1.2
wheel==0.36.2
setuptools==57.0.0
ffmpeg-python

# Tracking ------------------------------------
filterpy
scikit-image

```

Figure 4.33 Requirements file for project

Start with sample streamlit lines as shown below and see if it work to ensure all the required libraries are installed before we code the solution as per mock UI mentioned above.

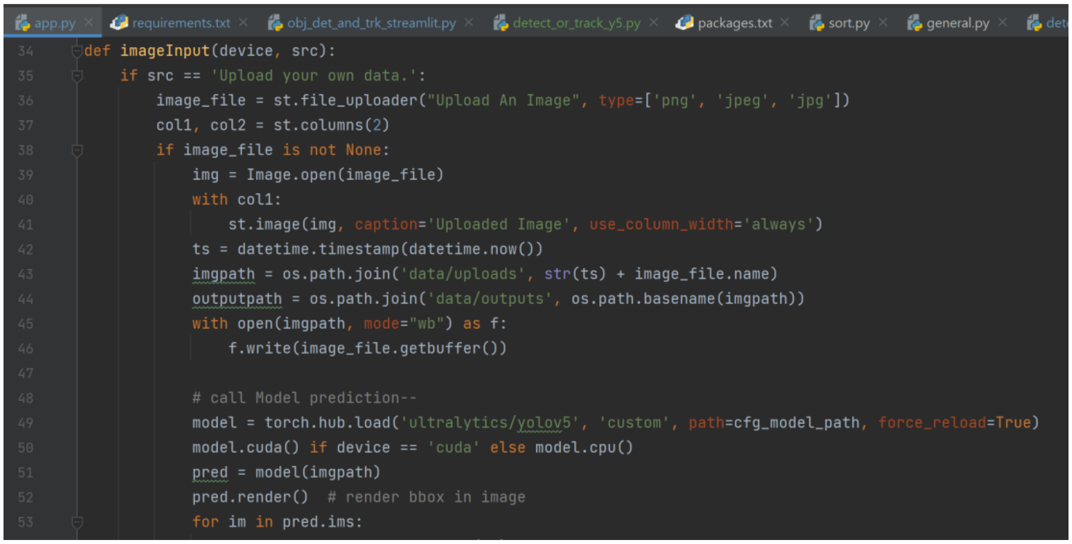


```

def main():
    # -- Sidebar
    st.sidebar.title('⚙  User Configurations')
    option = st.sidebar.radio("Select input type.", ['Video', 'Image'], index=0)
```

Figure 4.34 Sample streamlit code to test functionality

4.1.6.3 Build logic to detect objects for image

Code solution to invoke object detection of YOLOv5 with essential input image and ensure that resultant image gets generated after the model sends the output with highlighted bounding boxes with confidence score.

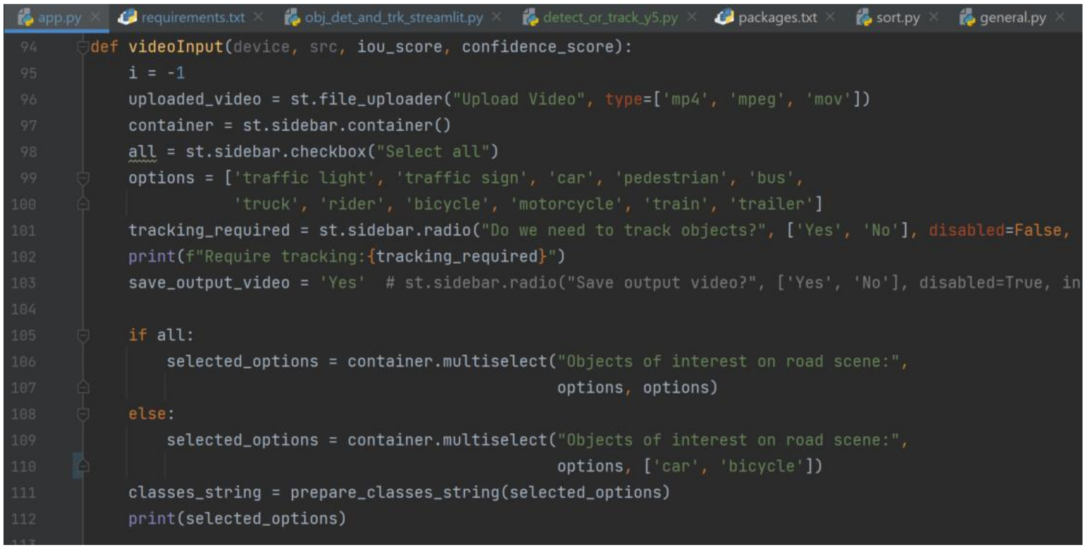


```python
def imageInput(device, src):
    if src == 'Upload your own data.':
        image_file = st.file_uploader("Upload An Image", type=['png', 'jpeg', 'jpg'])
        col1, col2 = st.columns(2)
        if image_file is not None:
            img = Image.open(image_file)
            with col1:
                st.image(img, caption='Uploaded Image', use_column_width='always')
            ts = datetime.timestamp(datetime.now())
            imgpath = os.path.join('data/uploads', str(ts) + image_file.name)
            outputpath = os.path.join('data/outputs', os.path.basename(imgpath))
            with open(imgpath, mode="wb") as f:
                f.write(image_file.getbuffer())

            # call Model prediction--
            model = torch.hub.load('ultralytics/yolov5', 'custom', path=cfg_model_path, force_reload=True)
            model.cuda() if device == 'cuda' else model.cpu()
            pred = model(imgpath)
            pred.render()  # render bbox in image
            for im in pred.ims:
```

Figure 4.35 Sample logic to implement object detection for image as input

4.1.6.4 Build logic to detect and track objects of interest for video

Code solution to invoke object detection of YOLOv5 with essential input video and ensure that resultant video gets generated after the model sends the output with highlighted bounding boxes with confidence score either with or without tracking.

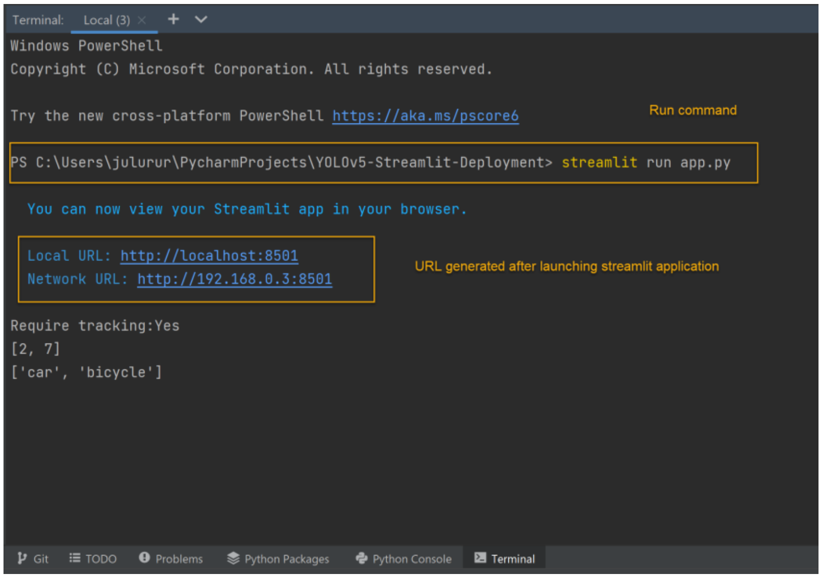


```python
def videoInput(device, src, iou_score, confidence_score):
    i = -1
    uploaded_video = st.file_uploader("Upload Video", type=['mp4', 'mpeg', 'mov'])
    container = st.sidebar.container()
    all = st.sidebar.checkbox("Select all")
    options = ['traffic light', 'traffic sign', 'car', 'pedestrian', 'bus',
               'truck', 'rider', 'bicycle', 'motorcycle', 'train', 'trailer']
    tracking_required = st.sidebar.radio("Do we need to track objects?", ['Yes', 'No'], disabled=False,
    print(f"Require tracking:{tracking_required}")
    save_output_video = 'Yes'  # st.sidebar.radio("Save output video?", ['Yes', 'No'], disabled=True, in

    if all:
        selected_options = container.multiselect("Objects of interest on road scene:",
                                                 options, options)
    else:
        selected_options = container.multiselect("Objects of interest on road scene:",
                                                 options, ['car', 'bicycle'])
    classes_string = prepare_classes_string(selected_options)
    print(selected_options)
```

Figure 4.36 Sample logic to implement object detection for videos with/ without tracking

4.1.6.5 Test the solution with input image and video and analyze the results

Start the streamlit application in console and see if its working as expected. As shown in actual outputs, we can perform combination of different inputs and parameters to see if application is working as per expectation.



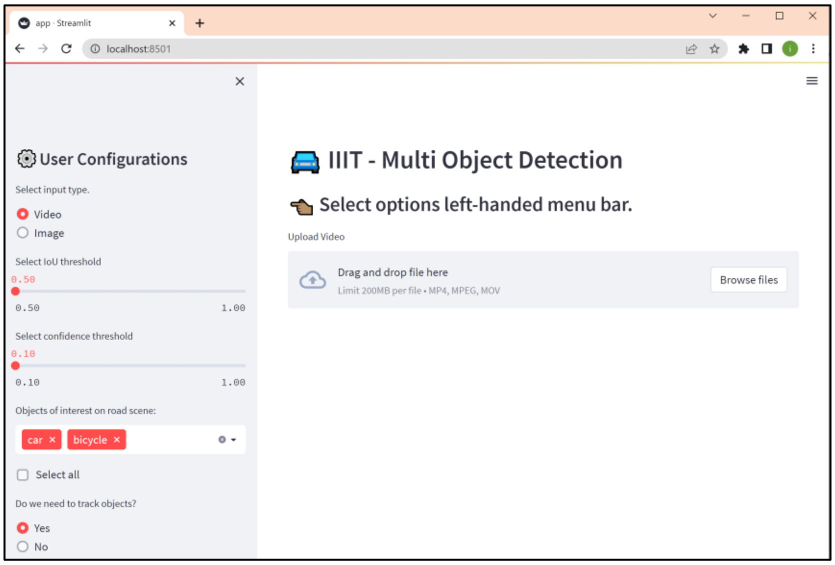
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\julurur\PycharmProjects\YOLOv5-Streamlit-Deployment> streamlit run app.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.0.3:8501

Require tracking:Yes
[2, 7]
['car', 'bicycle']
```

Run command

URL generated after launching streamlit application

Figure 4.37 Run command to invoke streamlit application

After launching the streamlit application, the browser looks something like below.



User Configurations

Select input type.

● Video
○ Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

Objects of interest on road scene:

car ✕ bicycle ✕

□ Select all

Do we need to track objects?

● Yes
○ No

IIIT - Multi Object Detection

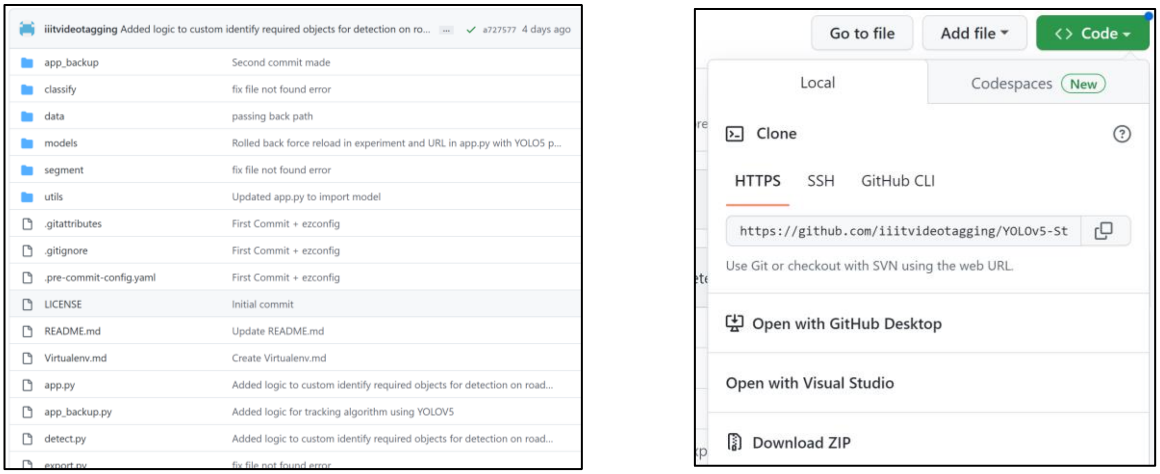
Select options left-handed menu bar.

Upload Video

Drag and drop file here
Limit 200MB per file • MP4, MPEG, MOV

Figure 4.38 Application running on localhost

Deploy the entire solution in Github and ensure all the required packages are pre-defined.



iiitvideotagging Added logic to custom identify required objects for detection on ro... ✓ a727577 4 days ago

app_backup	Second commit made
classify	fix file not found error
data	passing back path
models	Rolled back force reload in experiment and URL in app.py with YOLO5 p...
segment	fix file not found error
utils	Updated app.py to import model
.gitattributes	First Commit + ezconfig
.gitignore	First Commit + ezconfig
.pre-commit-config.yaml	First Commit + ezconfig
LICENSE	Initial commit
README.md	Update README.md
Virtualenv.md	Create Virtualenv.md
app.py	Added logic to custom identify required objects for detection on road...
app_backup.py	Added logic for tracking algorithm using YOLOV5
detect.py	Added logic to custom identify required objects for detection on road...
export.py	fix file not found error

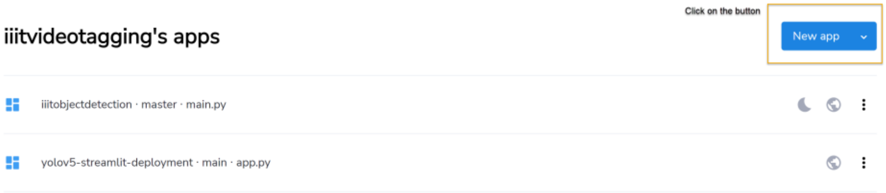
```
https://github.com/iiitvideotagging/YOLOv5-St
```

Use Git or checkout with SVN using the web URL.

Figure 4.39 GIT Repository for project

4.1.6.6 Deploy the solution on streamlit cloud

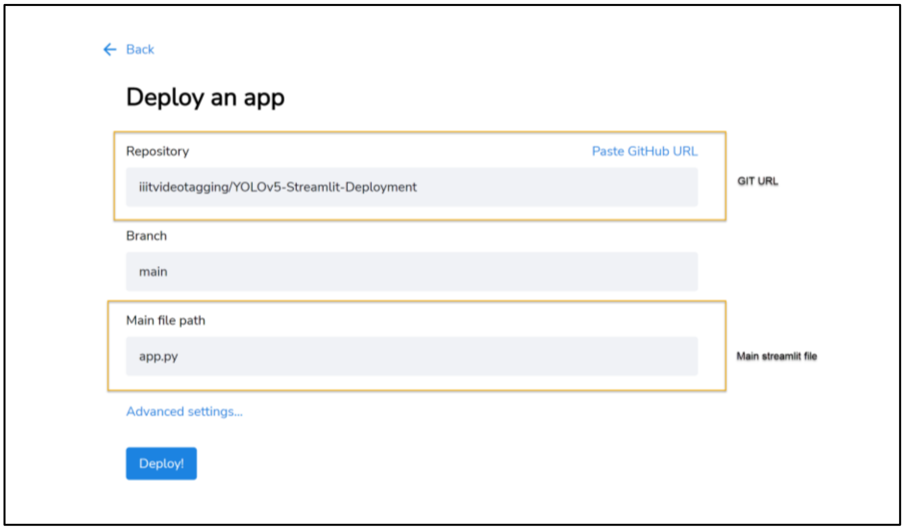
Create an account in Streamlit Cloud where we can create 3 applications and add a new application by pointing the repository we created and point to main branch with app.py being the starting file to invoke.



iiitobjectdetection · master · main.py

yolov5-streamlit-deployment · main · app.py

Figure 4.40 Add a new app in Streamlit cloud



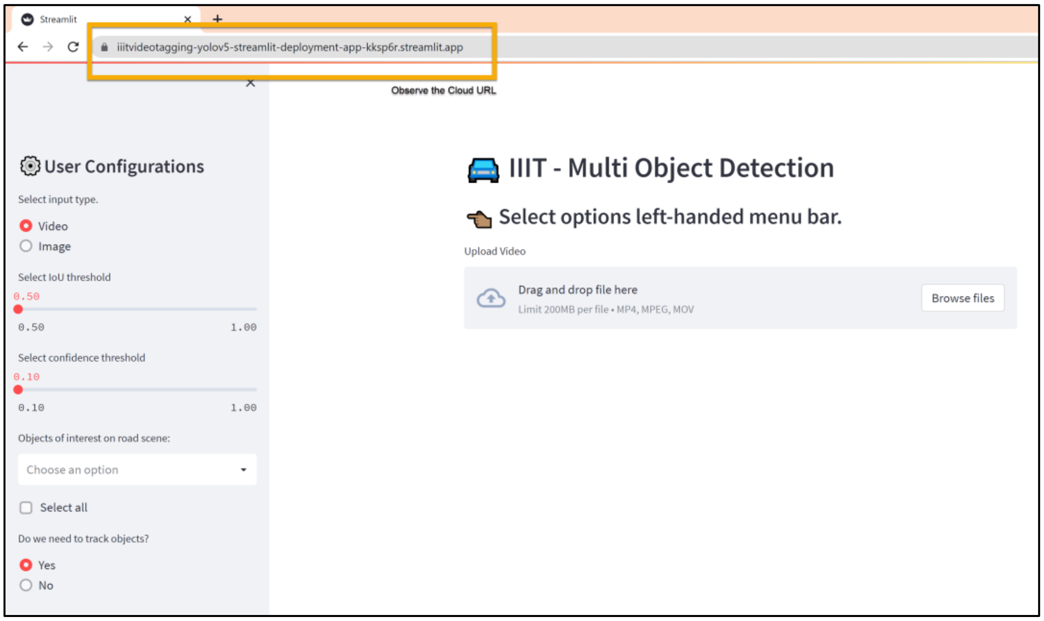
Repository: iiitvideotagging/YOLOv5-Streamlit-Deployment

Branch: main

Main file path: app.py

Figure 4.41 Deploy application options

Then we click on deploy button and we can see that the application is deployed into cloud and can be accessed by anyone with the required URL.



Streamlit

iiitvideotagging-yolov5-streamlit-deployment-app-kksp6r.streamlit.app

Observe the Cloud URL

User Configurations

Select input type.

● Video
○ Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

Objects of interest on road scene:

Choose an option

□ Select all

Do we need to track objects?

● Yes
○ No

IIIT - Multi Object Detection

Select options left-handed menu bar.

Upload Video

Drag and drop file here

Limit 200MB per file • MP4, MPEG, MOV

Browse files

Figure 4.42 Deployed app on cloud

4.2 Actual Outcome

4.2.1 Training metrics – YOLOv5

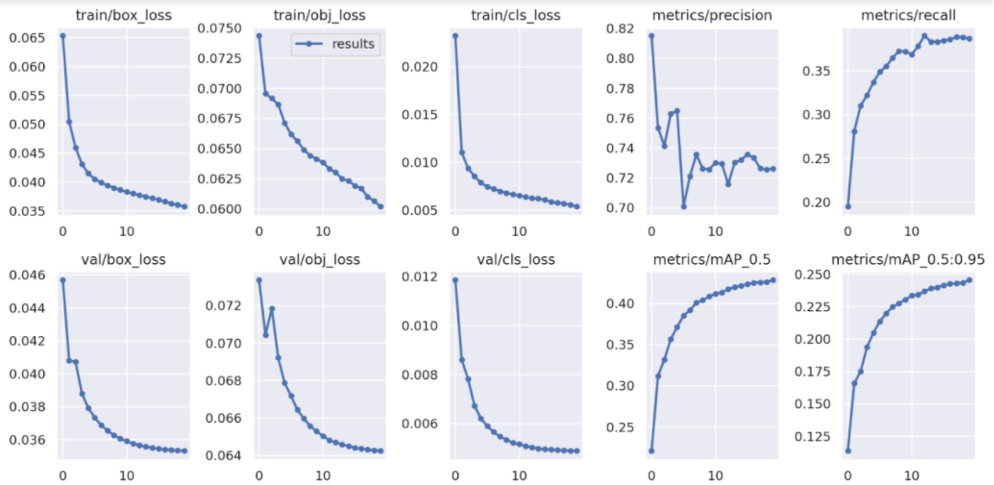


Figure 4.43 Precision, Recall & other metrics

Here is the high-level definition of terms mentioned in above screen capture.

- **box_loss**

    - Bounding box regression loss (Mean Squared Error).
- **obj_loss**
    - The confidence of object presence is the objectness loss.
- **cls_loss**
    - The classification loss (Cross Entropy).
- **Precision**
    - Measures how much of the bbox predictions are correct (True positives / (True positives + False positives))
- **Recall**
    - Measures how much of the true bbox were correctly predicted (True positives / (True positives + False negatives)).
- **mAP_0.5**
    - mean Average Precision (mAP) at IoU (Intersection over Union) threshold of 0.5.
- **mAP_0.5:0.95**
    - Average mAP over different IoU thresholds, ranging from 0.5 to 0.95.

4.2.2 Confusion matrix – Training - YOLOv5

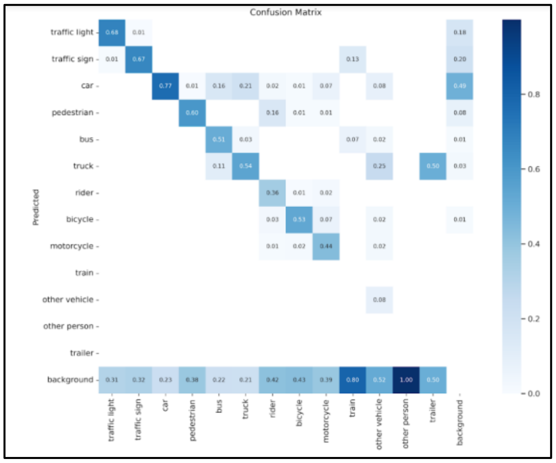


Figure 4.44 Confusion matrix - YOLOv5

4.2.3 Validation metrics – YOLOv5

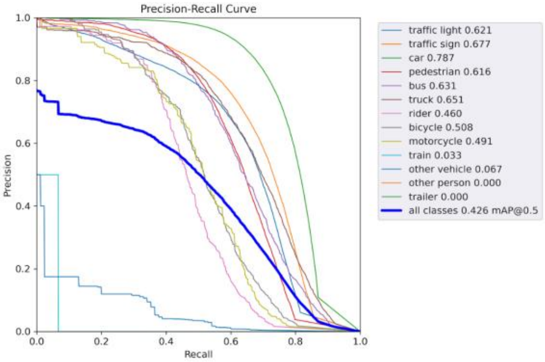


Figure 4.45 Precision - Recall Curve for different classes

4.2.4 Confusion matrix – Validation - YOLOv5

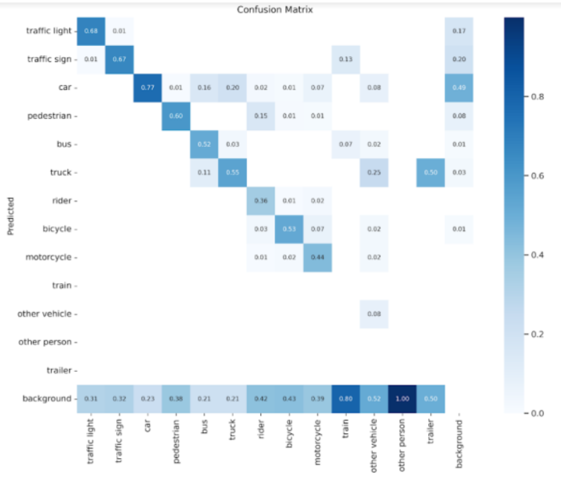
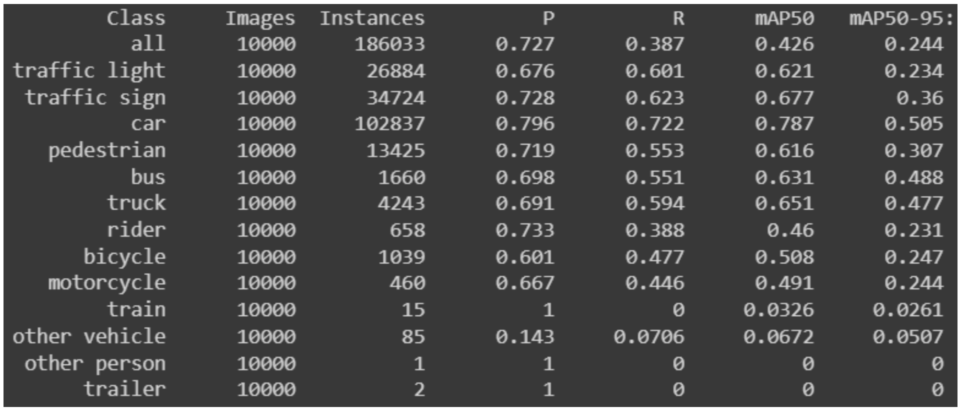


Figure 4.46 Confusion matrix - Validation - YOLOv5

4.2.5 Results – Test – YOLOv5



```
         Class     Images  Instances          P          R      mAP50   mAP50-95:
           all      10000     186033      0.727      0.387      0.426      0.244
 traffic light      10000      26884      0.676      0.601      0.621      0.234
  traffic sign      10000      34724      0.728      0.623      0.677       0.36
           car      10000     102837      0.796      0.722      0.787      0.505
    pedestrian      10000      13425      0.719      0.553      0.616      0.307
           bus      10000       1660      0.698      0.551      0.631      0.488
         truck      10000       4243      0.691      0.594      0.651      0.477
         rider      10000        658      0.733      0.388       0.46      0.231
       bicycle      10000       1039      0.601      0.477      0.508      0.247
    motorcycle      10000        460      0.667      0.446      0.491      0.244
         train      10000         15          1          0     0.0326     0.0261
 other vehicle      10000         85      0.143     0.0706     0.0672     0.0507
  other person      10000          1          1          0          0          0
       trailer      10000          2          1          0          0          0
```

Figure 4.47 Results after test YOLOv5

4.2.6 Sample Detected Images

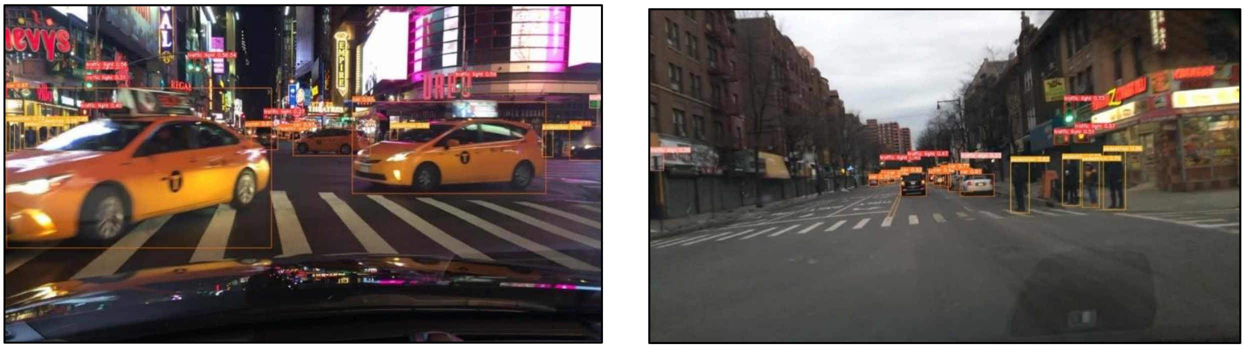


Figure 4.48 Detected Image Results

4.2.7 Sample detected videos

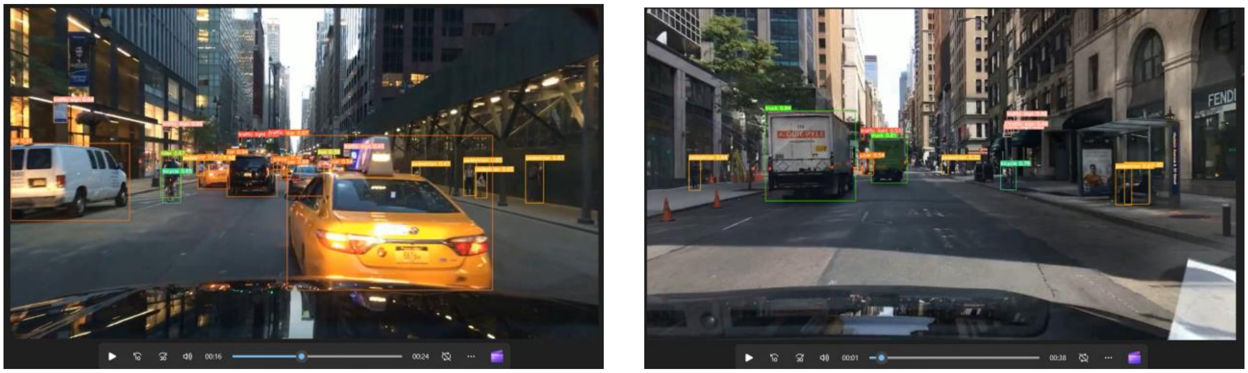


Figure 4.49 Detected Video Results

4.2.8 Additional analysis made during development

4.2.8.1 Comparison of pre-trained COCO vs YOLOv5

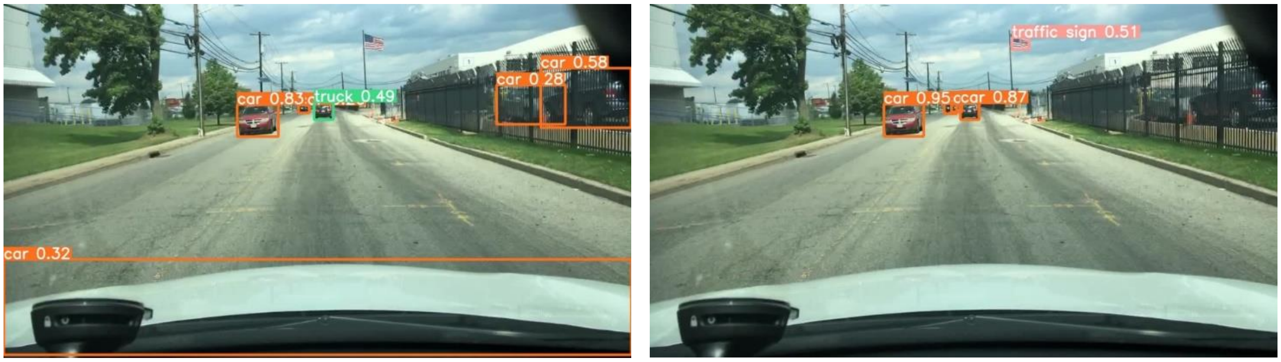


Figure 4.50 Pre-trained COCO vs YOLOV5

4.2.8.2 Comparison of YOLOv5 vs YOLOv7

4.2.8.2.1 Training results – YOLOv7

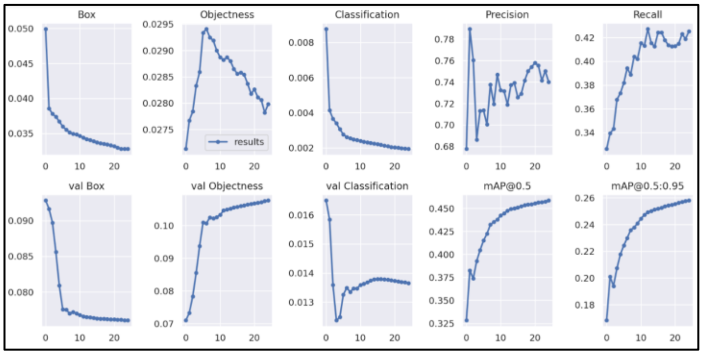


Figure 4.51 Training Metrics - YOLOv7

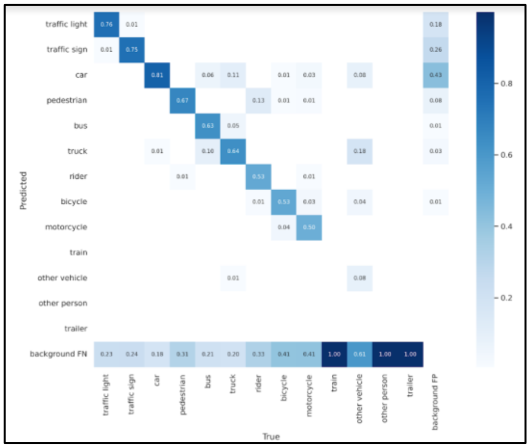


Figure 4.52 Confusion matrix - YOLOv7

4.2.8.2.2 Validation metrics – YOLOv7

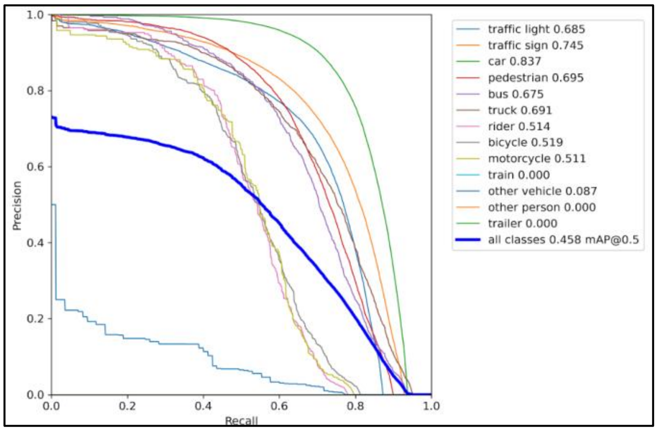


Figure 4.53 Precision Recall Curve - YOLOv7

4.2.8.2.3 Confusion matrix – Validation – YOLOv7

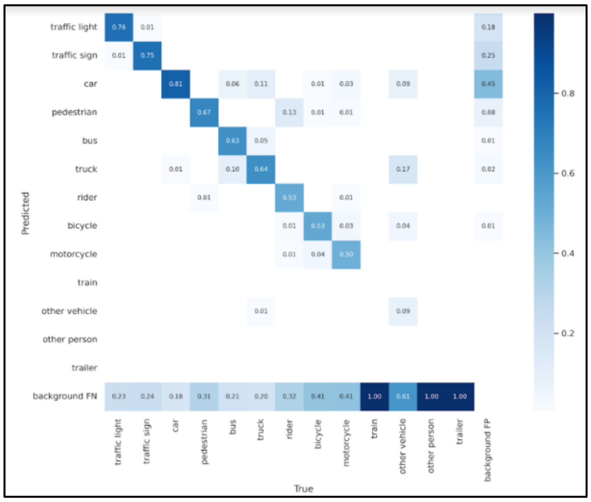
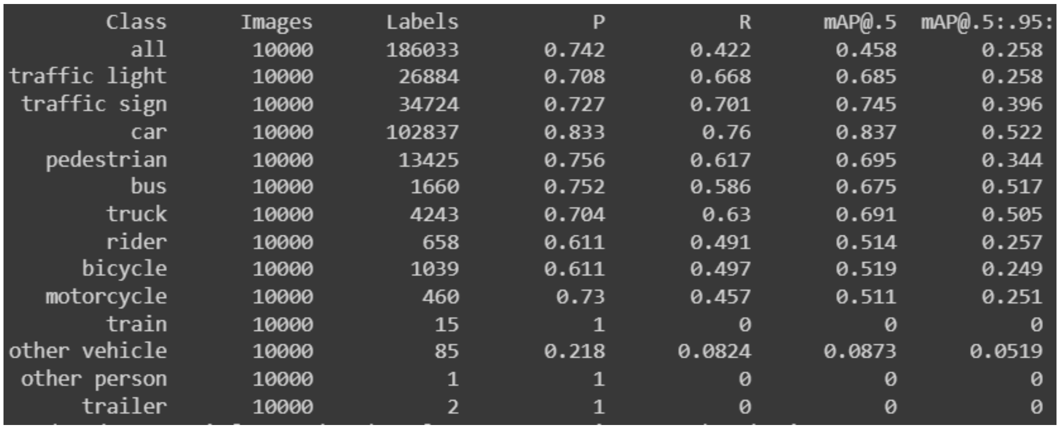


Figure 4.54 Confusion Matrix - YOLOv7

4.2.8.2.4 Validation Metrics – YOLOV7



```
         Class      Images      Labels           P           R      mAP@.5  mAP@.5:.95:
           all       10000      186033       0.742       0.422       0.458       0.258
 traffic light       10000       26884       0.708       0.668       0.685       0.258
  traffic sign       10000       34724       0.727       0.701       0.745       0.396
           car       10000      102837       0.833        0.76       0.837       0.522
    pedestrian       10000       13425       0.756       0.617       0.695       0.344
           bus       10000        1660       0.752       0.586       0.675       0.517
         truck       10000        4243       0.704        0.63       0.691       0.505
         rider       10000         658       0.611       0.491       0.514       0.257
       bicycle       10000        1039       0.611       0.497       0.519       0.249
    motorcycle       10000         460        0.73       0.457       0.511       0.251
         train       10000          15           1           0           0           0
 other vehicle       10000          85       0.218      0.0824      0.0873      0.0519
  other person       10000           1           1           0           0           0
       trailer       10000           2           1           0           0           0
```

Figure 4.55 Validation Results - YOLOV7

Note: Though the Yolo-v7 looks better interms of map values and confusion matrix compared to v5. Its to be noted that the v7 was trained with 5 epochs more than the v5 model.

4.2.8.2.5 Sample YOLOv7 detected images

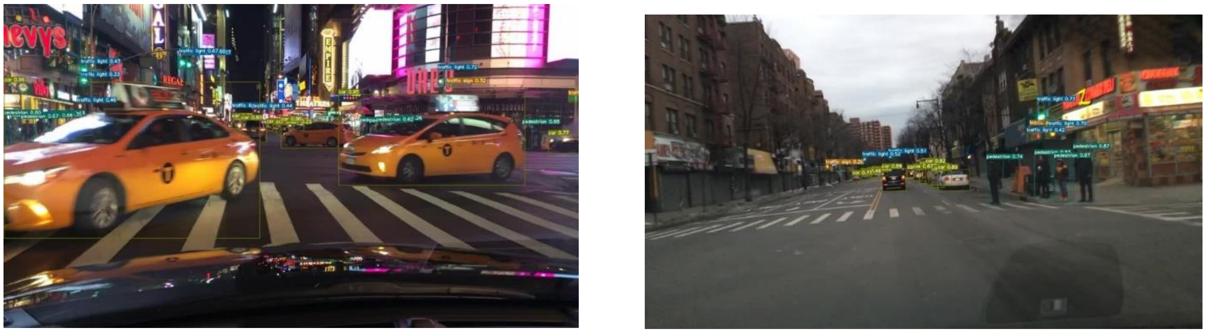


Figure 4.56 Sample YOLOv7 output images

4.2.8.2.6 Sample YOLOv7 detected videos

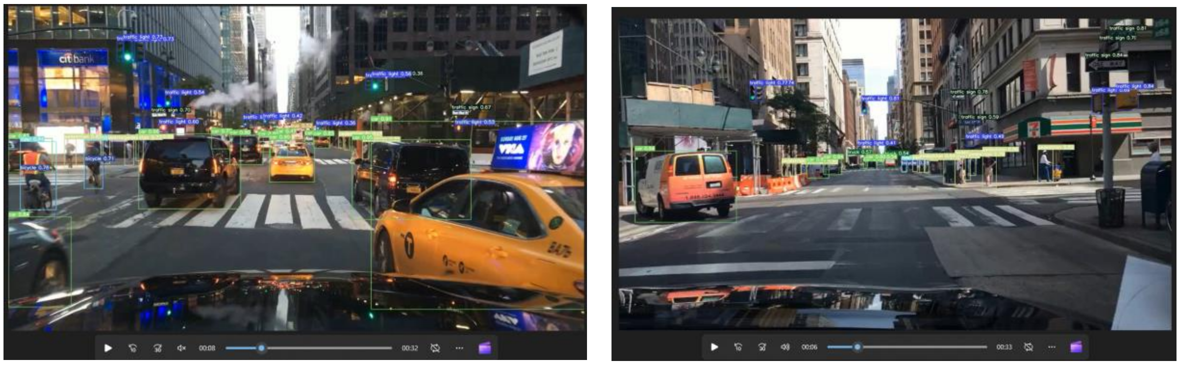


Figure 4.57 Sample YOLOv7 output videos

4.2.8.2.7 YOLOv5 vs YOLOv7 Image comparison

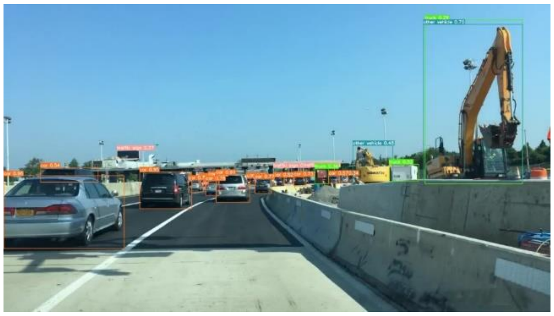


Figure 4.58 YOLOv5 Image

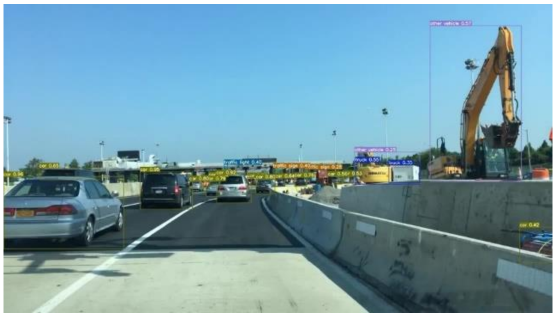


Figure 4.59 YOLOv7 Image

4.2.9 Tracking Objects

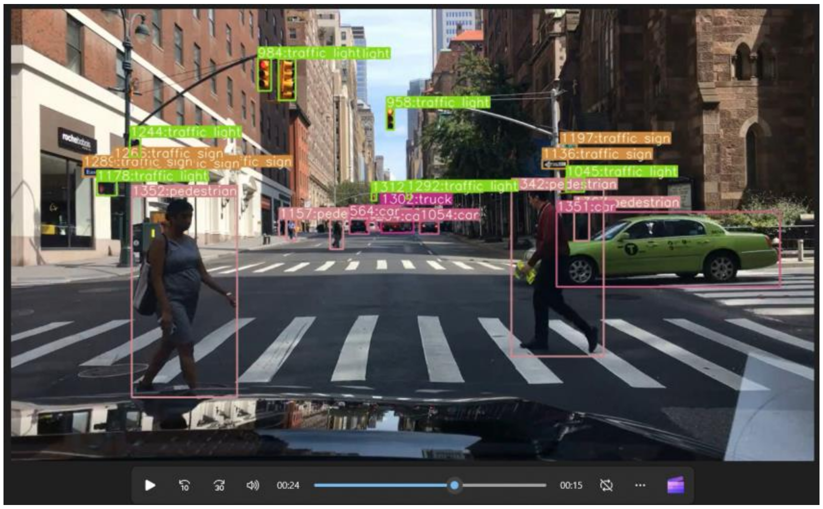


Figure 4.60 Tracking objects sample 1

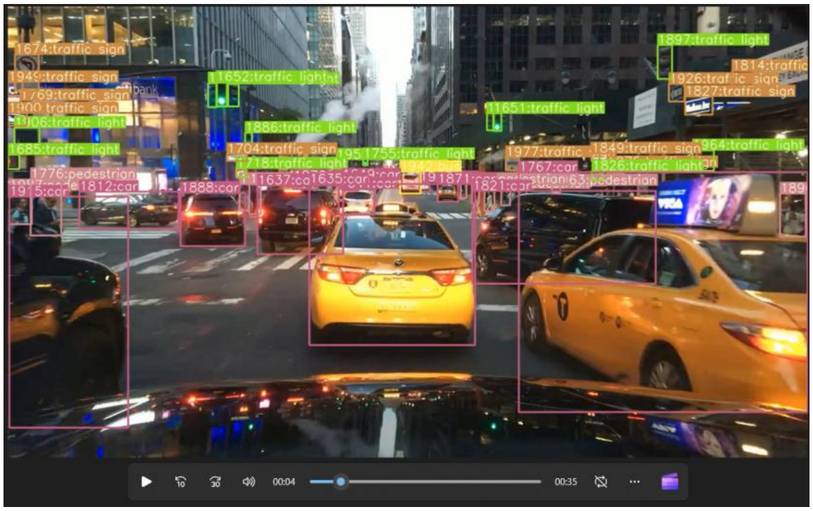
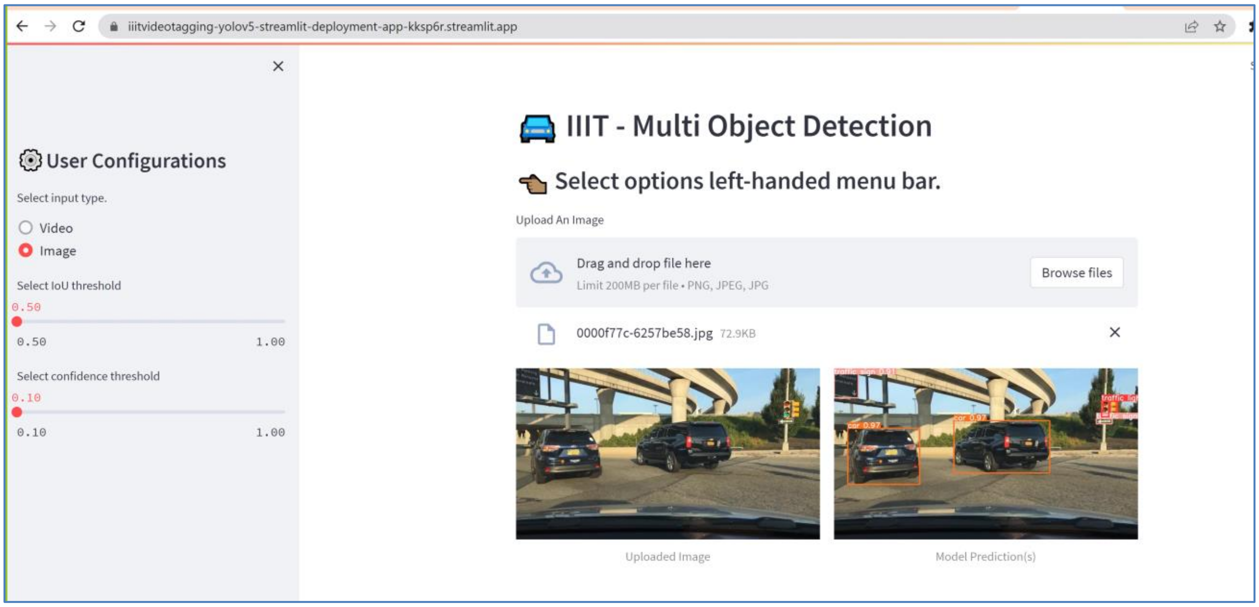


Figure 4.61 Tracking Object Sample 2

4.2.10 Application

4.2.10.1 Image detection



iiitvideotagging-yolov5-streamlit-deployment-app-kksp6r.streamlit.app

User Configurations

Select input type.

○ Video
● Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

IIIT - Multi Object Detection

Select options left-handed menu bar.

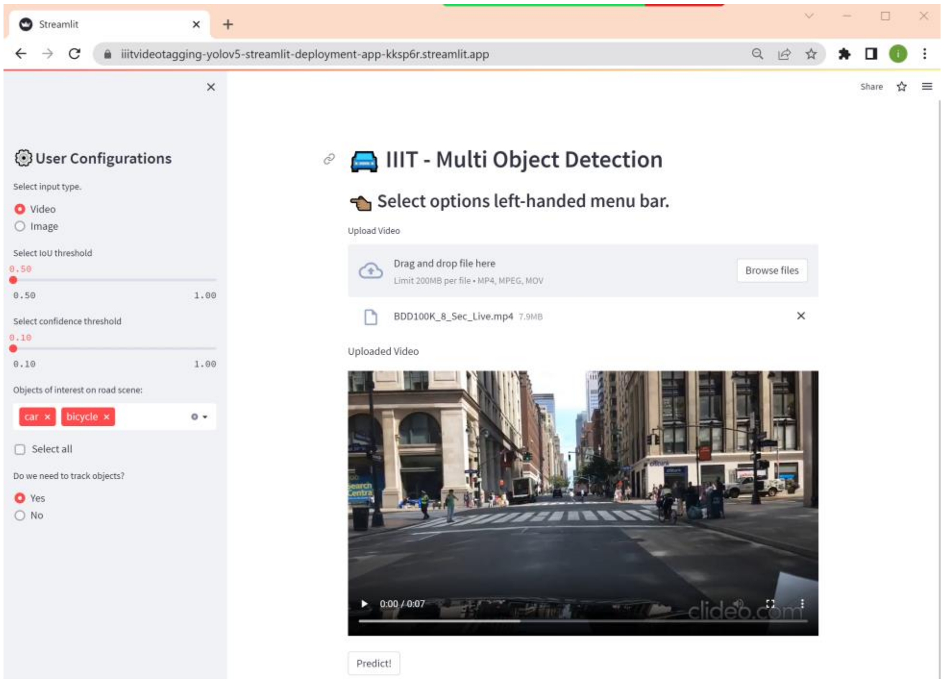
Upload An Image

Drag and drop file here
Limit 200MB per file • PNG, JPEG, JPG

0000f77c-6257be58.jpg 72.9KB

Figure 4.62 Model prediction for image

4.2.10.2 Video detection – Upload video



iiitvideotagging-yolov5-streamlit-deployment-app-kksp6r.streamlit.app

User Configurations

Select input type.

● Video
○ Image

Select IoU threshold

0.50

0.50 1.00

Select confidence threshold

0.10

0.10 1.00

Objects of interest on road scene:

car × bicycle ×

□ Select all

Do we need to track objects?

● Yes
○ No

IIIT - Multi Object Detection

Select options left-handed menu bar.

Upload Video

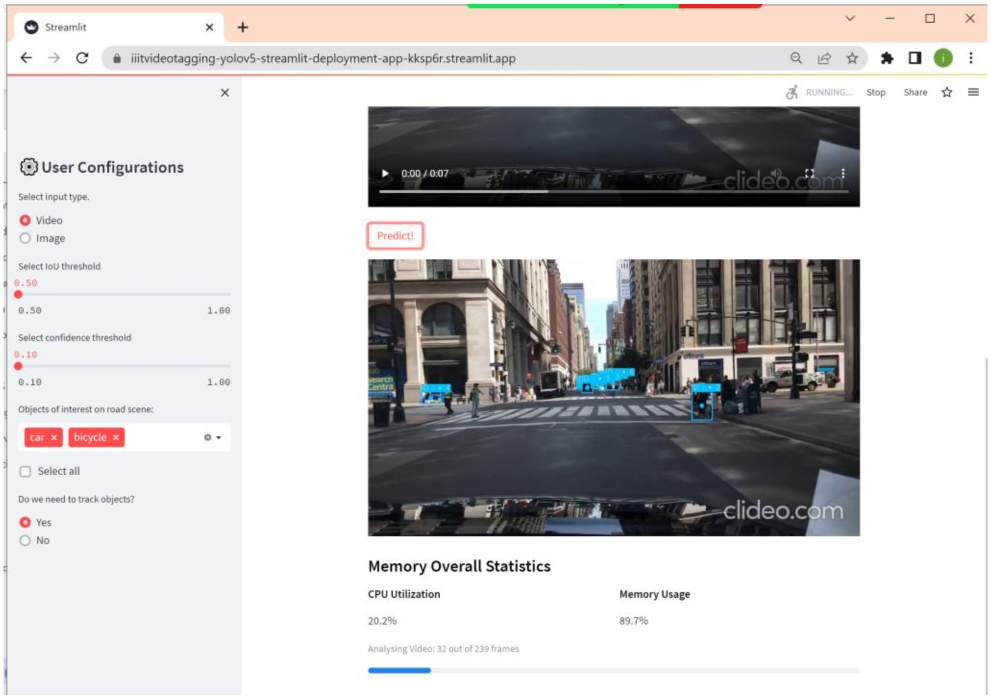
Drag and drop file here
Limit 200MB per file • MP4, MPEG, MOV

BDD100K_8_Sec_Live.mp4 7.9MB

Uploaded Video

Figure 4.63 After video upload

4.2.10.3 Video detection – Analyzing video with tracking


Streamlit
iiitvideotagging-yolov5-streamlit-deployment-app-kksp6r.streamlit.app
RUNNING...
Stop
Share
0:00 / 0:07
clideo.com
Predict!
clideo.com

User Configurations

Select input type.

● Video
○ Image

Select IoU threshold 0.50 (0.50 – 1.00)

Select confidence threshold 0.10 (0.10 – 1.00)

Objects of interest on road scene: car, bicycle

□ Select all

Do we need to track objects?

● Yes
○ No

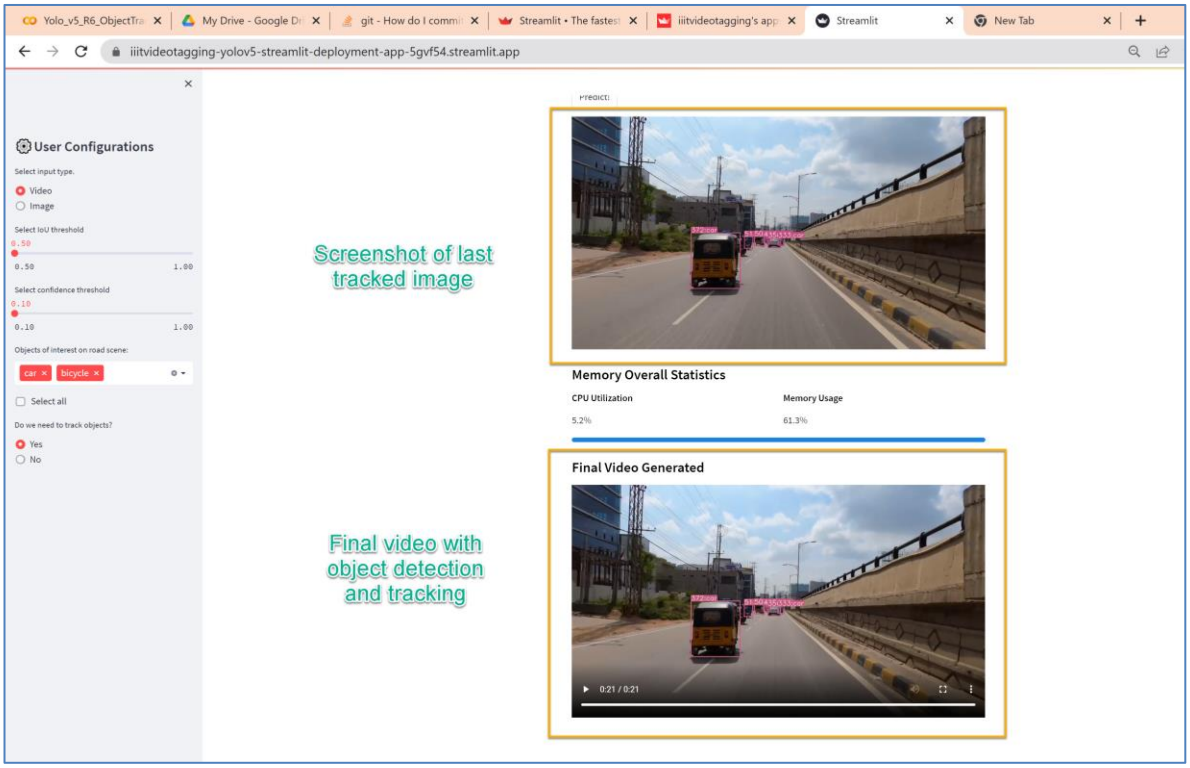
Memory Overall Statistics

CPU Utilization	Memory Usage
20.2%	89.7%

Analysing Video: 32 out of 239 frames

Figure 4.64 Analyzing video

4.2.10.4 Video detection – Final model prediction + tracker


Yolo_v5_R6_ObjectTra
My Drive - Google Dr
git - How do I commi
Streamlit • The fastest
iiitvideotagging's app
Streamlit
New Tab
iiitvideotagging-yolov5-streamlit-deployment-app-5gvf54.streamlit.app
Predict!
Screenshot of last tracked image
Final video with object detection and tracking
0:21 / 0:21

User Configurations

Select input type.

● Video
○ Image

Select IoU threshold 0.50 (0.50 – 1.00)

Select confidence threshold 0.10 (0.10 – 1.00)

Objects of interest on road scene: car, bicycle

□ Select all

Do we need to track objects?

● Yes
○ No

Memory Overall Statistics

CPU Utilization	Memory Usage
5.2%	61.3%

Final Video Generated

Figure 4.65 Final Model prediction + tracker

5 Challenges

1. **Preparation of BDD100K data**
   a. Missing labels for some of the images of BDD100k data.
   b. Downloading of BDD100k videos were time consuming.
   c. Tracking the objects from the BDD100K videos was time taking process and getting the accurate results was challenging task. To resolve the issues, we discussed it within the group and resolved the issues.
   d. Storage of models file to load them onto the cloud at run time.

2. **Resource constraint – GPU**:
   a. While training the model we faced disconnection of runtime in colab.

3. **Streamlit application development**:
   a. Understanding cloud constraints
   b. Sharing solution to team members and installation in their respective systems was a tedious task considering the packages / dependencies involved.
   c. Additionally, understanding prerequisite steps for running solution on cloud without any failure.
   d. Understanding installation of packages in streamlit cloud.
   e. Understanding video CODEC formats compatible with browsers.

6 Applicability in the real world

The application developed could be used as a prototype in advanced driver assistance systems, intelligent healthcare monitoring, autonomous vehicles, robot vision and smart traffic applications. The application can be extended to some of the areas:

- **Autonomous Driving:** Self-driving cars depend on object detection to recognize pedestrians, traffic signs, other vehicles, and more. For example, Tesla’s Autopilot AI heavily utilizes object detection to perceive environmental and surrounding threats such as oncoming vehicles or obstacles.
- **Animal detection in Agriculture:** Object detection is used in agriculture for tasks such as counting, animal monitoring, and evaluation of the quality of agricultural products. Damaged produce can be detected while it is in processing using machine learning algorithms.
- **People detection in Security:** A wide range of security applications in video surveillance are based on object detection, for example, to detect people in restricted or dangerous areas, suicide prevention, or automating inspection tasks in remote locations with computer vision.
- **Object detection in Retail:** Strategically placed people counting systems throughout multiple retail stores are used to gather information about how customers spend their time and customer footfall. AI-based customer analysis to detect and track customers with cameras helps to gain an understanding of customer interaction and customer experience, optimize the store layout, and make operations more efficient. A popular use case is the detection of queues to reduce waiting time in retail stores.
- **Vehicle detection with AI in Transportation:** Object recognition is used to detect and count vehicles for traffic analysis or to detect cars that stop in dangerous areas, for example, on crossroads or highways.
- **Medical feature detection in Healthcare:** Object detection has allowed for many breakthroughs in the medical community. Because medical diagnostics rely heavily on the study of images, scans, and photographs, object detection involving CT and MRI scans has become extremely useful for diagnosing diseases, for example with ML algorithms for tumor detection.
- **Detecting anomaly:** Another useful application of object detection is definitely spotting an anomaly and it has industry specific usages. For instance, in the field of agriculture object detection helps in identifying infected crops and thereby helps the farmers take measures accordingly. It could also help identify skin problems in healthcare. In the manufacturing industry the object detection technique can help in detecting problematic parts really fast and thereby allow the company to take the right step.
- **Counting the Crowd:** Crowd Counting or People counting is another significant application of object detection.

7 References

Reference	Year	Link
[1] Object Detection with Deep Learning: A Review	2019	https://arxiv.org/pdf/1807.05511.pdf
[2] Rich feature hierarchies for accurate object detection and semantic segmentation Tech report (v5)	2014	https://arxiv.org/pdf/1311.2524.pdf
[3] Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition	2015	https://arxiv.org/pdf/1406.4729v4.pdf
[4] FAST R-CNN	2015	https://arxiv.org/pdf/1504.08083.pdf
[5] Faster R-CNN	2016	https://arxiv.org/pdf/1506.01497.pdf
[6] YOLO	2016	https://arxiv.org/pdf/1506.02640v5.pdf
[7] Multiple Object Trackers in OpenCV: A Benchmark	2021	https://arxiv.org/pdf/2110.05102.pdf
[8] Multiple Object Tracking: A Literature Review	2022	https://arxiv.org/pdf/1409.7618.pdf