POLITECNICO
MILANO 1863

Image Analysis and Computer Vision Project

Ali Noshad, ali.noshad@mail.polimi.it, 101108

January 23, 2024

Visual analysis of moving vehicles Tracking, and counting

Prof. Vincenzo Caglioti

Contents

1 Introduction

1.1 Objective

The project's main goal is to create and put into place a reliable monitoring system for the visual analysis of moving automobiles. State of arts models and techniques are used in this system, which focuses on real-time vehicle tracking, and counting.

- The project involves fine-tuning a pre-trained YoLoV8 model using an annotated image dataset provided by the Roboflow team. This integration aims to enhance the efficiency and accuracy of object detection, allowing for the tracking and counting of vehicles entering and exiting highways and squares under various conditions, including day, night, and peak traffic hours.
- The implementation relies on core methods from ByteTrack, supervision functionalities, and the open-cv library. These components collectively contribute to the successful development of the monitoring system, providing essential functionalities for image and video processing.

The subsequent sections of this report will cover the why we need such system, methodology, data collection and preparation, model fun-tuning, object tracking, object counting, and finally a conclusion.

1.2 Why we need this system?

Technology and transportation are developing at a rapid pace, making it essential to have effective and sophisticated systems to monitor and control moving vehicles. The growing number of vehicles on the road means that traffic management, safety, and urban planning are greatly impacted by our understanding of their dynamics and behaviors. The goal of this project, "Visual Analysis of Moving Vehicles," is to make a significant contribution to this important field by using state-of-the-art computer science methods to glean insightful information from the real-time visual data that is collected.

The three main components of the project's fundamental functionality are tracking, and counting. The counting module ensures an accurate portrayal of vehicular density by using computer vision techniques to identify individual vehicles inside a given frame. Concurrently, the tracking system makes use of advanced methods to track the motion of cars over a series of frames, generating a consistent path for every object.

2 Methodology

2.1 Data collection

This part of the project is consist of the two different phase, because in the phase one we prepared a dataset of images along with their annotations which also us to train and fine-tuning a pre-trained YoLoV8 and the second phase was to prepare a set of videos with different characteristic which enables us to check the monitoring system in different conditions.

2.1.1 Images and annotations dataset

Before we start we collecting the images and labeling them we first need to select our model because depending on the model our data collection procedure could be different could be different. So, based on this we decided to create a procedure which allow us to create a dataset for every situation that we have as input to our monitoring system. To put it more clearly, we know that YoloV8 is trained extensively on large dataset which includes a wide variety of the objects, so, we don't need to create a very large dataset for fine-tune the model. Since, we are working with videos in different conditions we need to collect a dataset that can optimize the model for this certain situations. Based on this, we used the video it self to create images. We know that every video is made up by set of frames, based on this assumption we converted and saved every frame of the video as an image. Now that we have a set of images which highly related to the conditions that we want to test the monitoring system. Then, after this we needed to create the annotations. In this part we needed to select and save some coordinates on the image which create a bounding box describing the each object in each images, these coordinates are the labels that we want our model to predict. For creating these annotations there

several possible ways and websites, in this approach we used the Roboflow interface which allow us to upload the image dataset, and provides an easy tool to draw bounding box on the image. Please note that, we have also implemented our procedure to select coordinates on the image and create a bounding box. Furthermore, after this implementation we observed some kind of over-fitting in the model, because we can see that based on the frame rate of the camera many pictures look exactly the same with small variation of the objects location, so this cause the model to over-fit, and also it is very time consuming to annotate all of these images, so we implemented an augmentation technique which allows us by random zooming, random cropping some regions, and random rotations create a wide, along with horizontal and vertical flip create a wide variety of the images. This implementation of image augmentation was done using the TensorFlow library. Here we can see an snippet code for this implementation:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rescale=1./255,     # Optional: Rescaling factor to normalize pixel values
    rotation_range=20,  # Degree range for random rotations
    width_shift_range=0.2,  # Fraction of total width for random horizontal shifts
    height_shift_range=0.2,  # Fraction of total height for random vertical shifts
    shear_range=0.2,    # Shear intensity (shear angle in degrees)
    zoom_range=0.2,     # Range for random zoom
    horizontal_flip=True,  # Randomly flip inputs horizontally
    fill_mode='nearest'  # Strategy for filling in newly created pixels
    # (e.g., 'nearest' fills with the nearest pixel)
)

generator = datagen.flow_from_directory(
    'path/to/data_directory',
    target_size=(height, width),  # Dimensions to which all images will be resized
    batch_size=batch_size,
    class_mode=None  # Type of label array generated
    # ('binary', 'categorical', 'sparse', 'input', or None)
)
```

By using **ImageDataGenerator**, you can efficiently perform data augmentation on your image dataset, contributing to a more robust and generalized deep learning model. Figure 1 illustrates an example in our dataset corresponding to an image and also its annotated version of the image. Figure 2 illustrates an examples of augmented version of a single image.

2.1.2 Video dataset

In this part, we searched for some videos as case study to optimize and test our model. Two types of videos were selected:

- Firstly we searched and found videos regarding the areal imagery condition, in which these videos can weather provided by satellites, and also it can be taken from drones, recently these types of videos draw attention to them self for vehicle and traffic analysis. This kind of view allows you to obtain data that is accurate enough to identify a driver's shift in gear. It is considerably easier to monitor how drivers interact with one another as well as the connections between different traffic circumstances and objects. Drone-captured traffic data opens up a new chapter in the field of microscopic traffic analysis because of its availability, complexity, accuracy, and range.
- Another type of video used in our study was the normal video taken from the surveillance camera on the roads.

All of these videos are taken from YouTube and RoboFlow platform. Furthermore, to create a more accurate and stable system we used videos under different conditions and circumstances such as collecting videos in busy roads, not busy roads, night and day.

Figure 1: An example of original and annotated version of a instance in dataset

2.2 Model Fine-tuning

To explain the procedure of our fine-tune approach first we give a brief overview of the new version of the **YoloV8** model.

2.2.1 YoloV8

YOLO is a single-shot approach that uses an entire image as input to have a single neural network predict bounding boxes and class probabilities, allowing it to classify an object directly in a single pass. The YOLO family model is always changing. Since then, numerous research teams have published various YOLO versions; YOLOv8 is the most recent version. Figure 3 illustrates a plot representing the characteristics of the different version of Yolo.

YOLOv8 touts faster speed and higher accuracy. On the COCO dataset and A100 TensorRT, for example, the YOLOv8(medium) has a 50.2 mAP score at 1.83 milliseconds. Moreover, YOLO v8's CLI-based implementation and Python package make it simple to use and build. Based on the number of parameters, YOLOv8 is available in five variants: nano(n), small(s), medium(m), large(l), and extra large (x).

As it can be seen in the Figure 4 there are five models in YOLO V8. with the smallest one on top and the largest one on the bottom, For this exercise, I will using the smallest model YOLOv8n.

In Figure 5 we can see an example image passed through YoloV8 for object detection, we can see that the model is accurately detected the objects in the image.

The code that we use to generate the above example and use YoloV8 at inference time is:

```
!pip install ultralytics==8.0.20 #This the library that provides YoloV8
from ultralytics import YOLO # Importing the model
!yolo task=detect mode=predict model=yolov8n.pt conf=0.25 -
source='/kaggle/input/examplepicture/example.jpg' save=True
```

Figure 2: An example of original and annotated version of a instance in dataset

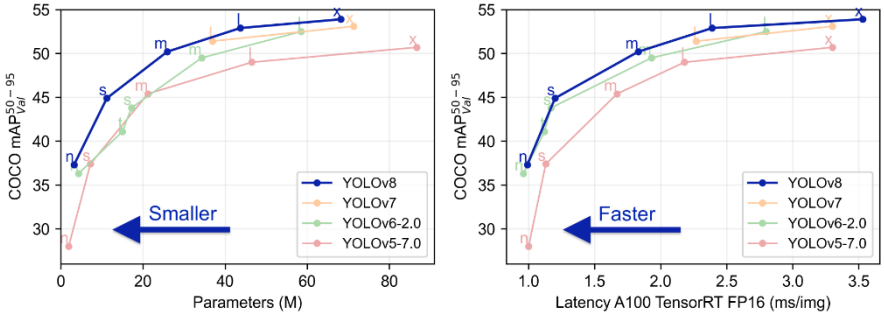

Figure 3: Comparison between different versions (Image taken from ultralytics's GitHub)

```
# Fast way to using the model at inference
```

2.2.2 Fine-tuning

As we mentioned above, for the purpose that our model can work accurately on the situation that we provided in the videos, we have to fine-tune the model using the similar data. In this study we used the "YoloV8n" which is the small version, because for our task this version can fully satisfy the needs and also using a larger version can slows down the procedure, and cause a problem due to our hardware limitations, but for the future use we can easily replace with different versions.

First, we started by creating the appropriate directory for training and validation and testing the YoloV8n model, of course in this study we did not use a test directory sine it was not really important we just want to have estimate of the desired performance, for this reason, we considered our validation set also as test set. The original structure of the YoloV8 directory is as follows:

```
yolov8
##  train
#### images (folder including all training images)
#### labels (folder including all training labels)
##  test
#### images (folder including all testing images)
#### labels (folder including all testing labels)
##  valid
#### images (folder including all testing images)
#### labels (folder including all testing labels)
```

For training YoloV8n, and almost all other version we need to create a configuration file, that we will refer during the training process is data.yaml. The path to the training, testing, and validation

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 4: Analysis of the YoloV8 models (https://github.com/ultralytics/ultralytics)

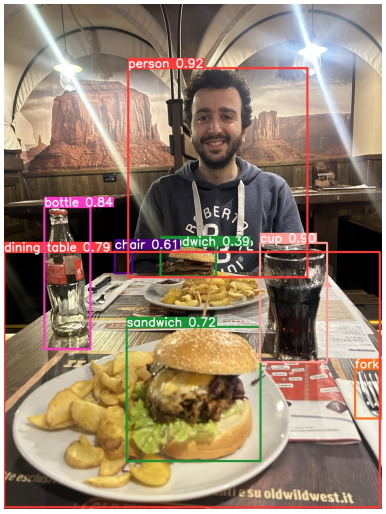

Figure 5: An example of object detection using YoloV8

directories as well as the number of classes required to override the Yolo output classification are all listed in the data.yaml file. The two (three in the original version) directories include the code below in the yolov8 format that we chose while exporting the dataset, next to the labels that match to them.

```
import yaml

data_yaml = dict(
    train = '/kaggle/input/yolov8-vehicle-dataset/yolov8/train/', # Path to training directory
    val = '/kaggle/input/yolov8-vehicle-dataset/yolov8/valid/', # Path to validation directory
    nc =  4, #Number of classes that we want to predict
    names = ['bus', 'cars', 'truck', 'van'] # Name of the clasess
)

with open('/kaggle/working/data.yaml', 'w') as outfile:
    yaml.dump(data_yaml, outfile, default_flow_style=True)   # Saving the file
```

YOLOv8 comes with a command line interface that lets you train, validate or infer models on various tasks and versions. The CLI requires no customization or code. You can run all tasks from the terminal. Usage is fairly similar to the scripts we are familiar with. To begin the training process of the YoloV8 we just need to pass the configurations of the training with command line as we can see below:

```
!yolo task=detect mode=train model=yolov8n.pt -
        data=/kaggle/working/data.yaml epochs=100 imgsz=800 plots=True
```

We trained the YoloV8 for 100 epochs on the dataset that we prepared and mentioned in the previous section. After training we saved the new weights and evaluation results. Figure 6 illustrates the training and validation results on different metric including **box loss**, **cls loss**, **dfl loss**, **map50**, **map50-95**, **precision** and **recall**. These metrics are associated to the Yolo's objective. Yolov8 the loss is divided into two sections: the regression branch, which is in charge of bounding box prediction, uses a combination of two independent losses, Distribution Focal Loss (DFL) and Complete Intersection over Union (CIoU) loss; and the classification branch, which uses Binary Cross Entropy (BCE) loss. The goal of DFL is to solve the issue of class imbalance in object detection tasks. It is also utilized in Yolov8 to enhance bounding box regression, particularly for unpredictable objects with hazy or ambiguous borders. DFL evaluates the probability distribution of bounding box coordinates and takes uncertainty in their placements into account rather than forecasting a fixed bounding box. CIoU loss, on the other hand, considers the aspect ratio differences between the predicted and ground truth boxes in addition to the overlap between them.

"box" refers to the loss associated with bounding box regression, "dfl" stands for Distribution Focal Loss, and "cls" is the standard classification Cross Entropy Loss, i.e. the class loss weight. Figure 6 illustrates the results obtained from training and validation procedure. Furthermore, Figure 7 provides an analysis of the precision and recall curves for the selected labels, along with the validation confusion matrix.

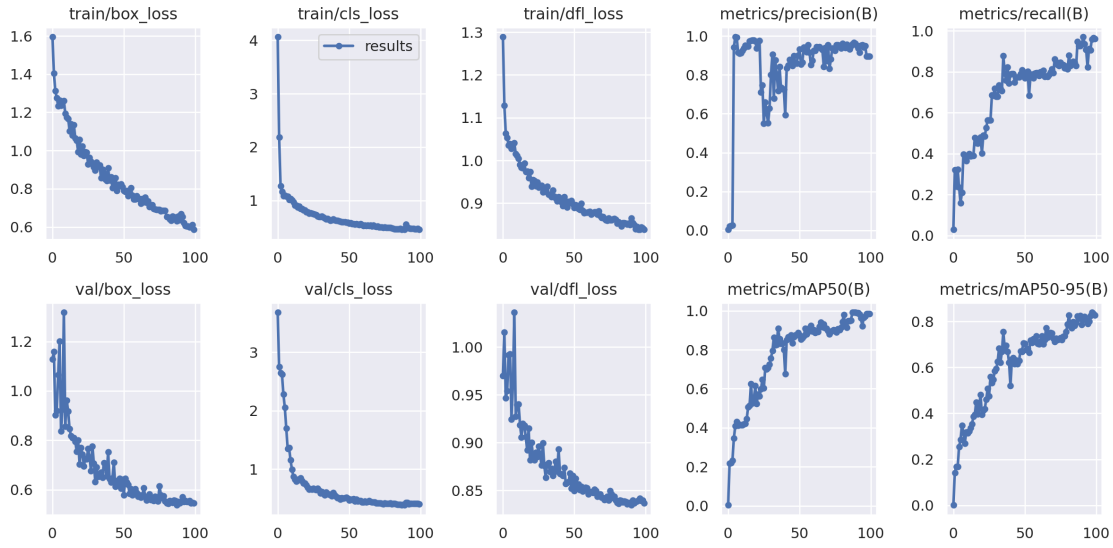

Figure 6: Training and validation analysis

The obtained results are satisfying for the labels that we want to predict, we can see from the confusion matrix that the model is able to accurately identify almost all the labels that we aimed to improve in our dataset. Figure 8 shows an example batch of training, validation, and prediction to better illustrate our procedure.

2.3 Object tracking

Before, we go any further we need to first understand the concept of the object tracking, to track an object in the video first we need to detect the object, and then track the detected objects in each frame of the video. Our implementations heavily uses the base implementations of the ByteTrack[1] and Supervision[2] library, but to better understand the procedure we opened and implemented all the necessary functionalities from these two libraries.

[1]ByteTrack: Multi-Object Tracking by Associating Every Detection Box: https://arxiv.org/abs/2110.06864, GitHub: https://github.com/ifzhang/ByteTrack

[2]Supervision: https://github.com/roboflow/supervision

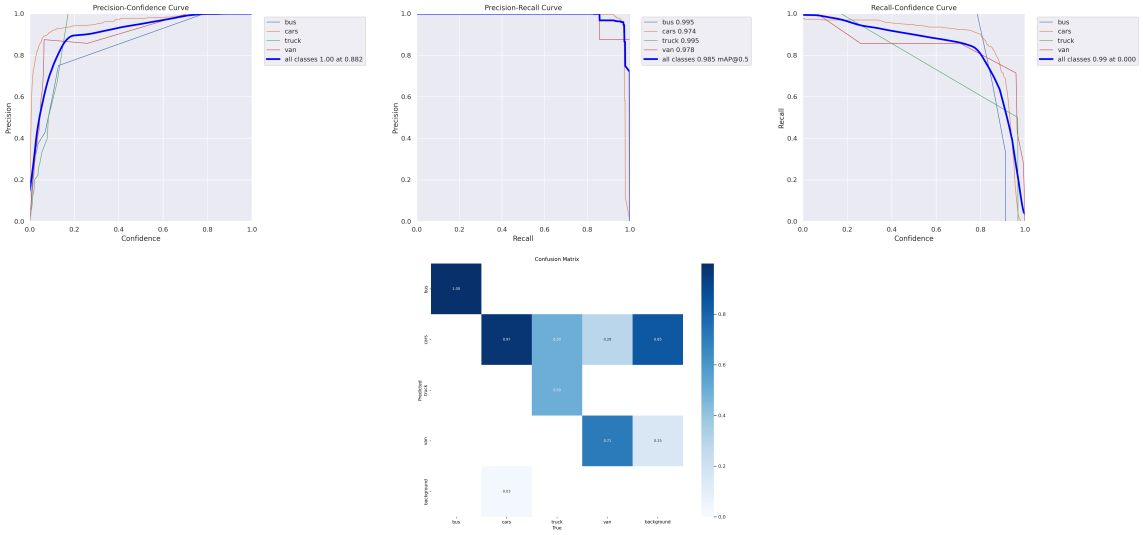

Figure 7: Precision and Recall analysis, Confusion matrix

2.3.1 Object Detection

Firstly, before we go to the object detection in a given video, we used an class concerning a video processing, to better work video and and getting the required information from it. Since the implementations are very long we do not put all the classes and functions used in our system, we just report and show the necessary functionalities (all the implementation can be found in the notebook). The class **VideoInfo** takes in the video path, and read the video with opencv class **VideoCapture** and creates a instance of that video, then it ouptus the information about the **width**, **height**, **frame per second**, and the number of **total frames**. All of these information will be used to process the video which we will explain more clearly in the following discussion.

```
video = cv2.VideoCapture(video_path)
width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps = int(video.get(cv2.CAP_PROP_FPS))
total_frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
```

Then, since we need the detection for every frame of the video we need to implemented and use a function (**get video frames generator**) that is able to read the video frame by frame.

```
generator = get_video_frames_generator(SOURCE_VIDEO_PATH)
```

After that we created a matrix of the frame of the video, we iterate over the matrix, and for each frame we feed the frame to the fine-tuned YoloV8n model that we build in the previous stage. After this we select the detections of the YoloV8n model

```
for index, frame in enumerate(
            get_video_frames_generator(source_path=source_path)):
    # model prediction on single frame
    results = model(frame, verbose=False)[0]
    detections = Detections.from_ultralytics(results)
```

To better illustrate what would be the results obtained from the YoloV8n, we can see the below output below, which shows the results of the detections with bounding box coordinates of the Figure 5, we can see that for every detection we have four coordinates, which are in **(left, top, right, bottom)** format describing a bounding box, of course the output also includes other information which we explain in the following discussion.

```
in: results[0].boxes.xyxy
```

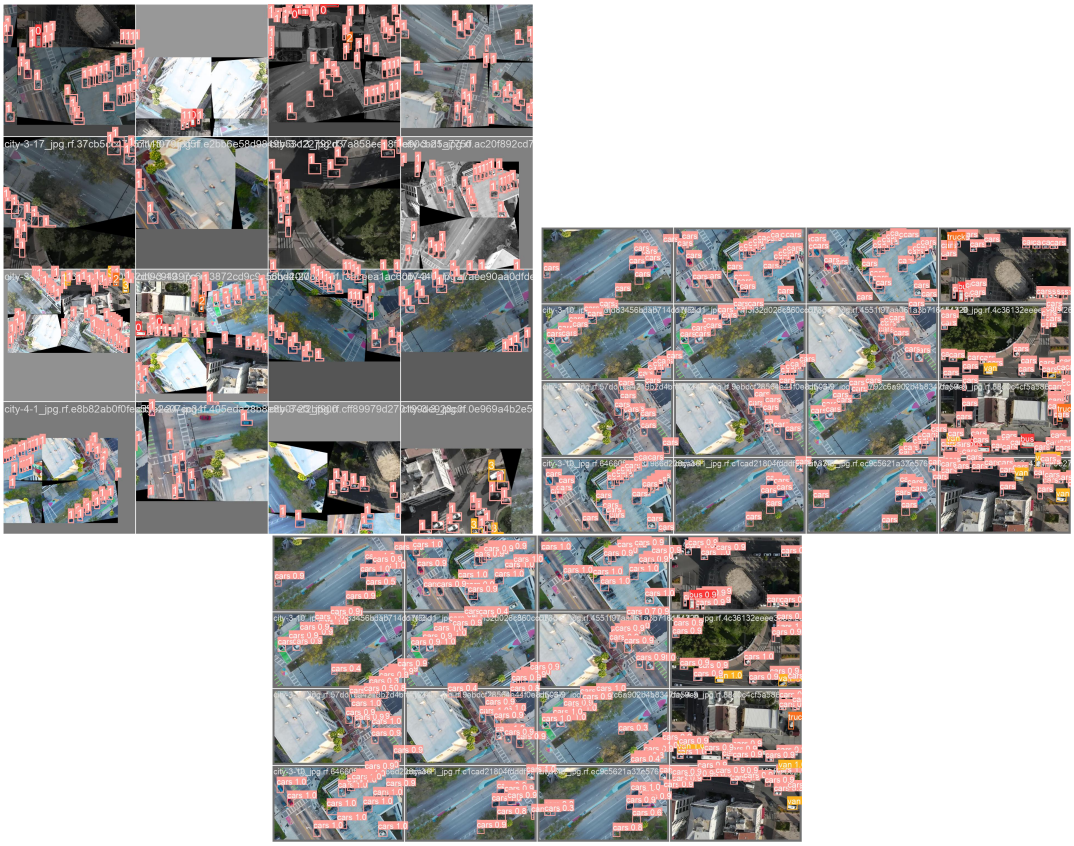

Figure 8: An example of validation, training and prediction batch

```
out: tensor([[ 315.,  162.,  770.,  692.],
        [ 650.,  605.,  820.,  907.],
        [ 102.,  514.,  219.,  874.],
        [   0.,  628.,  954., 1275.],
        [ 311.,  818.,  651., 1158.],
        [ 281.,  621.,  325.,  682.],
        [ 891.,  924.,  960., 1050.],
        [ 397.,  613.,  538.,  690.]], device='cuda:0')
```

After we compute the results of a given frame we feed these into implemented detection class which process these results. Specifically, as it can be seen in the code below, it retrieves, and stores the coordinates of the detections, confidence related to the detecions, class id which describes the class of each detection in the frame, and also it stores the tracker id corresponding to each bounding box (which we will explain in the tracking section) in the instance of the frame.

```
xyxy=ultralytics_results.boxes.xyxy.cpu().numpy(),
confidence=ultralytics_results.boxes.conf.cpu().numpy(),
class_id=ultralytics_results.boxes.cls.cpu().numpy().astype(int),
mask=extract_ultralytics_masks(ultralytics_results),
tracker_id=ultralytics_results.boxes.id.int().cpu().numpy()
if ultralytics_results.boxes.id is not None
else None,
```

After converting these prediction the correct format that we want to work with, now we need to clean our YoloV8n prediction. As we mentioned above, YoloV8n is originally trained on wide variety of the classes and this causes that model predicts every class that can find the frame regardless of

classes that we care about. Accordingly, we write a line of code to filter the classes and only keeping the selected classes that we want for our task.

```
# only consider class id from selected_classes define above
detections = detections[np.isin(detections.class_id, selected_classes)]
```

Also the class **Detections** provides different level of information which can be extracted from the results of the YoloV8n, this is just because of easier implementation of other parts.

2.3.2 Object Tracking

Now that we manged to detect the desired objects in a given frame, we have to track those object for several frames of the video. Regarding this, we employed a modified version of the ByteTracker just be coherent with rest of implementations. First, we need to create an instance of the ByteTracker Class to specify some configuration which is suitable for our system.

```
# create BYTETracker instance
byte_tracker = ByteTrack(track_thresh=0.25, track_buffer=30, match_thresh=0.8, frame_rate=30)
```

This class takes important parameters including, track_thresh which is a detection confidence threshold for track activation, track_buffer indicates the number of frames to buffer when a track is lost, match_thresh is a threshold for matching tracks with detections, and finally frame_rate is the frame rate of the video. After that we have created an instance of this class we will feed the prediction results to a function this class to initialize and update the tracking system. Let's analyze this more deeply.

```
detections = byte_tracker.update_with_detections(detections)
```

we used the above function to update our detections that we did in the previous stage. As we can see in the detection part there is variable (**tracker_id**) associated to tracking the detected object. Now, the procedure that we do with ByteTracker is just to update this information for our detections. Accordingly, given the output of the detection first we use the **detections2boxes** function we create a numpy tensor of shape **(x_min, y_min, x_max, y_max, confidence, class_id)**, then, we feed this numpy tensor to the function called **update_with_tensors**. This function updates the tracker with the provided tensors and returns the updated tracks. To better understand this procedure we should first explain the ByteTracker functionality.

```
tracks = self.update_with_tensors(
        tensors=detections2boxes(detections=detections))
```

ByteTracker's primary concept is straightforward: non-background low score boxes are retained for a secondary association step between the preceding and subsequent frames, depending on their tracklet similarity. By retaining pertinent bounding boxes that otherwise would have been eliminated due to poor confidence scores (caused by occlusion or appearance changes), this helps to enhance tracking consistency. ByteTracker is extremely flexible to any object detection (YOLO, RCNN) or instance association components (IoU, feature similarity) thanks to its generic structure.

ByteTracker's main novelty is that it uses low-score detection boxes for a subsequent association step instead of discarding non-background low confidence detection boxes, which are usually deleted after the first filtering of detections. Ocluded detection boxes often have confidence ratings lower than the threshold, but because they still contain some information about the objects, their scores are greater than those of background-only boxes. Because of this, it is still important to monitor these low confidence boxes during the association phase.

To better illustrate our point let's look at the example in the Figure 9 taken from the ByteTracker research paper. In first round **(a)** the model successfully detects all the people and shows the detection for all of the people in the all of the frame even the ones with lower confidence. Then, at the second round **(b)**, it only considered the detections with high confidence with previously initialized tracks. Finally, at round **(c)** ByteTracker attempts to link low-confidence bounding boxes to the remaining unmatched tracks. During this stage, the red object—which had been disregarded earlier—recovers, and falsely detected background object are eliminated.

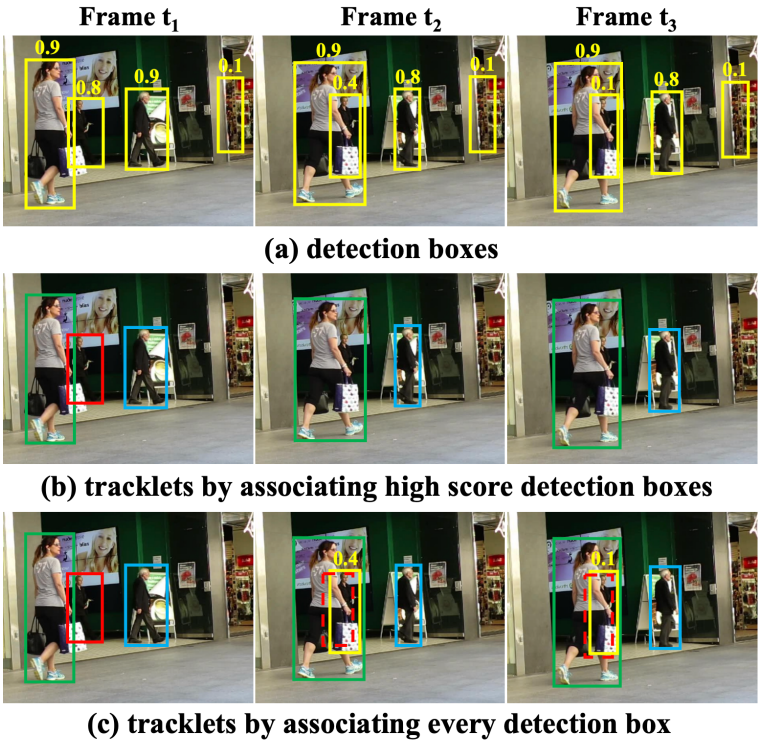

Figure 9: Image taken from ByteTracker paper

By using a Kalman filter, ByteTracker keeps track of how things move between frames. Similar to predicting a ball's landing point based on its present trajectory, this filter allows ByteTracker to anticipate an object's future location based on its past motion patterns. As a result, ByteTracker can continue to follow an object even if it is not recognized in a given frame by using its predictions. Since the python implementation is very long and complex we illustrated the pseudo-code (Figure 10) that was provided by the ByteTracker to better illustrate the idea behind it.

As it can be seen in the algorithm Figure 10, the input of ByteTracker is a video sequence V, along with an object detector Det. We also set a detection score threshold t. The output of ByteTracker is the tracks T of the video and each track contains the bounding box and identity of the object in each frame. For each frame of the video, the detecion model predicts the results of the detections and the scores. In this algorithm it seperates all the detections in two section high condifidence, denoted by D_{high}, and low confidence denoted by D_{low} based on the detection threshold that, we specify when initializing the algorithm. To illustrate it more clearly, detecions with higher confidence than the selected threshold are placed in the D_{high}, and the detecions with lower confindence than the threshold are placed in the D_{low}. After this sepration, the **Kalman Filter** is applied to estimate the new locations in the current frame of each track T.

To give a quick overview of the **Kalman Filter**, let's look at a basic illustration of tracking a moving object's position. Assume we have a sensor that tracks a ball's location as it travels horizontally in a straight line. Nevertheless, there could be mistakes and noise in the sensor readings. By combining the noisy data with predictions based on the ball's past position and velocity, the Kalman filter assists us in estimating the true position of the object. This is how it operates:

- Initialization: To begin, we first provide an initial estimate of the ball's velocity and position to the filter. We also provide a definition for the noise or uncertainty for these preliminary estimations.

- Prediction: We project the ball's projected position for the upcoming time step based on its estimated position and velocity from the previous time step. This forecast considers the ball's motion model, which assumes that it travels at a constant velocity.

- Measurement Update: We match each new measurement from the sensor with our estimated position. To account for measurement noise, we compute the "residual" or difference between the expected and measured positions.
- Kalman Gain: The weight or influence of the measurement in revising our estimate is determined by the Kalman filter's calculation of a parameter known as the Kalman gain. It considers the uncertainty of the sensor measurements as well as the anticipated state.
- State Update: We update our estimate of the ball's position and velocity depending on the measurement by using the Kalman gain. This update gives more weight to the more trustworthy information by combining the measurement and the expected condition.
- Iteration: With each new measurement, the prediction, measurement update, and state update processes are repeated, allowing the estimation of the ball's position to be improved iteratively as fresh data becomes available.

Then continuing our discussion, after updating the tracking IDs, we created an empty instance of the detection class to update and replace all the information in that instance, such as detection coordinates, class ids, and tracker id. Finally, we have the updated detection results that now include tracking IDs.

```
detections = Detections.empty()
if len(tracks) > 0:
    detections.xyxy = np.array(
        [track.tlbr for track in tracks], dtype=np.float32)
    detections.class_id = np.array(
        [int(t.class_ids) for t in tracks], dtype=int)
    detections.tracker_id = np.array(
        [int(t.track_id) for t in tracks], dtype=int)
    detections.confidence = np.array(
                [t.score for t in tracks], dtype=np.float32)
else:
    detections.tracker_id = np.array([], dtype=int)
```

2.4 Object counting

Now, that we have discussed how we detect and track our detections, we need to describe the way that we can count the cars in certain domain. As it can be seen in the Figure 11, we draw a line or in the videos that are taken from the intersection regions. These lines and polyzone regions are draw differently for each type of video. Let us describe how we processed with this, we first start with simple line placed on the image of the busy/empty road (the middle and bottom figures), we can see that in this line there are two numbers one is **in** that describes the number detections that are going away from us passing this line, and the **out** describes the number of detections passing this line coming towards us, we look at the two figures, in the middle one, on the right side are the detections that are going out, and the left side are the detections that are coming in, for the bottom figure this is different, on the left side we have the cars going out and on the right side we have the cars going in. We will describe this procedure also for intersections in the following discussion.

2.4.1 Simple Line

So, let us first discuss the idea behind the two figures (the middle and bottom one), first we need to select two point on the image, what do we mean?. as you know to draw a line we need two points so by selecting two point one denoted by the **LINE_START** which describes the position where the line starts and another one **LINE_END** which describes the position where the line ends. The way we selected these coordinates it was by taking an frame of the video at random and saving that frame as an image, then loading the image in another environment and selecting point. The code used for selecting the point on the image is described in the appendix section of the report. As an example here we showed the coordinates selected for the middle image, but the procedure is following the same logic also for the bottom image. So, we know that the shape of the frame in the middle image is **(width=1280,**

height=720, fps=30, total_frames=1802), since we want to our line covers the whole area of the video, we selected the start point at the coordinates **(1, 300)** in which **300** describes the height of the line which this depends on the case that we have under study, we reached to the conclusion that **300** is suitable for the middle image. Similarly, we selected the end point at approximately end of the width of the video frame which is **1280-1** with same height **300**. After that we setup our coordinates that we want to count the detected objects based on that, we will initiate an instance of the class **LineZone** based on these coordinates. This class is responsible for counting the number of objects that cross a predefined line. After that we initiated this class and also got the detection results and tracking results as it described in the previous section, we will use **trigger** function of this class and passing the results of the detections as an argument to this function.

```
# settings
LINE_START = Point(1, 300)
LINE_END = Point(1280-1, 300)

# create LineZone instance, it is previously called LineCounter class
line_zone = LineZone(start=LINE_START, end=LINE_END)
# update line counter
line_zone.trigger(detections)
```

In this function (**trigger**), we will update the **'in_count'** and **'out_count'** based on the objects that cross the line. Firstly, we will define two Boolean variables **crossed_in** and **crossed_out** initialized with **False**. Then, we enumerate over the detection results which as we described above, contains the bounding box coordinates (**xyxy**), confidence, class IDs, and updated tracker IDs that we did in the previous section. Then, we check whether that the detections are tracked or not, we only care about the detections that we tracking, so if they are not tracked we just skip those detections. Now, for the ones that we are tracking, with bounding box coordinate we will create a list of anchors, then in the next line with class **Vector** (that is initialized with coordinates of the line) and **is_in** function we check whether that given point is on the left or right side (with respect to the direction) of the vector. The **is_in** method takes a Point object as a parameter. It creates two vectors, **v1** and **v2**, where **v1** is the original vector (from start to end), and **v2** is a vector from start to the given point. It calculates the cross product of these two vectors using the determinant of a 2x2 matrix. If the cross product is less than **0**, it returns True, indicating that the point is on one side of the vector; otherwise, it returns False. This is a common technique used in computational geometry to determine the relative positions of points and vectors. If only **two** different trigger states are present among the anchors, it means the object is straddling the line and not fully crossed. In such cases, the iteration continues to the next detection. Otherwise, we check that if the anchor is inside which is means that the **is_in** function returned **True**, we increment the count of **cross_in**, on the contrary, if the function returns **False** we increment the count of **cross_out**. In summary, this is the logic behind the counting the number of vehicles that coming in and out of the road.

```
def trigger(self, detections: Detections) -> Tuple[np.ndarray, np.ndarray]:
        crossed_in = np.full(len(detections), False)
        crossed_out = np.full(len(detections), False)

        for i, (xyxy, _, confidence, class_id, tracker_id) in enumerate(detections):
            if tracker_id is None:
                continue

            x1, y1, x2, y2 = xyxy
            anchors = [
                Point(x=x1, y=y1),
                Point(x=x1, y=y2),
                Point(x=x2, y=y1),
                Point(x=x2, y=y2),
            ]
            triggers = [self.vector.is_in(point=anchor) for anchor in anchors]
```

```
            if len(set(triggers)) == 2:
                continue

            tracker_state = triggers[0]

            if tracker_id not in self.tracker_state:
                self.tracker_state[tracker_id] = tracker_state
                continue

            if self.tracker_state.get(tracker_id) == tracker_state:
                continue

            self.tracker_state[tracker_id] = tracker_state
            if tracker_state:
                self.in_count += 1
                crossed_in[i] = True
            else:
                self.out_count += 1
                crossed_out[i] = True

        return crossed_in, crossed_out
```

2.4.2 PolyGons Regions (Intersections)

Now, moving on to the intersections (top figure), is more complex, since in this condition we have **eight** paths, **four** of them **coming in** to the intersection and **four** of them **going out** from the intersection. In these types of video, we define and draw eight poly **POLYGONS** on the eight paths, which have four entrance (**ZONE_IN_POLYGONS**) and four exits (**ZONE_OUT_POLYGONS**), every line in this list illustrate a **PolyGons** which we selected four points using the mentioned procedure that we discussed above, to create it. Then, we initiate an instance of the **initiate_polygon_zones** function to create an polyzone class instance.

```
ZONE_OUT_POLYGONS = [
    np.array([[479, 237], [530, 277], [463, 355], [412, 309]]),
    np.array([[502, 579], [562, 665], [644, 637], [548, 540]]),
    np.array([[918, 422], [953, 457], [909, 536], [854, 489]]),
    np.array([[830, 170], [760, 116], [727, 154], [793, 209]]),
]

ZONE_IN_POLYGONS = [
    np.array([[589, 113], [506, 207], [553, 244], [639, 151]]),
    np.array([[416, 498], [478, 564], [529, 525], [468, 452]]),
    np.array([[881, 566], [795, 666], [743, 623], [829, 522]]),
    np.array([[969, 283], [876, 197], [830, 245], [926, 329]]),
]

# Specifying the Zone that the cars get in to
zones_in = initiate_polygon_zones(
            ZONE_IN_POLYGONS, video_info.resolution_wh, Position.CENTER
        )
# Specifying the Zone that the cars get out of
zones_out = initiate_polygon_zones(
            ZONE_OUT_POLYGONS, video_info.resolution_wh, Position.CENTER
        )
```

After specifying the points and drawing the **PolyGons** on the video frames, it is time describe how we proceed with counting. First, let us briefly discuss how, the counting is shown on the video.

Since, we want to give a separate analysis of the cars entering and existing the intersection, based on each path of the intersection, we used a color coded scheme which separates different paths of the intersection (both existing and entering), so based on this procedure we can describe for example the number of cars entering from the top path and existing from the right path in the intersection, as it can be seen in the Figure 11 in the top image, we have for example **two** cars entering from the below path (the green one) and existing from the right path (the yellow one).

Now that we understand how it works we need to describe the idea behind the implementation. Let us look at the code below. As we previously discussed this, this is the code for processing one frame of the video, in which at the first step, we will give the frame of the video as input to our detection model (YoloV8n) the we fit the results to the **from_ultralytics**, and then we finally update the tracker IDs using the ByteTracker class implementation and process them as we described fully in above sections. The important part of the code is inside the **for** which we want to first detect the cars passing through these zones. In this part, we iterate over the predefined zones that we described above, and check the detections whether are inside these zones or not (we do this for both entrance and existing zones). We do this by the **trigger** function of the **initiate_polygon_zones** to check consider the detections that are insides these regions (please note that we only consider the detections that are passing these regions whether going inside or going outside the intersection).

```
results = self.model(
            frame, verbose=False, conf=self.conf_threshold, iou=self.iou_threshold
        )[0]
detections = Detections.from_ultralytics(results)
detections.class_id = np.zeros(len(detections))
detections = self.tracker.update_with_detections(detections)

detections_in_zones = []
detections_out_zones = []

for i, (zone_in, zone_out) in enumerate(zip(self.zones_in, self.zones_out)):
    detections_in_zone = detections[zone_in.trigger(detections=detections)]
    detections_in_zones.append(detections_in_zone)
    detections_out_zone = detections[zone_out.trigger(detections=detections)]
    detections_out_zones.append(detections_out_zone)

    detections = self.detections_manager.update(
            detections, detections_in_zones, detections_out_zones
    )
```

Let us discuss more deeply what is happening inside this **trigger** function. We can see that this function receive the detection as input, and determines if the detections are within the polygon zone or not. The bounding boxes **(xyxy)** of the detections are clipped to fit within the frame resolution. The **clip_boxes** function is handling this operation. This function returns bounding box with coordinates clipped to fit within the frame resolution. The **clipped_xyxy** values are used to replace the original **xyxy** values in the detections object. This ensures that subsequent operations use the clipped bounding box coordinates. Then, we get the anchors (coordinates) for triggering within the bounding box. The **is_in_zone** array is obtained by indexing the mask attribute of the PolygonZone instance with the **clipped_anchors** coordinates. This array is a boolean mask indicating whether each anchor point is within the defined polygon zone or not. The **current_count** attribute is updated based on the number of detections within the polygon zone. This count is the number of True values in the **is_in_zone** array.

```
def trigger(self, detections: Detections) -> np.ndarray:
        clipped_xyxy = clip_boxes(
            xyxy=detections.xyxy, resolution_wh=self.frame_resolution_wh
        )
        clipped_detections = replace(detections, xyxy=clipped_xyxy)
        clipped_anchors = np.ceil(
            clipped_detections.get_anchors_coordinates(anchor=self.triggering_position)
```

```
        ).astype(int)
        is_in_zone = self.mask[clipped_anchors[:, 1], clipped_anchors[:, 0]]
        self.current_count = int(np.sum(is_in_zone))
        return is_in_zone.astype(bool)
```

We used the above function for both entrance and existing regions to detect the objects insider these regions. Following this, now we need to count the objects, as you can see in the Figure 10 we only count the only the detecions that are existing from the intersection, because by summing these number we can also compute the number of detections coming inside the intersection. For counting these detections we used the below procedure (code), in this code we enumerate over all the **zone_out**, detected objects, the in the second line we get the polygon center fur illustration purposes. Then, we check whether that if this detection are exists in our detections list, if this was the case we retrieve the counts that we set in the code above for every polygon zone exist, and finally in the second loop we iterate over the IDs of the entrance polyzone (because we also have ID that describes which entrance zone is the detecion belongs to), then we both zone in and zone out IDs we retrieve the count, and show it on the video frame.

```
for zone_out_id, zone_out in enumerate(self.zones_out):
            zone_center = get_polygon_center(polygon=zone_out.polygon)
            if zone_out_id in self.detections_manager.counts:
                counts = self.detections_manager.counts[zone_out_id]
                for i, zone_in_id in enumerate(counts):
                    count = len(self.detections_manager.counts[zone_out_id][zone_in_id])
                    text_anchor = Point(x=zone_center.x, y=zone_center.y + 40 * i)
                    annotated_frame = draw_text(
                        scene=annotated_frame,
                        text=str(count),
                        text_anchor=text_anchor,
                        background_color= COLORS.colors[zone_in_id],
                    )
```

3 Conclusion

All in all, this project provide details about the systematic approach taken to develop a traffic monitoring system. We start off with data collection and annotation for both images, in this project we undertook fine-tuning of the YOLOv8n model to suit specific monitoring conditions. Object tracking, facilitated by the ByteTracker library, enhanced the model's ability to associate detections across frames. Object counting strategies were implemented, incorporating both simple line approaches for busy/empty roads and complex polygon zone methods for intersections. The comprehensive methodology, integrating data processing, model adaptation, tracking, and counting techniques, lays the foundation for a robust and adaptable traffic surveillance system capable of addressing diverse scenarios. In the future analysis, a good improvment of this project would be to implement and analysis the speed of the vehicles passing through a road. Figure 12 represents the results capture from the videos.

4 Appendix

4.1 Evaluation metric

4.1.1 Intersection over Union (IoU)

A metric called Intersection over Union (IoU) is used to quantify how much a predicted bounding box and a ground truth bounding box overlap. It is essential to assessing how accurate object localization is.

$$IoU = \frac{\text{Area\_of\_Overlap}}{\text{Area\_of\_Union}}$$

4.1.2 Average Precision (AP)

AP is calculated by summing up the precision values at different recall levels and then integrating over the recall range. It can be represented as:

$$AP = \int_0^1 \text{Precision}(R)\, dR$$

4.1.3 Mean Average Precision (mAP)

mAP is the mean Average Precision which is calculated according to the following formula, if we consider N as the number of classes:

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i$$

where AP_i is the AP at class i.

mAP extends the concept of AP by calculating the average AP values across multiple object classes. This is useful in multi-class object detection scenarios to provide a comprehensive evaluation of the model's performance.

- mAP50: Mean average precision calculated at an intersection over union (IoU) threshold of 0.50. It's a measure of the model's accuracy considering only the "easy" detections.
- mAP50-95: The average of the mean average precision calculated at varying IoU thresholds, ranging from 0.50 to 0.95. It gives a comprehensive view of the model's performance across different levels of detection difficulty.

4.1.4 Precision

Precision quantifies the proportion of true positives among all positive predictions, assessing the model's capability to avoid false positives.

$$Precision = \frac{\text{True Positives}}{\text{True Positives + False Positives}}$$

4.1.5 Recall

Recall calculates the proportion of true positives among all actual positives, measuring the model's ability to detect all instances of a class.

$$Recall = \frac{\text{True Positives}}{\text{True Positives + False Negatives}}$$

4.1.6 F1-Score

The F1 Score is the harmonic mean of precision and recall, providing a balanced assessment of a model's performance while considering both false positives and false negatives.

$$F1\ Score = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.1.7 True Positive, False Positive, False Negative

A prediction is said to be correct if the class label of the predicted bounding box and the ground truth bounding box is the same and the IoU between them is greater than a threshold value.

Based on the IoU, threshold, and the class labels of the ground truth and the predicted bounding boxes, we calculate the following metrics

- True Positive: The model predicted that a bounding box exists at a certain position (positive) and it was correct (true).
- False Positive: The model predicted that a bounding box exists at a particular position (positive) but it was wrong (false).
- False Negative: The model did not predict a bounding box at a certain position (negative) and it was wrong (false) i.e. a ground truth bounding box existed at that position.
- True Negative: The model did not predict a bounding box (negative) and it was correct (true). This corresponds to the background, the area without bounding boxes, and is not used to calculate the final metrics.

4.2 Point Selection

This Python code uses the OpenCV library to display an image and capture mouse click events on the image. The script defines a function **click_event** that is called when a mouse click event occurs. The function displays the coordinates of the clicked point and, in the case of a left mouse click, also displays the pixel color at that point. All the explanations are provide as comment in each line of the code.

```
# importing the module
import cv2

# function to display the coordinates of
# of the points clicked on the image
def click_event(event, x, y, flags, params):

# checking for left mouse clicks
if event == cv2.EVENT_LBUTTONDOWN:

# displaying the coordinates
# on the Shell
print(x, ' ', y)

# displaying the coordinates
# on the image window
font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(img, str(x) + ',' +
str(y), (x,y), font,
1, (255, 0, 0), 2)
cv2.imshow('image', img)

# checking for right mouse clicks
```

```
if event==cv2.EVENT_RBUTTONDOWN:

# displaying the coordinates
# on the Shell
print(x, ' ', y)

# displaying the coordinates
# on the image window
font = cv2.FONT_HERSHEY_SIMPLEX
b = img[y, x, 0]
g = img[y, x, 1]
r = img[y, x, 2]
cv2.putText(img, str(b) + ',' +
str(g) + ',' + str(r),
(x,y), font, 1,
(255, 255, 0), 2)
cv2.imshow('image', img)

# driver function
if __name__=="__main__":

# reading the image
img = cv2.imread("C:/Users/alino/Downloads/saved_frame (3).png", 1)

# displaying the image
cv2.imshow('image', img)

# setting mouse handler for the image
# and calling the click_event() function
cv2.setMouseCallback('image', click_event)

# wait for a key to be pressed to exit
cv2.waitKey(0)

# close the window
cv2.destroyAllWindows()
```

Algorithm 1: Pseudo-code of BYTE.

Input: A video sequence V; object detector Det; detection score threshold τ

Output: Tracks $\mathcal{T}$ of the video

Initialization: $\mathcal{T} \leftarrow \emptyset$
for *frame* f_k *in* V **do**
  /* Figure 2(a) */
  /* predict detection boxes & scores */
  $\mathcal{D}_k \leftarrow \texttt{Det}(f_k)$
  $\mathcal{D}_{high} \leftarrow \emptyset$
  $\mathcal{D}_{low} \leftarrow \emptyset$
  for d *in* $\mathcal{D}_k$ **do**
    if $d.score > \tau$ **then**
      $\mathcal{D}_{high} \leftarrow \mathcal{D}_{high} \cup \{d\}$
    end
    else
      $\mathcal{D}_{low} \leftarrow \mathcal{D}_{low} \cup \{d\}$
    end
  end
  /* predict new locations of tracks */
  for t *in* $\mathcal{T}$ **do**
    $t \leftarrow \texttt{KalmanFilter}(t)$
  end
  /* Figure 2(b) */
  /* first association */
  Associate $\mathcal{T}$ and $\mathcal{D}_{high}$ using Similarity#1
  $\mathcal{D}_{remain} \leftarrow$ remaining object boxes from $\mathcal{D}_{high}$
  $\mathcal{T}_{remain} \leftarrow$ remaining tracks from $\mathcal{T}$
  /* Figure 2(c) */
  /* second association */
  Associate $\mathcal{T}_{remain}$ and $\mathcal{D}_{low}$ using similarity#2
  $\mathcal{T}_{re-remain} \leftarrow$ remaining tracks from $\mathcal{T}_{remain}$
  /* delete unmatched tracks */
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \mathcal{T}_{re-remain}$
  /* initialize new tracks */
  for d *in* $\mathcal{D}_{remain}$ **do**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{d\}$
  end
end
Return: $\mathcal{T}$

Figure 10: Image taken from ByteTracker paper

Figure 11: Line and polyzone regions for counting and detection

Figure 12: Results taken from videos