Computer Vision Project

on

Counting and Tracking Vehicles

By

Yugandhar Rajendra Dhande (142202023)
Amar Parajuli (142202004)

Guided by

Dr.Satyajit Das

Dec 2023

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

Contents

1 Introduction

1.1 What is the project all about?

Our project focuses on the challenge of vehicle count in video footage. At first glance, it might seem that simply detecting and tallying vehicles would suffice. However, this approach is suitable for still images, not videos. In video analysis, each frame is processed individually. Therefore, in a video with a frame rate of 30 frames per second, a single vehicle would be erroneously counted 30 times. To resolve this, we incorporate object tracking into our methodology. Object tracking involves observing and following an object's movement across a sequence of video frames or image streams. This task can be complex, particularly under conditions such as occlusion, motion blur, or when the object exits and re-enters the frame after a brief absence. Various techniques have been developed to tackle these challenges, and our report will explore some of these methods in detail.

1.2 State-of-the-art algorithms and our choice

In our pursuit of excellence, we delved into the current state-of-the-art tracking algorithms, drawing inspiration from them. As of the latest benchmarks, specifically MOT17 and MOT20, SMILEtrack1 emerges as the leading tracker (referenced in figure 1). Introduced in November 2022, SMILEtrack integrates a Similarity Learning Module (SLM), inspired by the Siamese network, to effectively extract vital appearance features of objects and merge these with motion features.

Rank	Model	MOTA↑	IDF1	HOTA
1	SMILEtrack	78.2	77.5	63.4
2	SparseTrack	78.2	77.3	63.4
3	BoT-SORT	77.8	77.5	63.3
4	ByteTrack	77.8	75.2	61.3
5	STGT	77.5	75.2	

Rank	Model	HOTA	MOTA↑	IDF1
1	SMILEtrack	65.24	81.06	80.5
2	SparseTrack	65.1	81.0	80.1
3	BoT-SORT	65.0	80.5	80.2
4	ByteTrack	63.1	80.3	77.3
5	StrongSORT	64.4	79.6	79.5

Figure 1: left: MOT20 ranking of trackers Right: MOT17 ranking of trackers

However, the performance gap between SMILEtrack and the other top four trackers is marginal, less than 1 percent. It's noteworthy that SMILEtrack's development heavily relied on insights from ByteTrack and YOLOv7, both of which rank among the top five trackers. Considering this, and to avoid merely replicating SMILEtrack's approach, we chose to focus on ByteTrack and YOLO.

Our rationale stems from the fact that ByteTrack significantly contributed to SMILEtrack's success. Furthermore, a more recent version of YOLO has been released post the publication of the SMILEtrack paper, offering new avenues for exploration and combination in our project.

2 Our approach

2.1 Cascade Classification

Prior to experimenting with ByteTrack, we explored several alternatives to understand their limitations. One such method was utilizing a cascade classifier trained on images of vehicles. This classifier operates through multiple levels, each containing a series of 'weak' classifiers. The term 'weak' implies that while these classifiers may not be highly accurate in object detection individually, they are efficient in terms of computational resources. Although this approach wasn't entirely ineffective, it resulted in a significant number of false negatives, even in simple single-image tests. This issue was exacerbated in more complex scenarios, such as those involving occlusion, as evidenced by similar findings in other experiments.

2.2 DEtection TRansformer

The DEtection TRansformer (DETR) marks a transformative approach in computer vision for object detection. Developed by Facebook AI Research, it uniquely integrates the Transformer model, commonly used in natural language processing, with a standard convolutional neural network (CNN) backbone. This integration signifies a shift from traditional methods, which often rely on multiple hand-designed components.

DETR stands out for its end-to-end training capability, streamlining the object detection process. Unlike conventional methods that use a multi-stage pipeline including region proposal networks and non-maximum suppression (NMS), DETR directly predicts bounding boxes and object classes in a single step, enhancing efficiency and reducing pipeline complexity.

A key feature of DETR is its set-based global loss function, which ensures unique predictions through bipartite matching. This method optimally aligns predictions with ground truth, differing from the per-element loss functions in traditional systems.

However, DETR's innovation comes with the trade-off of longer training times, attributed to the complexity of the Transformer architecture and the global nature of its loss function. Despite this, DETR's novel approach and simplification of the object detection process make it a noteworthy development in the field.

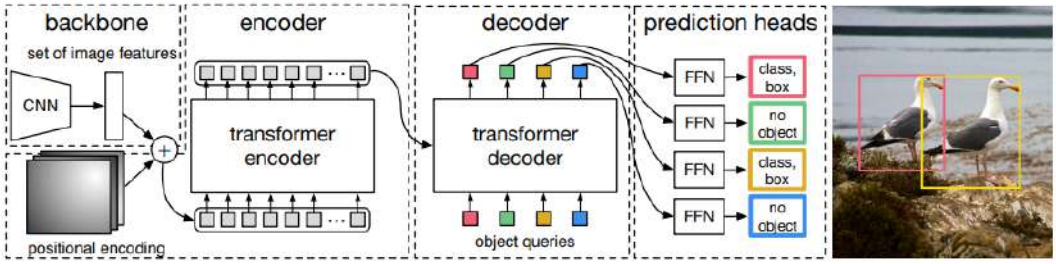

Figure 2: DEtection TRansformer Architecture

2.3 YOLO

After conducting further research, we came across the YOLO2 object detection model, which has gained widespread popularity for its effectiveness in this domain. YOLO stands for "You Only Look Once," and its nomenclature is aptly descriptive of its approach. YOLO distinguishes itself by its speed, straightforward architecture, and its departure from region-proposal methods used by previous approaches like R-CNN and Faster R-CNN.

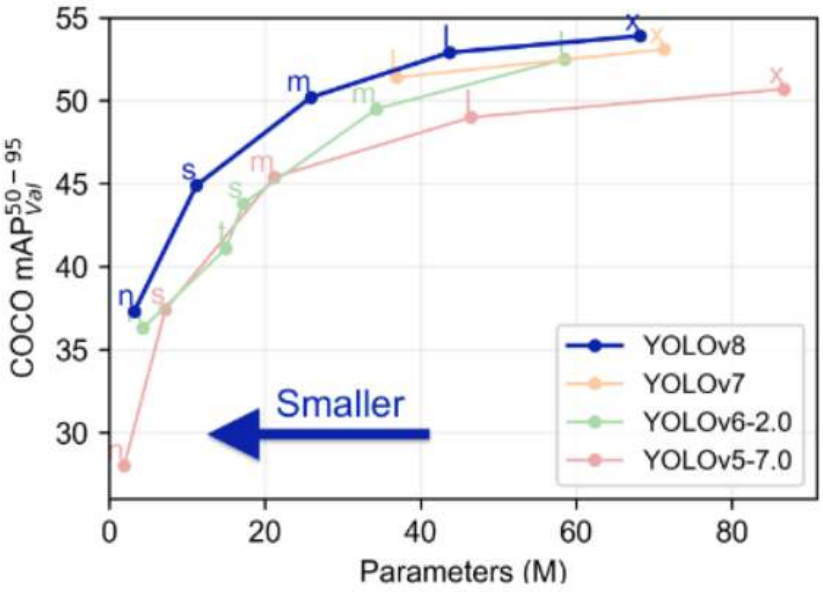

Figure 3: YOLO version comparision on COCO mAP 50-95

Unlike prior methods that involve generating a set of region proposals or candidate object regions in an image through techniques like selective search

or region proposal networks, YOLO takes a different route. It predicts both bounding boxes (regions) and class probabilities in a single pass, thus earning its moniker "You Only Look Once." This streamlined process eliminates the need for additional processing steps, ultimately reducing computation time.

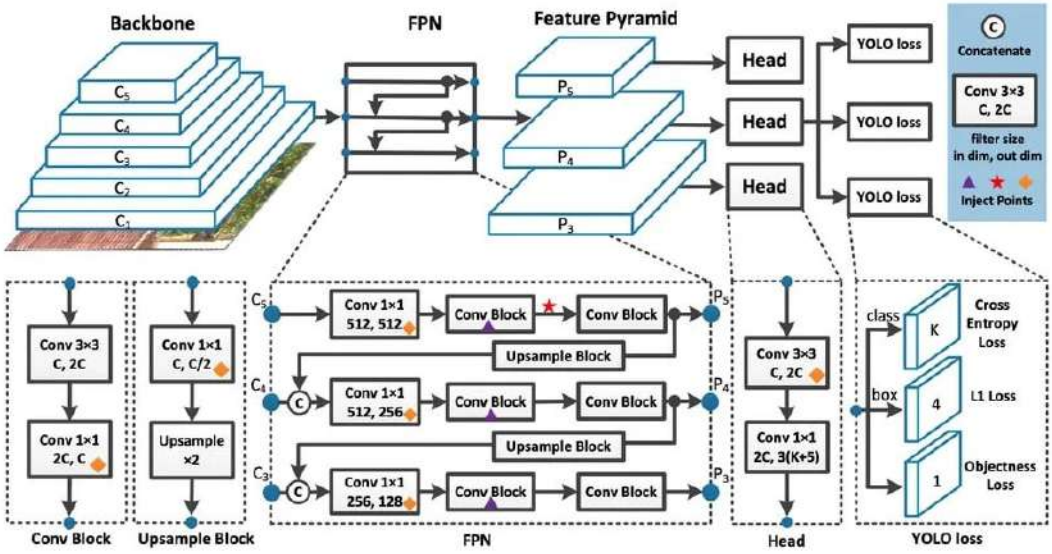

Figure 4: YOLOv8 Architecture

Another noteworthy feature of YOLO is its utilization of Non-Maxima Suppression (NMS) to address the issue of multiple bounding boxes for a single object. NMS leverages the intersection over union (IoU) metric to remove redundant boxes through the following steps:

- Overlapping Regions: Determine the overlap between bounding boxes.
- Sorting: Sort bounding boxes by confidence score.
- Highest Confidence Box: Select the box with the highest confidence as the reference.
- Intersection over Union (IoU): Measure overlap using IoU.
- Thresholding: Set a minimum IoU threshold (e.g., 0.5).
- Suppression: Discard overlapping boxes exceeding the threshold.
- Iterative Process: Repeat steps 3-6 with the next highest confidence box.
- Final Bounding Boxes: The remaining non-overlapping boxes constitute the final predictions.

It's worth noting that YOLO comes in various versions, each with its own subversions. These subversions vary in terms of parameters, impacting accuracy and speed differently.

The different iterations of YOLO models are subjected to a comparative analysis using the mAP (mean Average Precision) metric. This evaluation encompasses IoU (Intersection over Union) thresholds spanning from 50 to 95, with increments of 5. The mAP is a critical measure, and its computation follows a specific formula:

$$\text{mAP} = (\sum_{k=1}^{n} \text{AP}_k)\frac{1}{n}$$

AP_k = Average Precision of class k

n = the number of classes

Within this formula, the process begins by computing individual precisions for each class by applying varying IoU thresholds. A detection is classified as a true positive when the IoU metric surpasses the specified threshold between the predicted bounding box and the actual bounding box. After calculating these precisions for a particular class, their average is determined, and this process is repeated for all other classes. Ultimately, the mean of these Average Precisions across all classes yields the mAP value for the respective YOLO model.

In our selection process, we opted for YOLOv8 version x due to its superior mAP compared to other versions. This decision prioritized accuracy over speed, as real-time video processing was not a requirement for our project.

2.4 DETR Vs YOLOv8

The DEtection TRansformer (DETR) and YOLOv8 represent two distinct approaches in computer vision's object detection. DETR, from Facebook AI Research, merges the Transformer model with a CNN backbone, deviating from traditional, component-heavy object detection methods. Its standout feature is the end-to-end training process, simplifying the pipeline by directly predicting object classes and bounding boxes in one step, enhancing efficiency and reducing complexity.

Central to DETR's innovation is its global loss function based on bipartite matching, a contrast to traditional localized loss functions. However, its advanced Transformer architecture leads to longer training times.

In contrast, YOLOv8, the latest in the YOLO series, focuses on speed and efficiency, processing images in a single pass, ideal for real-time applications. It's faster and more real-time optimized, though it might involve complex multi-scale detection setups.

DETR offers a holistic, Transformer-based approach with global loss calculation, while YOLOv8 continues the YOLO series' legacy with fast, efficient detection. Both models cater to different needs in computer vision, highlighting diverse yet critical paradigms in object detection technology.

2.5 SORT and DeepSORT

As our research into suitable methods for vehicle counting continued, we came across the SORT5 algorithm, which demonstrated promising results upon its introduction. Following SORT, an extended version called DeepSORT6 was published. Both of these algorithms adhere to the "tracking by detection" paradigm, relying on bounding box detections provided by object detection algorithms, in our case, YOLOv8x. Both SORT and DeepSORT incorporate Kalman filters to predict the motion of objects and estimate their future positions based on past observations. The Kalman filter plays a crucial role in maintaining smooth and consistent object tracks.

However, DeepSORT takes a step further by incorporating deep appearance descriptor networks, often based on convolutional neural networks (CNNs). These networks extract discriminative appearance features that encode detailed visual information such as color, texture, and shape. According to the DeepSORT paper, this approach leads to improved tracking accuracy and robustness, particularly in scenarios involving occlusions and crowded scenes.

During our implementation of DeepSORT, we encountered a tracking issue. Occasionally, it generated two separate tracks for the same object, even if they belonged to the same class (e.g., "car," "truck," "motorbike," etc.). This issue primarily stemmed from YOLO's object detections. Although DeepSORT benefits from appearance features, it should logically consider these objects as a single entity, not two. Despite experimenting with various YOLO parameters, we couldn't resolve this issue.

To address this problem, we devised a solution involving the calculation of Euclidean distances between bounding boxes. The Euclidean distance formula in 2D is:

$$\text{Euclidean distance} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Given that the bounding boxes were in close proximity, we established a threshold, which should ideally be a small value. This threshold helped merge closely located bounding boxes into a single entity. However, a subtle yet critical bug emerged when two distinct vehicles approached each other in cluttered environments. The approach could erroneously remove one of the bounding boxes.

To mitigate this issue, we introduced a dynamic threshold equal to the minimum Euclidean distance between all bounding boxes. While this approach improved tracking, a new challenge surfaced in specific scenarios. When a vehicle partially left the frame, YOLO sometimes detected it as a different class of vehicle than before. DeepSORT, which keeps track of object history, would generate a new track for it based on this class change, even when the bounding boxes were very close. This posed a problem.

Ultimately, due to these tracking challenges and the unsatisfactory performance of the appearance descriptor in DeepSORT, we decided to transition to ByteTrack as our chosen method for vehicle counting.

2.6 ByteTrack

As previously mentioned, ByteTrack ranks among the top 5 available trackers and has significantly influenced the development of SMILEtrack, currently the leading tracker. ByteTrack is a fusion of the BYTE algorithm and YOLOX.

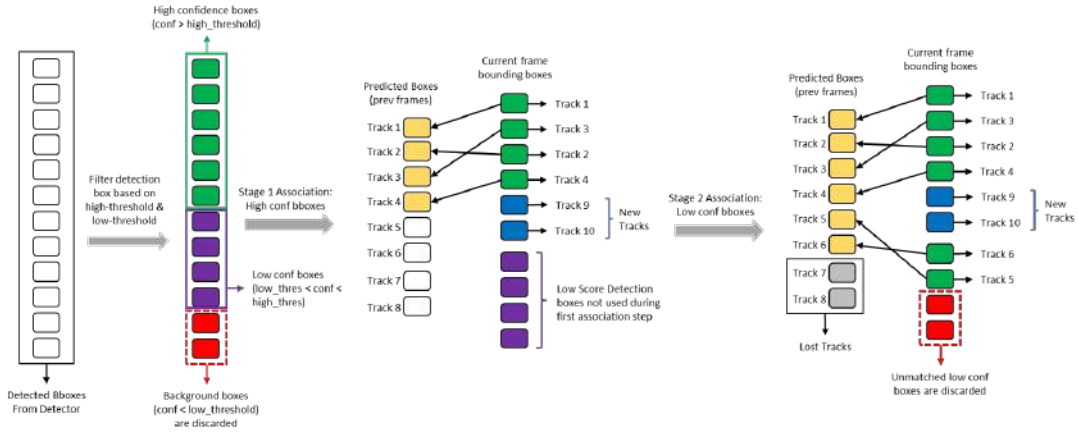

Figure 5: BYTE algorithm

The BYTE algorithm excels in establishing associations between detections and tracks. One of its notable strengths lies in its ability to handle low-confidence detection boxes. These boxes are treated as potentially occluded, motion-blurred, or challenging-to-identify objects. The accompanying figure, directly borrowed from the ByteTrack paper, illustrates its functionality:

- In all frames, the detection model accurately identifies people in the scene but occasionally generates false positives by recognizing portions of the background as objects.
- In the initial phase, BYTE aims to associate high-confidence detection boxes with previously initialized tracks. While this association proves reasonably effective, there are instances (e.g., frame t2 and t3) where it disregards detection boxes with low confidence.
- In the second phase, represented in section c, BYTE strives to associate low-confidence bounding boxes with the remaining unmatched tracks. During this phase, we observe the recovery of previously ignored objects with low confidence and the elimination of falsely detected background objects.

This feature equips BYTE with the capability to track occluded vehicles in complex and cluttered environments. Moreover, BYTE can recognize the same object even if it temporarily exits the frame and reappears later.

ByteTrack utilizes a Kalman filter to monitor object movement across frames. This filter enables ByteTrack to predict the future location of an object based

on its past motion patterns. It operates similarly to predicting the landing spot of a moving object based on its trajectory. Consequently, even if an object is not detected in a specific frame, ByteTrack can rely on its predictions to maintain tracking continuity.

Our own experiments confirm these capabilities. As depicted in the image below, the left frame shows the detection of a car assigned the ID number 5. In the subsequent frame (middle image), the same car becomes completely occluded by a blue car. Yet, in the following frame (right image), the car is detected once more and retains the same ID as before. This demonstrates ByteTrack's resilience in tracking objects under challenging conditions.

To provide a concise explanation of the Kalman filter, let's consider its application in tracking the position of a moving object. In our scenario, we're tracking vehicles instead of a ball. Here's a simplified overview of how the Kalman filter works:

- Initialization: We begin by initializing the Kalman filter with an initial estimate of the vehicle's position and velocity. Additionally, we define the uncertainties associated with these initial estimates.
- Prediction: Using the previous estimated position and velocity, we make a prediction for the vehicle's expected position in the next time step. This prediction accounts for the vehicle's motion model, assuming constant velocity.
- Measurement Update: When a new measurement arrives from our sensors, we compare it with our predicted position. We calculate the difference between the predicted and measured positions, taking into consideration measurement noise.
- Kalman Gain: The Kalman filter computes a parameter known as the Kalman gain. This parameter determines the weight or influence of the measurement in updating our estimate. It considers the uncertainties associated with both the predicted state and the sensor measurements.
- State Update: Utilizing the Kalman gain, we update our estimate of the vehicle's position and velocity based on the measurement. This update blends the predicted state with the measurement, giving more weight to the more reliable information.
- Iteration: The process of prediction, measurement update, and state update is repeated for each new measurement, iteratively enhancing the accuracy of our estimation as we receive new data.

The Kalman filter excels at estimating the true state of the system by considering both predictions and measurements. It effectively manages noisy measurements and provides a robust estimation that balances the influence of predictions and measurements. In our project, the Kalman filter's role is to predict the movement of vehicles, and while the overall procedure remains the same, the specific mathematical details can be found in the appendix for further reference.

2.7 Adaptive Histogram Equalization (AHE)

To enhance the performance of both YOLO and ByteTrack in our project, we made the decision to improve the contrast of frames within our video. Adaptive Histogram Equalization (AHE) emerged as a valuable image enhancement technique for enhancing contrast and improving the visibility of image details. Its effectiveness lies in the redistribution of pixel intensities across different regions of the image.

Let's delve into the concept of adaptive histogram equalization as it applies to our experiment. Images often contain various regions with distinct levels of brightness and contrast. Some regions may appear excessively dark, while others may be overly bright, hindering the clear visibility of details.

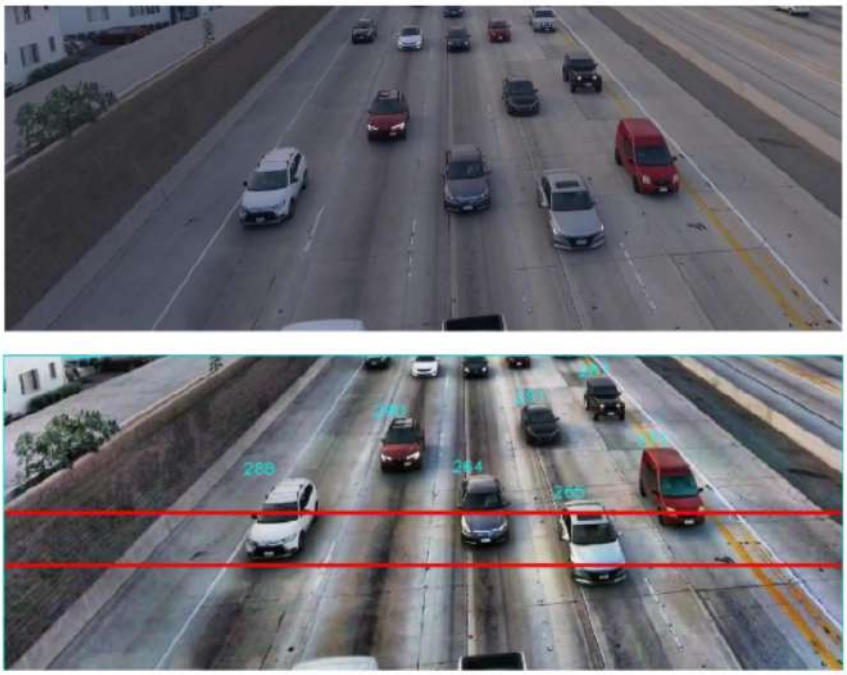

Figure 6: Before and after applying AHE. In the bottom image, the whole tracking process is carried out, and the two red lines and the number above each vehicle is their track id

In the context of adaptive histogram equalization, the image is divided into smaller sub-regions referred to as "tiles" or "blocks." For each tile, a histogram of pixel intensities is computed, representing the distribution of pixel values within that specific region.

Subsequently, the histogram is equalized for each tile independently. Equalization entails the redistribution of pixel intensities to achieve a more uniform

distribution across the intensity range. This process effectively stretches the contrast and enhances the details within each tile.

Crucially, adaptive histogram equalization takes into account the local characteristics of each tile. Rather than applying a uniform equalization process to the entire image, it adapts the equalization approach based on the content of each individual tile. This localized enhancement process contributes to improved contrast and better visibility of image details, which is advantageous for our vehicle tracking project.

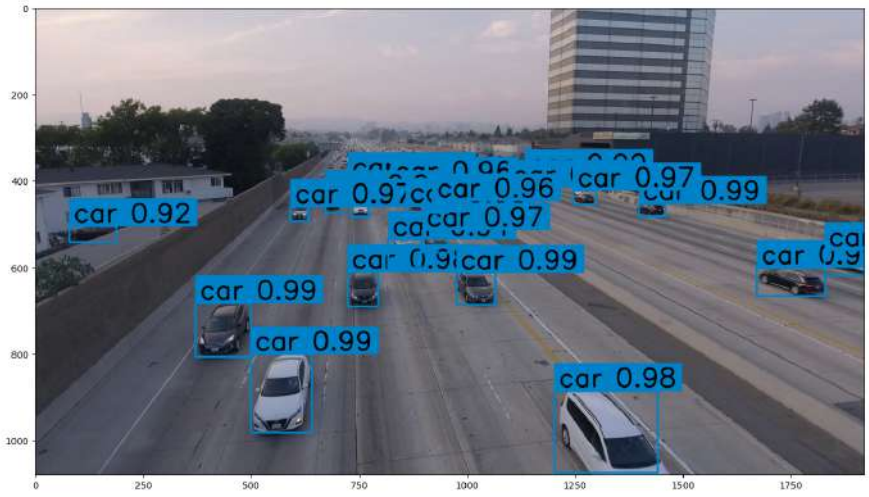

Figure 7: DEtection TRansformers output with label and confidence score

3 Description of the code implementation

Let us explain how the code works step by step.

To begin, we incorporate the necessary libraries and customize our system path. This adjustment is necessary because we are utilizing a local version of ByteTrack. The original version was not readily executable as-is, prompting the need for modifications.

Next, we create a class to manage ByteTrack arguments, which is designed to accept arguments in the form of an object. These arguments serve specific purposes in our project. Refer Figure 8:

- **track-thresh** This parameter sets the threshold value for tracking. It determines when a track should be initiated.
- **track-buffer** Represents the buffer size for tracks, indicating how many

```
class BYTETrackerArgs:
    def __init__(self, track_thresh, track_buffer, mot20, match_thresh, \
                 aspect_ratio_thresh, min_box_area):
        self.track_thresh        = track_thresh
        self.track_buffer        = track_buffer
        self.mot20               = mot20
        self.match_thresh        = match_thresh
        self.aspect_ratio_thresh = aspect_ratio_thresh
        self.min_box_area        = min_box_area
```

Figure 8: BYTETrackerArgs Class

previous frames of track history should be stored and considered during the association process.

- **mot20** While related to the MOT20 dataset, this variable is not directly relevant to our project's objectives. However, it still needs to be defined.
- **match-thresh** The match-thresh parameter establishes a threshold value for matching, defining the required similarity level for matching a detection to an existing track.
- **aspect-ratio-thresh** This parameter defines a threshold for the aspect ratio of a bounding box. It helps filter out bounding boxes with aspect ratios exceeding a specified threshold.
- **min-box-area** The min-box-area parameter sets a minimum area threshold for bounding boxes. It is employed to filter out bounding boxes with areas smaller than the specified threshold.

```
def countVehicles(video_path, output_file_name, vertical, roi_xxyy=(0,0,0,0)):
    frames_list = []

    assert type(video_path)       == str, "video_path argument should be string"
    assert type(output_file_name) == str, "output_file_name argument should be string"
    assert type(vertical)         == bool, "vertical argument should be boolean"

    args = BYTETrackerArgs(track_thresh = 0.25,
                           track_buffer = 30,
                           mot20 = False,
                           match_thresh = 0.8,
                           aspect_ratio_thresh = 3.0,
                           min_box_area = 1.0)

    obj_tracker = BYTETracker(args)
```

Figure 9: CountVehicles Function

These argument settings are tailored to suit the requirements of our vehicle tracking project.

We've meticulously outlined a step-by-step pipeline to ensure the proper execution of our project. Let's break down the key components of this pipeline. Refer Figure 9:

- **video-path** This parameter signifies the path to the video that needs processing.
- **output-file-name** It's the designated name for the output video file where our results will be saved.
- **vertical** A boolean value indicating the approximate direction of vehicle movement. We employ different criteria for vertical and horizontal movements.
- **roi-xxyy** "ROI" stands for "Region Of Interest," and this argument defines the coordinates of the top-left and bottom-right points of the ROI. It should be in the format xxyy, where the first two numbers represent the x-coordinates, and the last two numbers represent the y-coordinates. The default value is set to (0, 0, 0, 0) to use default values if no specific ROI is provided to the pipeline.

```
if not vertical:
    areaLine1   = x_starting_point + int((x_ending_point - x_starting_point)/2) - 15
    areaLine2   = x_starting_point + int((x_ending_point - x_starting_point)/2) + 15
else:
    areaLine1   = y_ending_point - 150
    areaLine2   = y_ending_point - 100
```

Figure 10: the code

Following this, we define the arguments for ByteTrack, selecting settings that have demonstrated effectiveness in similar projects. Additionally, we establish areaLine1 and areaLine2 as two red lines oriented differently in the scene. If vehicles are moving horizontally, we position the area lines in the middle with a gap of 30 pixels. For vertical vehicle movement, the area lines are placed near the bottom of the frame with a 50-pixel gap.

We integrate Adaptive Histogram Equalization (AHE) using the OpenCV library in Python. It's worth noting that since we don't need to process the entire frame, AHE is selectively applied to the defined ROI. Refer Figure 12

Figure 11: The two area lines in vertical and horizontal orientations

```
# apply adaptive histogram equalization (AHE) in order to increase the contrast in our region of interest.
clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(8,8))

R, G, B = cv2.split(frame[y_starting_point:y_ending_point, x_starting_point:x_ending_point]) # we don't need to
                                                                                            # apply AHE
                                                                                            # to the whole
                                                                                            # frame


cl1 = clahe.apply(R)
cl2 = clahe.apply(G)
cl3 = clahe.apply(B)

orig_frame       = frame.copy() # we take a copy of our original frame before loosing it.
frame            = cv2.merge((cl1, cl2, cl3))
frame_h, frame_w = frame.shape[:2]
frame_size       = np.array([frame_h, frame_w])
orig_frame[y_starting_point:y_ending_point, x_starting_point:x_ending_point] = frame # we replace the region of
                                                                                    # interest with
                                                                                    # the enhanced version of
                                                                                    # it.
```

Figure 12: Adaptive Histogram Equalization(AHE) Code

```
for result in res:
    for box, r in zip(result.boxes, result.boxes.data):
        x, y, w, h = box.xywh[0]
        # we add x_starting_point and y_starting_point to x and y coordinate because we shrinked the frame earlier
        x1, y1, x2, y2 = int(x) + x_starting_point - int(w/2), int(y) + y_starting_point - int(h/2),\
                         int(x) + x_starting_point + int(w/2), int(y) + y_starting_point + int(h/2)

        # if class of the detected object is not vehicle then discard it
        if r[-1] > 0 and r[-1] < 8:
            xyxys.append([x1, y1, x2, y2, r[-2]]) # xyxy and score
        confss.append(r[-2])
        oids.append(r[-1]) # class of the detected object


if len(xyxys) > 0:
    tracks = obj_tracker.update(np.array(xyxys), frame_size, frame_size)
else:
    tracks = np.array([])
```

Figure 13: the code

Moving forward, we iterate through the bounding boxes, extracting confidence scores and object IDs, and pass them to the tracker. Once we have obtained the tracks, we annotate each bounding box with its corresponding ID. Several conditions are then imposed: the first verifies if the bounding box lies within the area lines; the second ensures that a track is counted only once, as

the process is repeated in each frame; the last condition dictates counting tracks with a confidence score surpassing a specified threshold. If all conditions are met, the track ID is added to the list of IDs.

```
for track in tracks:

    cv2.putText(orig_frame, str(track.track_id), (int(track.tlbr[0]), int(track.tlbr[1])),cv2.FONT_HERSHEY_SIMPLEX,0.5, [255, 255, 0], thickness=1, lineType=cv2.LINE_AA)
    if not vertical:
        conditions = ((track.tlbr[0] > areaLine1 and track.tlbr[0]< areaLine2) and # upper left corner of the bbox should be in the area
                      track.track_id not in ids and
                      track.score > 0.6)
    else:
        conditions = ((track.tlbr[1] > areaLine1 and track.tlbr[1]< areaLine2) and # upper left corner of the bbox should be in the area
                      track.track_id not in ids and
                      track.score > 0.6)

    if (conditions):

        cv2.putText(orig_frame, str(track.track_id), (int(track.tlbr[0]), int(track.tlbr[1])),cv2.FONT_HERSHEY_SIMPLEX,0.8, [0, 255, 0], thickness=2, lineType=cv2.LINE_AA)
        ids.append(track.track_id)
```

Figure 14: the code

Subsequently, we draw green rectangles around the counted vehicles and display the vehicle being counted above the screen to facilitate visibility. Finally, we save the video file and allow the user to interrupt the process at any time by pressing the "Escape" key on the keyboard.

```
    n = 20   # starting row for displaying the counted vehicle image in the original frame
    m = 250  # starting column for displaying the counted vehicle image in the original frame

    for track in tracks:
        if (track.track_id in ids):
            cv2.rectangle(orig_frame, (int(track.tlbr[0]),int(track.tlbr[1])), (int(track.tlbr[2]),int(track.tlbr[3])), (0,255,0), 2)
            cv2.rectangle(orig_frame, (m, n), (m+int(track.tlwh[2]), n+int(track.tlwh[3])), (0,255,0), 2)
            try:
                orig_frame[n:n+int(track.tlwh[3]), m:m+int(track.tlwh[2])] = \
                    orig_frame[int(track.tlwh[1]):int(track.tlwh[1])+int(track.tlwh[3]), int(track.tlwh[0]):int(track.tlwh[0])+int(track.tlwh[2])]
                m += int(track.tlwh[2])+5
            except:
                print("Error!")

    # drawing our RoI (Region of Interest)
    cv2.rectangle(orig_frame, (x_starting_point, y_starting_point), (x_ending_point,y_ending_point), (255,255,0), 1)

    writer.write(orig_frame)
    cv2_imshow(orig_frame) #('frame',)

    frames_list.append(orig_frame)
    # press esc for quitting the video
    if cv2.waitKey(1) & 0xFF == 27:
        break
else:
    break
```

Figure 15: the code

To initiate the entire pipeline, a single function call to countVehicles is all that's required. Our implementation has undergone testing on three distinct videos, and the file names are provided with comments for reference. Please ensure that each file is run with the appropriate "vertical" parameter.

```
# options for videos:
# vehicles moving vertically  (vertical: True): los_angeles.mp4, highway.mp4
# vehicles moving horzintally (vertical: False): driving1.mp4
countVehicles( '/content/drive/MyDrive/Semester 3/CV/Project/CV course project/los_angeles.mp4',
               '/content/drive/MyDrive/Semester 3/CV/Project/CV course project/driving1.mp4', True)
```

Figure 16: Function *countVehicles:* (InputFilePath, OutputFilePath, Horizontal(or)Vertical)

4 Conclusion

In summary, the task of counting vehicles in a video or live stream from a camera poses significant challenges due to various complex scenarios that can arise, including occlusions, motion blurs, low-quality cameras, and more. Our approach involved exploring different methods to address these challenges, ultimately leading us to select YOLOv8 and ByteTrack as the ideal combination for our project. While this combination proves effective in many situations, it does have some limitations that we aim to address in the future. We also saw results of DETR, where it is unable to identify a Truck, whereas the YOLOv8 model was able to do it.

One notable limitation is that ByteTrack struggles with tracking small objects, often losing track when such objects are momentarily occluded. Additionally, difficulties arise when there are two different cars of the same brand, with the same color, and moving in the same direction. In these scenarios, ByteTrack can become confused, indicating a need for further refinement in these specific situations.

5 Codes and Videos

This Link contains the zip file{CVProject-zip-file} which contains the relevant *.ipynb* files and *.mp4* video files, on which we tested the Counting and Tracking Model. A READme file has been created for the user to understand the working of the code.

References

[1] Dillon Reis, Jordan Kupec, Jacqueline Hong, Ahmad Daoudi, *"Real-Time Flying Object Detection with YOLOv8"*, (JAN 2023),Georgia Institute of Technology, USA.

[2] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, Sergey Zagoruyko, *"End-to-End Object Detection with Transformers"*, (2020),Facebook Research Team, USA.

[3] Yifu Zhang, Zehuan Yuan, *"ByteTrack: Multi-Object Tracking by Associating Every Detection Box"*, (OCT 2021),The University of Hong Kong,China.