Image Analysis and Computer Vision

Professor Vincenzo Caglioti

Object Tracking and Vehicle Counting

(Full Project)

2nd Semester – 2022-2023 A.Y.

Milad Goudarzi

10765491

POLITECNICO

MILANO 1863

Contents

1. Introduction

1.1. What is the project all about?

Our project is about counting vehicles in a video. In first glance, it looks like detecting vehicles and counting them would lead us to what we want. However, this is true only for images and not videos. In videos, the whole process, i.e., object detection, will be repeated on each frame and therefore if our video has say 30 frames per second, each vehicle will be counted 30 times and it is obviously wrong. So, in addition to object detection, we need to have a technique to determine if we have already counted a vehicle and this could be achieved by tracking vehicles.

Object tracking is following and monitoring the movement of an object in a video or series of images like camera streams. Tracking objects could be quite a complex problem in some situations, e.g., when occlusion happens, or when the object is motion blurred, or even when the object leaves the frame completely and then comes into the picture after a few frames. Many techniques were developed to address this problem and we will have a look at some of them throughout this report.

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

It is always best to study what are state-of-the-art methods and get inspired by them. As of now, the best available tracker is SMILEtrack[1] according to MOT17 and MOT20 benchmarks (figure 1). SMILEtrack is proposed on Nov. 2022, and it proposes a Similarity Learning Module (SLM) motivated from the Siamese network to extract important object appearance features and a procedure to combine object motion and appearance features effectively. However, there is a really slight difference (less than 1%) between the top 5 trackers. SMILEtrack authors say a large portion of their work is borrowed from ByteTrack and YOLOv7 and it is of no surprise that ByteTrack is among top 5 available trackers and since we do not want to naively copy and paste SMILEtrack developers job, and also, since ByteTrack helped SMILEtrack to become the best tracker, it is quite rational to decide to jump into ByteTrack and YOLO details and try to combine them on our own because a newer version of YOLO was made available after the publication of SMILEtrack paper.

[1] SMILEtrack: SiMIlarity LEarning for Multiple Object Tracking (https://arxiv.org/pdf/2211.08824v2.pdf)

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 (https://paperswithcode.com/sota/multi-object-tracking-on-mot20-1). Right: MOT17 ranking of trackers (https://paperswithcode.com/sota/multi-object-tracking-on-mot17). Highlighted rows are the two discussed trackers.

2. Our approach

2.1. Cascade classification

Before trying to mess with ByteTrack, we tried a few other possibilities to see their shortcomings. Firstly, we tried to use a cascade classifier trained on vehicle images. The cascade classifier is composed of multiple stages, where each stage consists of several weak classifiers. The term "weak" refers to the fact that these classifiers alone may not be very accurate in detecting objects, but they are computationally inexpensive. It was not the worst thing ever, but it had lots of false negatives even on a single image as it is also seen from other experiments, let alone complicated conditions like occlusion.

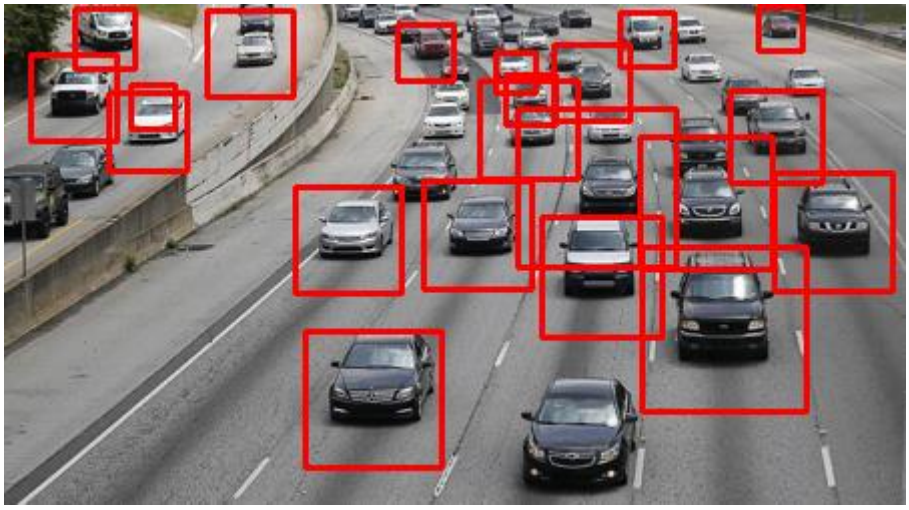


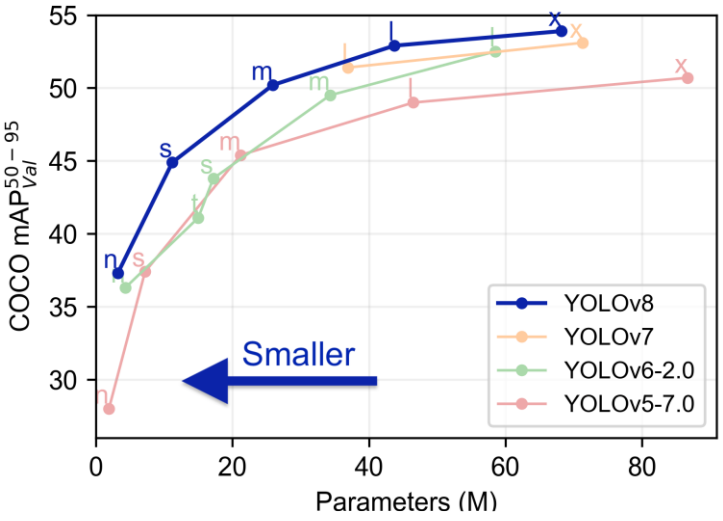
Figure 2. Cascade classifier on a single image. A lot of false negatives are present in the results.

2.2. YOLO

After doing some more research we found that YOLO[2] object detection model is a popular and widely used model to this end. YOLO is an acronym for "You Only Look Once" and we will see why it is called this way. YOLO is fast, has a simple architecture, and it does not use region-proposal methods for finding objects, as previous methods like R-CNN[3], Faster R-CNN[4]. These methods first generate a set of region proposals or candidate object regions in the image using techniques like selective search or region proposal networks. These proposals are then classified and refined to obtain the final object detections. Region-proposal methods typically involve additional processing steps, which can increase the overall computation time. YOLO, in contrast, predicts bounding boxes (regions) and class probabilities in a single shot and hence the name "You Only Look Once". Furthermore, it is common for object detection networks to generate multiple bounding boxes for a single object. YOLO employs Non-Maxima Suppression (NMS), which leverages the intersection over union metric, to eliminate redundant boxes. Each predicted bounding box has a confidence score and NMS removes redundant boxes by following these steps:

1. Overlapping Regions: It determines the overlap between bounding boxes.
2. Sorting: It sorts bounding boxes by confidence score.
3. Highest Confidence Box: It selects the box with the highest confidence as the reference.
4. Intersection over Union (IoU): It measures overlap using IoU.
5. Thresholding: It sets a minimum IoU threshold (e.g., 0.5).
6. Suppression: It discards overlapping boxes that exceed the threshold.
7. Iterative Process: It repeats steps 3-6 with the next highest confidence box.
8. Final Bounding Boxes: The remaining non-overlapping boxes are the final predictions.

In addition, YOLO has a few different versions and each of them has a different number of subversions. These subversions benefit from different number of parameters and hence, different accuracy and speed.



[2] You Only Look Once: Unified, Real-Time Object Detection (https://arxiv.org/abs/1506.02640)

[3] Rich feature hierarchies for accurate object detection and semantic segmentation (https://arxiv.org/abs/1311.2524v5)

[4] Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks (https://arxiv.org/abs/1506.01497)

The various versions of YOLO models are compared to one another using the mAP metric, utilizing IoU thresholds ranging from 50 to 95 with a step size of 5. mAP is the mean Average Precision which is calculated according to the following formula:

$$\text{mAP} = \frac{1}{n}\sum_{k=1}^{n} AP_k$$

AP_k = Average Precision of class k

n = the number of classes

Within this equation, we begin by determining distinct precisions for each class by applying various IoU thresholds. This involves categorizing a detection as a true positive when the IoU metric exceeds the specified threshold between the predicted box and the actual box. Once we have computed these precisions for a particular class, we calculate their average and repeat the process for other classes. Ultimately, we derive the mean of these Average Precisions across all classes, yielding the mAP value for the respective model.

Due to the higher mAP of YOLOv8 version x compared to other versions, we selected it as our detector. This means that we prioritized accuracy over speed, as we were not required to process videos in real-time.

2.3. SORT and DeepSORT

As our research in finding popular methods for our purpose (counting vehicles) kept going, we found out SORT[5] algorithm did a good job when it was introduced. After SORT, an extension of it, i.e., DeepSORT[6], was published. Both algorithms follow the "tracking by detection" paradigm, where they rely on bounding box detections provided by object detection algorithms as input, which in our case was coming from YOLOv8x. Both SORT and DeepSORT integrate Kalman filter to predict motion of objects and estimate their future positions based on past observations. The Kalman filter helps maintain smooth and consistent tracks. However, DeepSORT enhances SORT by incorporating deep appearance descriptor networks, typically based on convolutional neural networks (CNNs), to extract discriminative appearance features. These appearance features encode detailed visual information such as color, texture, and shape. According to the DeepSORT paper, it achieves better tracking accuracy and robustness in scenarios with occlusions and crowded scenes. In our case, when we implemented DeepSORT, we faced some issues with the tracking. It sometimes generated two different tracks for the same object, i.e., two tracks of the same class, e.g., “car”, “truck”, “motorbike”, etc. This issue was mainly coming from YOLO but since the tracker benefits from appearance features, it is expected that it considers both as a single object and not two. After playing around with different parameters of YOLO and not succeeding in resolving this issue, finally, we came up with the idea of calculating the Euclidean distance between the two bounding boxes.

Euclidean distance of two vectors in 2d = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Since the two bounding boxes were close to each other, we could decide for a threshold, which rationally must be a small number because otherwise we could combine two different bounding boxes from two different vehicles into a single one and define a condition like “if the distance

[5] Simple Online and Realtime Tracking (https://arxiv.org/abs/1602.00763)

[6] Simple Online and Realtime Tracking with a Deep Association Metric (https://arxiv.org/abs/1703.07402)

was less than lambda, keep one of them". But this approach had a really small, yet important bug. The bug becomes prominent when two distinct vehicles come close to each other, which is usually the case with cluttered environments, and the described approach would lead to a false decision for removing one of the bounding boxes. So, we decided to change our criteria and go for a dynamic threshold and put it equal to the minimum Euclidean distance of all bounding boxes from one another. This approach helped a lot but suddenly another problem arose. In really specific scenarios, i.e., when the vehicle was leaving the frame and was partially present in the frame, YOLO would detect it to be from another class of vehicles with respect to what it was before and since the tracker keeps the history of its tracks in memory and knows that this vehicle was for instance a "car" and not a "truck", when YOLO predicts an object to be a "truck", the tracker would generate a new track for it, even if the bounding boxes are super close because they have different classes.

Figure 3. DeepSORT tracker generates two different tracks for the same object with even two super close bounding boxes.

At this point we decided to leave DeepSORT as its appearance descriptor did not satisfy us and we moved on to ByteTrack.

2.4. ByteTrack[7]

As mentioned earlier, ByteTrack is among the top 5 available trackers until now and had a great impact on SMILEtrack which is the best tracker available. ByteTrack is the combination of the so-called BYTE algorithm and YOLOX.

BYTE demonstrates an effective association between detections and tracks. One notable advantage of BYTE is its ability to handle low confidence detection boxes by treating them as potentially occluded or motion-blurred, or in any case hard-to-recognise objects.

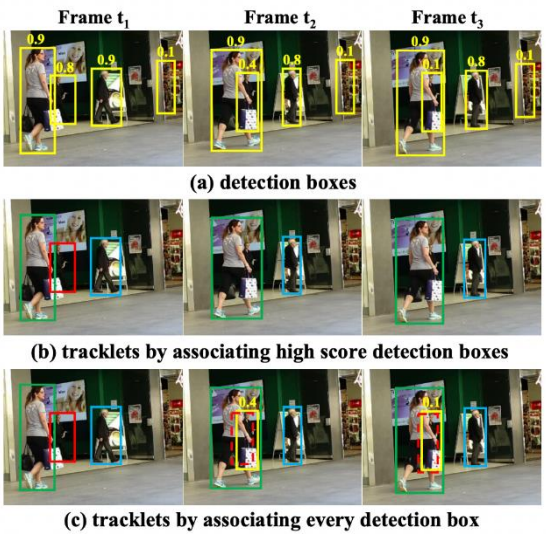
The figure on the right, which is borrowed directly from ByteTrack paper, demonstrates:

(a) In all frames, the detection model successfully identifies people in the scene but also generates false positives by detecting a portion of the background as an object.

[7] ByteTrack: Multi-Object Tracking by Associating Every Detection Box (https://arxiv.org/abs/2110.06864)

(b) In the first phase, BYTE attempts to associate high-confidence detection boxes with previously initialized tracks. This association appears reasonably effective, but in frame t2 and t3, it disregards the red bounding box due to its low confidence.

(c) In the second phase, represented in section c, BYTE endeavors to associate low-confidence bounding boxes with the remaining unmatched tracks. In this phase, we observe the recovery of the red object, which was previously ignored, and the elimination of the falsely detected background object.



This feature enables BYTE to track occluded vehicles in complex environments with clutter. Furthermore, BYTE has the capability to recognize the same object even if it temporarily leaves the frame and reappears later. ByteTrack incorporates the use of a Kalman filter to monitor the movement of objects across frames. This filter enables ByteTrack to predict the future location of an object based on its previous motion patterns, similar to anticipating the landing spot of a ball based on its current trajectory. As a result, even if an object is not detected in a particular frame, ByteTrack can utilize its predictions to maintain tracking of the object.

We have also observed it in our experiments. As you can see in the picture below, in the left image a car is detected and ByteTrack has assigned id number 5 to it. In the very next frame, i.e., the image in the middle, that car is occluded completely by the blue car. Again, in the next frame, which is the image on the right, that car is detected once more and is assigned the same id as before.

Figure 4. Three consecutive frames of a video being processed by YOLOv8x (for detection) and ByteTrack.

To briefly explain the Kalman filter, let's consider a simple example of tracking the position of a moving object. Suppose we have a sensor that measures the position of a ball as it moves horizontally along a straight line. However, the sensor measurements are noisy and may contain errors. The Kalman filter helps us estimate the true position of the ball by combining the noisy measurements with predictions based on the ball's previous position and velocity. Here's how it works:

1. Initialization: We start by initializing the filter with an initial estimate of the ball's position and velocity. We also define the uncertainty or noise associated with these initial estimates.

2. Prediction: Based on the previous estimated position and velocity, we predict the expected position of the ball in the next time step. This prediction takes into account the motion model of the ball, assuming it moves at a constant velocity.

3. Measurement Update: When a new measurement from the sensor arrives, we compare it with our predicted position. We calculate the difference or "residual" between the predicted and measured positions, accounting for the measurement noise.

4. Kalman Gain: The Kalman filter calculates a parameter called the Kalman gain, which determines the weight or influence of the measurement in updating our estimate. It takes into account the uncertainties of both the predicted state and the sensor measurements.

5. State Update: Using the Kalman gain, we update our estimate of the ball's position and velocity based on the measurement. This update combines the predicted state with the measurement, giving more weight to the more reliable information.

6. Iteration: The process of prediction, measurement update, and state update is repeated for each new measurement, iteratively improving the estimation of the ball's position as new data becomes available.

The Kalman filter considers both the predicted state and the measurement to estimate the true state of the system. It handles noisy measurements and provides a robust estimation that balances the influence of the prediction and the measurement. In our use case, the Kalman filter's job is to predict movement of vehicles, instead of a ball. The overall procedure is the same, but for the mathematical details of it, you can check the appendix.

2.5. Adaptive Histogram Equalization (AHE)

What is more, in order to help YOLO and ByteTrack, we decided to increase the contrast of frames in our video. Adaptive histogram equalization is an image enhancement technique that improves the contrast and visibility of details in an image. It achieves this by redistributing the pixel intensities in different regions of the image.

To understand adaptive histogram equalization, consider our experiment. The image contains different regions with varying levels of brightness or contrast. Some regions might appear too dark, while others might be too bright, making it difficult to see the details clearly.

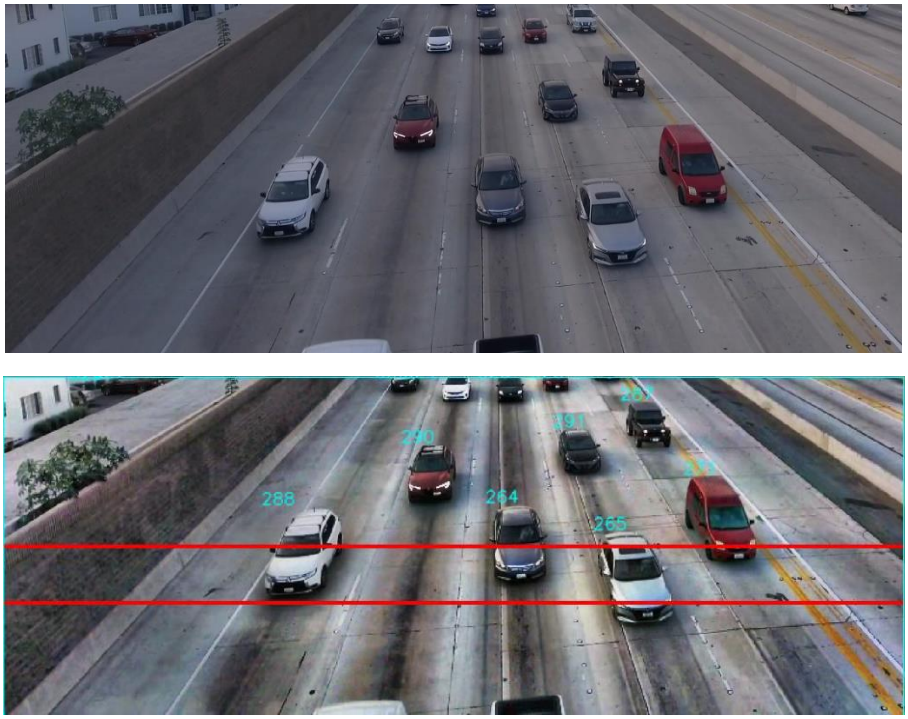


Figure 5. Before and after applying AHE. In the bottom image, the whole tracking process is carried out, and the two red lines and the number above each vehicle is their track id.

In adaptive histogram equalization, the image is divided into smaller sub-regions called "tiles" or "blocks." For each tile, a histogram of the pixel intensities is calculated, which shows the distribution of pixel values in that particular region.

Next, the histogram is equalized for each tile individually. Equalization means redistributing the pixel intensities to spread them out more evenly across the intensity range. This process helps to stretch the contrast and bring out the details in each tile.

However, one important aspect of adaptive histogram equalization is that it takes into account the local characteristics of each tile. Instead of applying a global equalization to the entire image, it adapts the equalization process based on the content of each tile.

3. Description of the code implementation

Let us explain how the code works step by step.

```
import sys
sys.path.append(".\ByteTrack")
from ultralytics import YOLO
import numpy as np
import cv2
from collections import deque
from math import sqrt
from deep_sort_pytorch_2nd.utils.parser import get_config
from deep_sort_pytorch_2nd.deep_sort import DeepSort
import torch
from yolox.tracker.byte_tracker import BYTETracker

# YOLO with less parameters
# model = YOLO(model='yolov8n.pt')

# YOLO with large number of parameters
model = YOLO('YOLOv8x.pt')

```

Figure 6. the code

First, we import the required libraries and add our specific system path because we are using a local version of ByteTrack, since the original one could not be run right out of the box.

```
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 7. the code

Then we define a class for ByteTrack arguments, as it is designed in a way to receive arguments as an object only. These arguments are described as follows:

- track_thresh: Represents the threshold value for tracking. It is a value used to determine if a track should be initiated.
- track_buffer: Represents the buffer size for tracks. It determines how many past frames of track history should be stored and considered during the association process.
- mot20: A variable related to the MOT20 dataset or functionality specific to it. We do not really need it because we are not working with this dataset. However, we need to define it.
- match_thresh: Represents the threshold value for matching. It is a value used to determine the similarity required for matching a detection to an existing track.
- aspect_ratio_thresh: Represents the threshold value for the aspect ratio of a bounding box. It is used to filter out bounding boxes that have aspect ratios above a certain threshold.
- min_box_area: Represents the minimum area threshold for a bounding box. It is used to filter out bounding boxes that have an area smaller than the specified threshold.

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

    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 8. the code

We have defined a pipeline for our project in order for everything to happen in the right order.

- video_path: It is the path to the video which is going to be processed.
- out_put_file_name: This is the name of the video file we are going to save our results in.
- vertical: It is a Boolean, indicating the approximate direction of vehicles movement. We consider 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 xxyy format, meaning that the first two numbers are the two x-coordinates and the last two numbers are the two y-coordinates. We consider the default value of (0, 0, 0, 0) because if no special roi is passed to the pipeline, we want to use the default values.

Then, we define the arguments of the ByteTrack, which are shown to be working best in similar projects.

```
76            if not vertical:
77                areaLine1   = x_starting_point + int((x_ending_point - x_starting_point)/2) - 15
78                areaLine2   = x_starting_point + int((x_ending_point - x_starting_point)/2) + 15
79            else:
80                areaLine1   = y_ending_point - 150
81                areaLine2   = y_ending_point - 100
```

Figure 9. the code

areaLine1 and areaLine2 are the two red lines in different scene orientations. If vehicles are moving horizontally in the scene, we want the area lines to be in the middle with a gap of 15+15=30 pixels. If the vehicles move vertically, we want the area lines to be placed close to the bottom of the frame with a 50 pixels gap between them.

Figure 10. The two area lines in vertical and horizontal orientations.

```
83            # apply adaptive histogram equalization (AHE) in order to increase the contrast in our region of interest.
84            clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(8,8))
85
86            R, G, B = cv2.split(frame[y_starting_point:y_ending_point, x_starting_point:x_ending_point]) # we don't need to
87                                                                                                        # apply AHE
88                                                                                                        # to the whole
89                                                                                                        # frame
90
91            cl1 = clahe.apply(R)
92            cl2 = clahe.apply(G)
93            cl3 = clahe.apply(B)
94
95            orig_frame       = frame.copy() # we take a copy of our original frame before loosing it.
96            frame            = cv2.merge((cl1, cl2, cl3))
97            frame_h, frame_w = frame.shape[:2]
98            frame_size       = np.array([frame_h, frame_w])
99            orig_frame[y_starting_point:y_ending_point, x_starting_point:x_ending_point] = frame # we replace the region of
100                                                                                               # interest with
101                                                                                               # the enhanced version of
102                                                                                               # it.
```

Figure 11. the code

Here, we applied the AHE, using OpenCV library for python. The point here is that since we do not need to process the whole frame, then we also do not need to apply AHE to the whole frame. That is why we defined roi. Therefore, we apply AHE on the roi only.

```
            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 earli
                    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 12. the code

Here, we looped through the bounding boxes and separated confidence scores and object ids and if there were any, we pass them to the tracker.

```
            for track in tracks:

                cv2.putText(orig_frame, str(track.track_id), (int(track.tlbr[0]), int(track.tlbr[1])),cv2.FONT_HERSHEY_SIMPL
                if not vertical:
                    conditions = ((track.tlbr[0] > areaLine1 and track.tlbr[0]< areaLine2) and # upper left corner of the bb
                                  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 bb
                                  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_S
                    ids.append(track.track_id)

```

Figure 13. the code

After having the tracks, we write the id of each bounding box above it and then we define some conditions. The first condition determines if the bounding box is inside the area lines. The second condition determines if we have counted a track before. Since this process is repeated in each frame, each track would be counted more than once. So, by this condition, we ensure they are counted only once. The last condition enforces to count the tracks with a confidence score more than a threshold. If all these conditions get satisfied, we add the track id to the ids list.

```
            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]
                    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(trac
                        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('frame', orig_frame)

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

Figure 14. the code

Then, we draw a green rectangle around the counted vehicles and then show the vehicle being counted above the screen, just to make it easier to see which vehicles are counted and which ones are not. In the end, we save the video file and then make the user to stop the process any time by pressing escape from keyboard.

```
# options for videos:
# vehicles moving vertically  (vertical: True): los_angeles.mp4, highway.mp4
# vehicles moving horzintally (vertical: False): driving1.mp4
countVehicles('highway.mp4', 'test3.mp4', True)
```

Figure 15. the code

We can run the whole pipeline just by calling the function `countVehicles`. We have tested our implementation on three different videos, and they are all commented to let you know what the file names are, just in case someone wants to try it out. Just remember that each file should be run with the correct “vertical” parameter.

4. Conclusion

Overall, counting vehicles in a video or live stream camera could be a pretty challenging task to do due to complicated situations that might happen, such as occlusions, motion blurs, low quality camera, etc. We tried to tackle these challenges by utilizing different approaches, which in the end lead us to choose YOLOv8 and ByteTrack as our perfect combination. Although this combination works well in most situations, it has some shortcomings which we hope to be able to solve in future. For instance, ByteTrack has a hard time keeping track of small objects, meaning that it loses track when small objects are occluded even for a short time. It also experiences difficulties when there are two different cars from the same brand but with the same color and the same moving direction (moving direction is important because of Kalman Filter). In these settings, ByteTrack gets confused and there is still work to do in the mentioned situations.