



Northeastern University

FINAL PROJECT REPORT

A REAL-TIME VEHICLE ENUMERATION AND CLASSIFICATION SYSTEM

Group II

Pragya Avinash Mishra, Sai Prashanth Reddy Machannagari,

Srinidheesh Ranganathan,

Yashwanth Balan Arumugam,

Northeastern University, College of Professional Studies,

EAI 6080 Advance Analytical Utilization

Fall 'A 2022

Prof. Mimoza Dimodugno

October 29, 2022

Table of Contents

S. No	Content	Page No
1.	Abstract	3
2.	Problem Statement	4
3.	Introduction	4
4.	Dataset Description	6
5.	Methodology	7
A.	Pre-requisites	7
B.	Architecture and Development	8
C.	Design and Implementation	11
D.	Results and Demonstration	23
6.	Challenges, Recommendations and Solutions	27
7.	Conclusion	32
8.	References	33
9.	Appendix	34

Abstract

Intelligent vehicle detection and counting are becoming more important in highway management. However, due to the various sizes of vehicles, detection remains a challenge, which has a direct impact on the accuracy of vehicle counts. This paper proposes a vision-based vehicle detection and counting system to address this issue. This study publishes a new high-definition highway vehicle dataset with 328K images. In comparison to existing public datasets, the proposed dataset contains annotated tiny objects in the image, providing a complete data foundation for deep learning-based vehicle detection. The highway road surface in the image is first extracted and divided into a remote area and a proximal area by a newly proposed segmentation method in the proposed vehicle detection and counting system; the method is critical for improving vehicle detection. The above two areas are then fed into the YOLOv3 network to detect the vehicle type and location. Finally, the ORB algorithm is used to obtain vehicle trajectories, which can be used to judge the driving direction of the vehicle and determine the number of different vehicles. Several highway surveillance videos from various scenes are used to validate the proposed methods.

II. Problem Statement:

The mistake of continual backdrop updating and the challenge of regulating the update pace in traffic video locations where moving vehicles are present are the problems with getting the initial background. Additionally, given the increasing number of streets and traffic worldwide, effective traffic observation and control using modern innovations is now a clear necessity. The primary goal in this field is vehicle detection, and counting vehicles also plays a significant part. These two applications are crucial. The goal is to create a classifier system that can count, identify, and categorize diverse cars on varied surfaces, such as streets, roads, highways, and other transportation hubs.

III. Introduction:

Traffic issues are a significant issue occurring in many urban areas in the world. There are many significant reasons for the traffic issue. The quantity of individuals moving into a metropolitan region has developed generously, prompting an emotional expansion in the number of vehicles. However, the street limit has become generally lethargic and gets lacking. This causes an irregularity between the quantities of vehicles and streets, bringing about street gridlock, particularly in enormous urban areas. An insufficiency of public transportation frameworks likewise causes a similar issue. The process of tracking a moving object with a camera is known as vehicle tracking. It's difficult to enhance tracking performance without capturing vehicles in surveillance camera video sequences. The number of applications for this technology is growing, including those for traffic control, monitoring, flow, security, etc. Utilizing this technology is anticipated to be relatively affordable. In many cities and metropolitan areas, video and image processing has been employed for traffic surveillance, analysis, and monitoring of traffic conditions.

Vehicle detection seeks to deliver data assisting with vehicle counts, vehicle speed measurement, identifying traffic accidents, forecasting traffic flow, etc. Vehicle detecting and counting have a significant influence on numerous systems that help to regulate and control traffic in urban areas.

The fundamental goal is to detect and count moving vehicles with clear accuracy and to have the option to do such on streets, highways, in little paths, etc. OpenCV using YOLOv3 model-analysis and understanding of images and videos taken by an advanced camera has acquired more approval and been utilized in numerous fields including industry, medication, robotics, and so on. Computer vision has likewise been applied for addressing traffic and transportation problems. In the area of automotive object detection, deep convolutional networks (CNNs) have had incredible success. CNNs are adept at learning image characteristics and are capable of a variety of related tasks, including classification and bounding box regression. For instance, a video sequence of streets can be handled and analyzed to identify and count vehicles. Additional data, such as vehicle speed or traffic density, can likewise be determined with the help of computer vision. This may directly help two kinds of people. Street users and traffic organizations. If street users know the constant traffic data, they can utilize the data to pick the most ideal path for traveling and can keep away from congestion. Then again, traffic organizations can use the traffic data in their traffic control systems, bringing about better traffic to the board.

IV. Dataset Description:

Datasets are a crucial component of deep learning and machine learning in general. A model with a high level of precision and recall will benefit from a solid dataset. There are some well-known names that researchers and practitioners frequently use and refer to when discussing object recognition in photographs or movies. The list contains the names Pascal, ImageNet, SUN, and COCO.

Common Objects in Context, or COCO. The photos in the COCO dataset are drawn from real-world scenarios, as suggested by the name, giving the things photographed in the scenes "context." To further clarify this, we can use an analogy. Imagine that we wish to find a person or object in an image. A close-up picture of a person taken out of context will be considered an isolated image. We can only conclude from the image that it is a portrait of a person. However, it will be difficult to describe the scene where the shot was taken without including other images that show both the subject and the studio or surrounding area.

Dataset Statistics:

By putting the issue of object recognition in the perspective of the more general issue of scene understanding, we want to advance the state-of-the-art in object recognition. This is accomplished by compiling photographs of intricate everyday scenarios with typical objects in their natural settings. Per-instance segmentations are used to label objects, which helps with accurate object localization. Our dataset includes images of **91 different object kinds** that a 4-year-old might easily recognize. Our dataset was produced using unique user interfaces for category recognition, instance spotting, and instance segmentation, with a total of **2.5 million** tagged instances in **328k photos**.

Data Implementation:

In this module, the 'coco.names' is the dataset file name which contains different object categories of 80 instances which are used to train the YOLOv3 model. Since this is the vehicle detection and classification approach where we will only detect bikes, cars, and trucks, we created an index led 'required_class_index' which contains those specific classes. Each class represents each object from the dataset to construct a vehicle detection algorithm.

```
# Store Coco Names in a list
classesFile = "coco.names"
classNames = open(classesFile).read().strip().split('\n')
print(classNames)
print(len(classNames))

# class index for our required detection classes
required_class_index = [2, 3, 5, 7]

detected_classNames = []
```

Fig1. Dataset Classes

V. Methodology:

A. Pre-requisites:

Let's look at the YOLO (You Only Look Once) real-time object detection algorithm, which is one of the most effective object detection algorithms and incorporates many of the most innovative ideas from the computer vision research community. Object detection is an important feature of autonomous vehicle technology. It's a branch of computer vision that's exploding and performing far better than it did just a few years ago. Below are the technologies which have been used to build this project.

1. **Python** – 3.x (We used python 3.8.8 in this project)
2. **OpenCV** – 4.4.0
 - a. It is strongly recommended to run DNN models on GPU. We can install OpenCV via “pip install OpenCV-python opencv_contrib-python”.
3. **NumPy** – 1.20.3
4. **YOLOv3** - Pre-trained model weights and Config Files.

B. Model Architecture and Deployment:

YOLO: You Only Look Once

Object detection is a classic computer vision problem in which you work to recognize what and where — specifically, what objects are inside a given image and where they are in the image. Object detection is a more complex problem than classification, which can recognize objects but does not indicate where they are in the image. Furthermore, classification does not work on images that contain more than one object.

YOLO takes a completely different approach. YOLO is a clever convolutional neural network (CNN) for real-time object detection. The algorithm uses a single neural network to process the entire image, then divides it into regions and predicts bounding boxes and probabilities for each. The predicted probabilities are used to weigh these bounding boxes.

YOLO is popular because it achieves high accuracy while running in real-time. The algorithm "only looks once" at the image in the sense that making predictions requires only one forward propagation pass through the neural network.

After non-max suppression (which ensures that the object detection algorithm detects each object only once), it outputs recognized objects along with bounding boxes.

A single CNN predicts multiple bounding boxes and class probabilities for those boxes using YOLO. YOLO trains full images and optimizes detection performance directly.

This model has several advantages over other object detection methods:

- YOLO moves extremely quickly.
- During training and testing, YOLO sees the entire image, so it implicitly encodes contextual information about classes as well as their appearance.
- YOLO learns generalizable representations of objects, allowing it to outperform other top detection methods when trained on natural images and tested on artwork.

YOLO's framework functionality:

YOLO follows the following techniques simultaneously:

1. Residual Blocks - This function basically divides an image into $N \times N$ grids.
2. Bounding Box Regression - The model receives each grid cell. Then YOLO calculates the likelihood that the cell contains a specific class, and the class with the highest probability is chosen.
3. Intersection Over Union (IOU) - IOU is a metric that evaluates the intersection of the predicted and ground truth bounding boxes. A Non-max suppression technique is used to eliminate the close bounding boxes by performing the IoU with the one with the highest-class probability among them.

Here's the combination of all the 3 steps mentioned above:

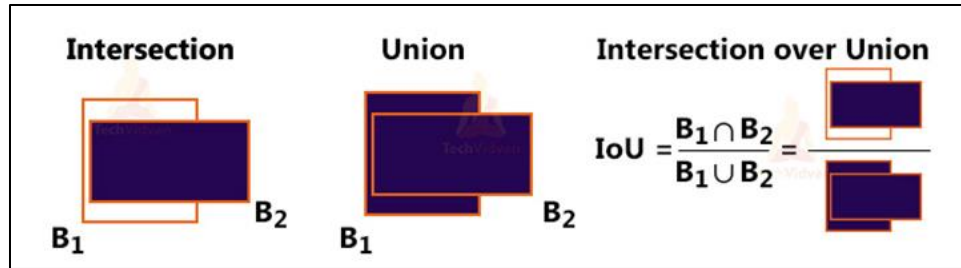


Fig2. Combination of these 3 techniques

YOLO's Architecture:

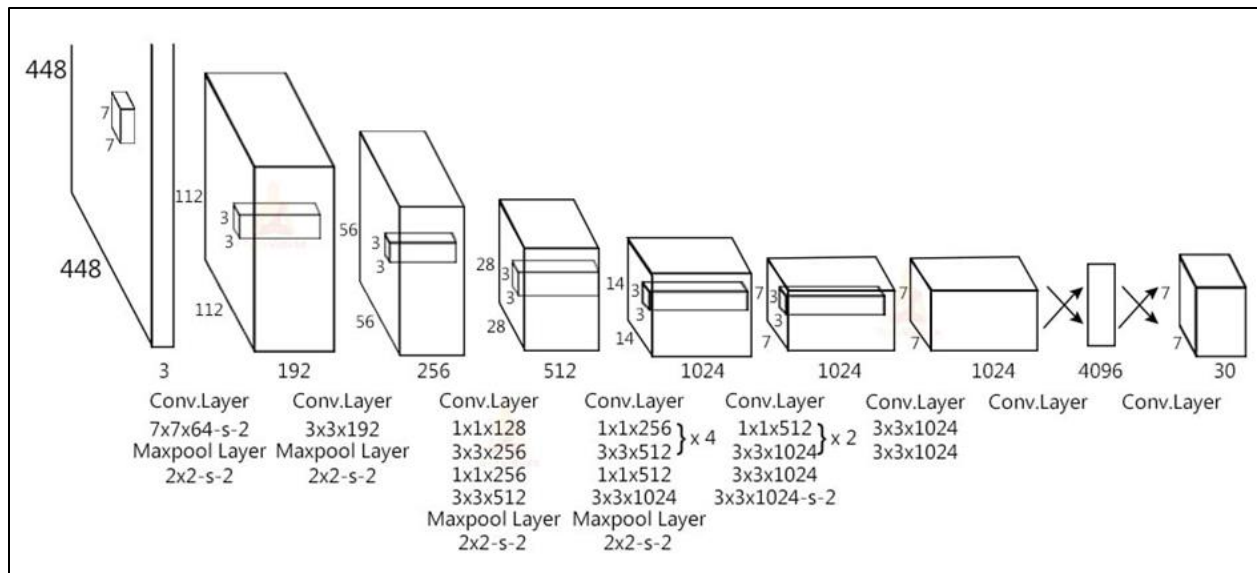


Fig3. Our detection network is made up of 24 convolutional layers, which are followed by two fully connected layers. The features space from preceding layers is reduced by alternating 1x1 convolutional layers. On the ImageNet classification task, we pretrain the convolutional layers at half the resolution (224x224 input image) and then double the resolution for detection.

- The YOLO network is made up of 24 convolutional layers that are followed by two fully connected layers. The convolutional layers are trained on the ImageNet classification task at half the resolution (224x224 input image) before being double trained for detection.

- The various layers to reduce the feature space from previous layers, alternate 1x1 reduction layer and 3x3 convolutional layers.
- The final four layers are added to train the network to detect objects.
- The final layer forecasts the object class and bounding box probabilities.

To interact with YOLO directly, we have used OpenCV's DNN module. DNN is an abbreviation for Deep Neural Network. OpenCV includes a function for running DNN algorithms.

C. Design and Implementation:

A free framework called OpenCV combines computer vision and machine learning. This platform makes it possible for machine perception to be incorporated into commercially available works more quickly by offering a universal architecture for computer vision applications.

OpenCV provides access to more than 2,500 new and old algorithms. With the aid of this library, users may do a wide range of tasks, like removing red eyes, extracting 3D models of objects, tracking eye movements, etc. We begin by configuring our Tracker for object detection with the aforementioned information in mind before developing the code to identify and classify the vehicles. The implementation process is explained in depth below.

Tracker System:

Filename: Tracker.py

To track an item, the tracker essentially employs the Euclidean distance notion. The distance between two points is known as the Euclidean distance in coordinate geometry.

The length of the line segment separating the two places should be measured in order to determine the distance between them. Euclidean distance is frequently employed in machine learning algorithms as a default distance metric to determine how similar two recorded observations are to one another.

If there is a discrepancy between two center points of an object in the current frame and the previous frame, it checks to see if the difference is less than a threshold distance and, if so, certifies that the object is the same object from the preceding frame. We can quickly see how it works by looking at the figure below.

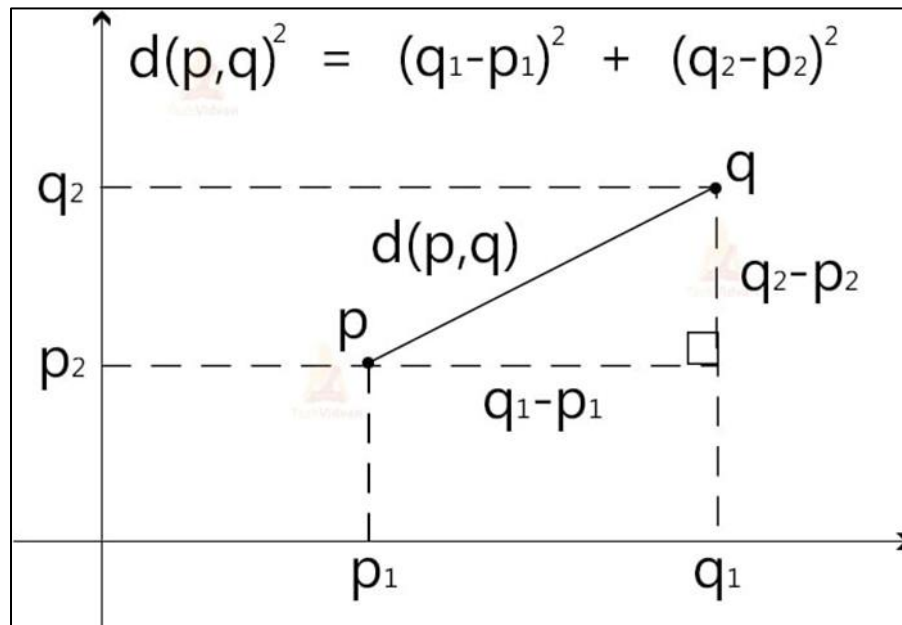


Fig4. Euclidean distance

To build the tracker, we utilized the Python math library. A built-in Python module that is always available is the math module. We used `import math` to import this module's mathematical functions for use.

Once that has been completed, we will build the class "EuclideanDistTracker" and a function called "__init__" to send arguments to the class instance once it has been initiated. Then, we define the variables "self.center points" and "self.id count" to keep track of the objects that have been identified and to increase the count by one whenever a new object is found. It is illustrated below.

```
import math

class EuclideanDistTracker:
    def __init__(self):
        self.center_points = {}
        self.id_count = 0
```

Fig5. Euclidean distance code

Additionally, we must change the object IDs to stop previously discovered items from being counted more than once. And in order to fulfill the latter, we assign an ID to an object that has already been discovered and clear the dictionary by centering on IDs that are no longer in use. Here, we also make sure that the item is the same as it was in the previous frame if the distance is less than 25. Everything is explained in the following code sippet.

```

def update(self, objects_rect):
    objects_bbs_ids = []
    for rect in objects_rect:
        x, y, w, h, index = rect
        cx = (x + x + w) // 2
        cy = (y + y + h) // 2

        # Find out if that object was detected already
        same_object_detected = False
        for id, pt in self.center_points.items():
            dist = math.hypot(cx - pt[0], cy - pt[1])
            if dist < 25:
                self.center_points[id] = (cx, cy)
                objects_bbs_ids.append([x, y, w, h, id, index])
                same_object_detected = True
                break

        # New object is detected we assign the ID to that object
        if same_object_detected is False:
            self.center_points[self.id_count] = (cx, cy)
            objects_bbs_ids.append([x, y, w, h, self.id_count, index])
            self.id_count += 1

        # Clean the dictionary by center points to remove IDS not used anymore
        new_center_points = {}
        for obj_bb_id in objects_bbs_ids:
            _, _, _, _, object_id, index = obj_bb_id
            center = self.center_points[object_id]
            new_center_points[object_id] = center

        # Update dictionary with IDs not used removed
        self.center_points = new_center_points.copy()
        return objects_bbs_ids

```

Fig6. Calculating points

Vehicle detection and Classification:

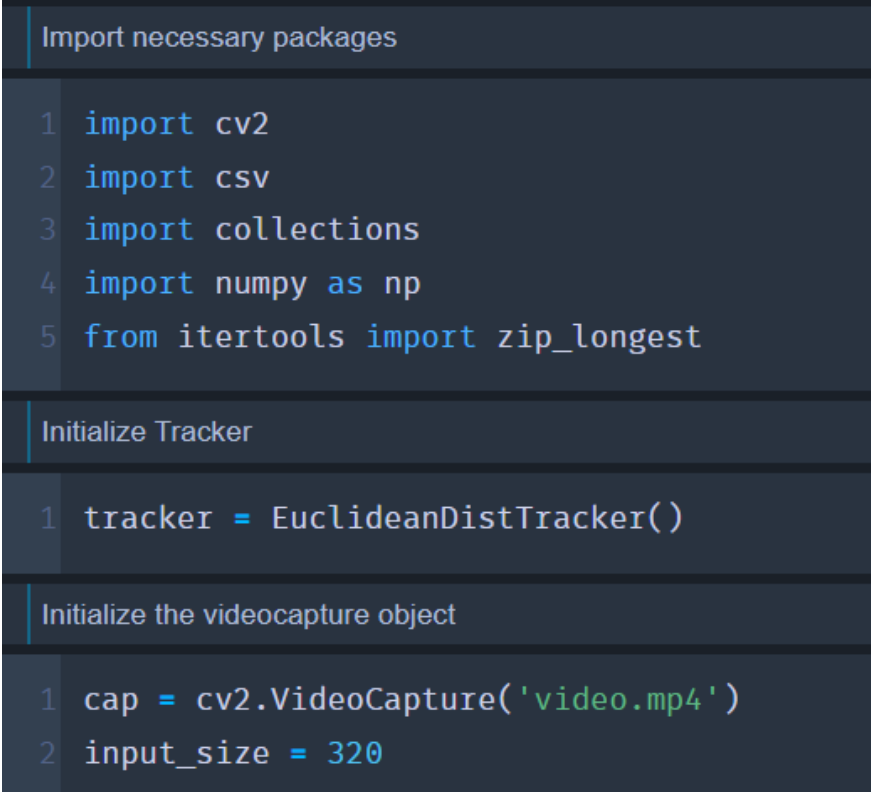
Filename: vehicle_counter.py

In this project, we will detect and classify cars, HMV (Heavy Motor Vehicle), and LMV (Light Motor Vehicle) on the road, as well as count the number of vehicles on the road. And the data will be saved to analyse various vehicles on the road.

To complete this project, we have developed two programs. The first is a vehicle detection tracker that uses OpenCV to keep track of every detected vehicle on the road, and the second is the main detection program.

STEPS:

1. Import the necessary packages and start the network.
2. Retrieve frames from a video file.
3. Run the detection after pre-processing the frame.
4. Perform post-processing on the output data.
5. Count and track all vehicles on the road.
6. Save the completed data as a CSV file.

Elaborating the steps:**Step 1. Import the necessary packages and start the network:**

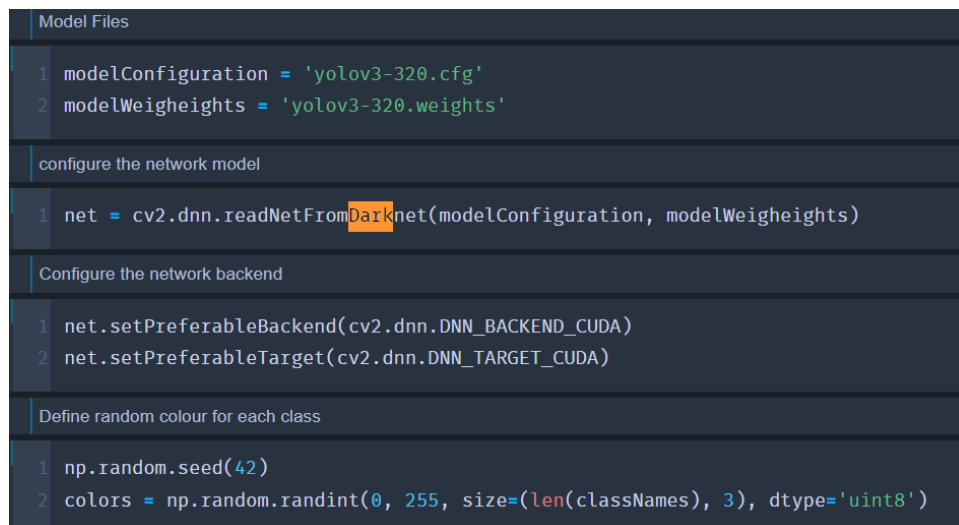
```
Import necessary packages
1 import cv2
2 import csv
3 import collections
4 import numpy as np
5 from itertools import zip_longest

Initialize Tracker
1 tracker = EuclideanDistTracker()

Initialize the videocapture object
1 cap = cv2.VideoCapture('video.mp4')
2 input_size = 320
```

Fig7. Import the necessary packages

- First, we import all the project's required packages. Then, from the tracker program, we initialize the `EuclideanDistTracker()` object and set the object to "tracker." `confThreshold` and `nmsThreshold` are the detection and suppression minimum confidence score thresholds, respectively. These are the crossing line positions that will be used to count the vehicles.
- Because YOLOv3 is trained on the COCO dataset, we read the file containing all the class names and store them in a list. The COCO dataset includes 80 distinct classes.



```
Model Files
1 modelConfiguration = 'yolov3-320.cfg'
2 modelWeights = 'yolov3-320.weights'

configure the network model
1 net = cv2.dnn.readNetFromDarknet(modelConfiguration, modelWeights)

Configure the network backend
1 net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
2 net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)

Define random colour for each class
1 np.random.seed(42)
2 colors = np.random.randint(0, 255, size=(len(classNames), 3), dtype='uint8')
```

Fig8. Training YOLO model

- For this project, we only need to detect cars, motorcycles, buses, and trucks, so the required class index contains the index of those classes from the COCO dataset.
- We use the `cv2.dnn.readNetFromDarknet()` function to configure the network.
- Set the DNN backend to CUDA because we're using GPU and comment out those lines if you're using CPU. In this case, we are using CPU.

- We generate a random colour for each class in our dataset using the `np.random.randint()` function. These colours will be used to draw the rectangles around the objects.
- The `random.seed()` function saves the state of a random function so that it can generate some random number on each execution, even if it generates the same random numbers on other machines as well.

Step 2. Retrieve frames from a video file.

```
Initialize the videocapture object  
1 cap = cv2.VideoCapture('video.mp4')  
2 input_size = 320  
  
Detection confidence threshold  
1 confThreshold = 0.2  
2 nmsThreshold = 0.2  
  
1 font_color = (0, 0, 255)  
2 font_size = 0.5  
3 font_thickness = 2
```

```
def realTime():  
    while True:  
        success, img = cap.read()  
        img = cv2.resize(img, (0, 0), None, 0.5, 0.5)  
        ih, iw, channels = img.shape  
        blob = cv2.dnn.blobFromImage(img, 1 / 255, (input_size, input_size), [0, 0, 0], 1, crop=False)  
  
        # Set the input of the network  
        net.setInput(blob)  
        layersNames = net.getLayerNames()  
        outputNames = [(layersNames[i[0] - 1]) for i in net.getUnconnectedOutLayers()]  
        # Feed data to the network  
        outputs = net.forward(outputNames)  
  
        # Find the objects from the network output  
        postProcess(outputs, img)  
  
        # Draw the crossing lines  
        cv2.line(img, (0, middle_line_position), (iw, middle_line_position), (255, 0, 255), 2)  
        cv2.line(img, (0, up_line_position), (iw, up_line_position), (0, 0, 255), 2)  
        cv2.line(img, (0, down_line_position), (iw, down_line_position), (0, 0, 255), 2)
```

Fig9. Retrieve frames from a video file

- Cap.read() reads each frame from the capture object by reading the video file through the videoCapture object.
- We reduced our frame by half by using cv2.reshape().
- Then, using the cv2.line() function, we draw the intersecting lines in the frame.
- Finally, we used the cv2.imshow() function to display the output image.

```
# Draw counting texts in the frame
cv2.putText(img, "Up", (110, 20), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
cv2.putText(img, "Down", (160, 20), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
cv2.putText(img, "Car: " + str(up_list[0]) + " " + str(down_list[0]), (20, 40), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
cv2.putText(img, "Motorbike: " + str(up_list[1]) + " " + str(down_list[1]), (20, 60), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
cv2.putText(img, "Bus: " + str(up_list[2]) + " " + str(down_list[2]), (20, 80), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
cv2.putText(img, "Truck: " + str(up_list[3]) + " " + str(down_list[3]), (20, 100), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)

# Show the frames
cv2.imshow('Output', img)
if cv2.waitKey(1) == ord('q'):
    break
```

Fig10. Draw Counting texts in the frame

Step 3. Run the detection after pre-processing the frame.

```
# Set the input of the network
net.setInput(blob)
layerNames = net.getLayerNames()
outputNames = [(layerNames[i[0] - 1]) for i in net.getUnconnectedOutLayers()]
# Feed data to the network
outputs = net.forward(outputNames)

# Find the objects from the network output
postProcess(outputs, img)
```

Fig11. Setting up the input of the network

- Our YOLO version accepts 320320 image objects as input. The network's input is a blob object. The function dnn.blobFromImage() accepts an image as input and returns a blob object that has been resized and normalized.
- The image is fed into the network using net.forward(). And it produces a result.
- Finally, to post-process the output, we invoke our custom postProcess() function.

Step 4. Perform post-processing on the output data.

```
def postProcess(outputs,img):
    global detected_classNames
    height, width = img.shape[:2]
    boxes = []
    classIds = []
    confidence_scores = []
    detection = []
    for output in outputs:
        for det in output:
            scores = det[5:]
            classId = np.argmax(scores)
            confidence = scores[classId]
            if classId in required_class_index:
                if confidence > confThreshold:
                    # print(classId)
                    w,h = int(det[2]*width) , int(det[3]*height)
                    x,y = int((det[0]*width)-w/2) , int((det[1]*height)-h/2)
                    boxes.append([x,y,w,h])
                    classIds.append(classId)
                    confidence_scores.append(float(confidence))
```

Fig12. Post-processing on output data

- The network forward output has three outputs. Each output object is an 85-length vector.
- 4 times the bounding box (centerX, centerY, width, height)
- 1 confidence box
- 80x class assurance
- First, we created an empty list called 'detected classNames,' in which we will store all the detected classes in a frame.
- We iterate through each vector of each output using two for loops to collect the confidence score and classId index.
- Then we check to see if the class confidence score is higher than our predefined confThreshold. The information about the class is then collected and stored in three separate lists: box coordinate points, class-Id, and confidence score.
- We reduce the number of boxes and take only the best detection box for the class using the NMSBoxes() method.

- Text is drawn in the frame by cv2.putText.
- We draw a bounding box around the detected object using cv2.rectangle().

Step 5. Count and track all vehicles on the road.

```
# Update the tracker for each object
boxes_ids = tracker.update(detection)
for box_id in boxes_ids:
    count_vehicle(box_id, img)
```

Fig13. Count and track

- After receiving all the detections, we use the tracker object to keep track of those objects. The tracker.update() function keeps track of all detected objects and updates their positions.
- The custom function Count vehicle counts the number of vehicles that have passed through the road.
- Then we created a count_vehicle function and followed by that created two temporary empty lists to store the vehicles id's that enter the entry crossing line. The center point of a rectangle box is returned by the find center function.
- Furthermore, we keep track of each vehicle's position and Id.

```
# List for store vehicle count information
temp_up_list = []
temp_down_list = []
up_list = [0, 0, 0, 0]
down_list = [0, 0, 0, 0]
```

Fig14. Storing vehicle count information

- First, we check to see if the object is between the up-crossing line and the middle crossing line, and then we store the object's id in up list for up route vehicle counting. We also do the opposite for down-route vehicles.
- Then we check to see if the object has crossed the down line. If the object crossed the down line, its id is counted as an up-route vehicle, and we add 1 with the specific type of class counter.
- Because we're counting vehicles on the Y-axis, we only need y coordinate points.

Step 6. Save the completed data as a CSV file.

```
6  # Write the vehicle counting information in a file and save it
7  with open("data.csv", 'w') as f1:
8      cwriter = csv.writer(f1)
9      cwriter.writerow(['Direction', 'car', 'motorbike', 'bus', 'truck'])
10     up_list.insert(0, "Up")
11     down_list.insert(0, "Down")
12     cwriter.writerow(up_list)
13     cwriter.writerow(down_list)
14     f1.close()
15     # print("Data saved at 'data.csv'")
16     # Finally release the capture object and destroy all active windows
17     cap.release()
18     cv2.destroyAllWindows()
```

Fig15. Saving the completed data

- We opened a new file data.csv with write permission only using the open function.
- Then we write three rows: the first with class names and directions, the second with up and down route counts, and the third with both.
- The writerow() function saves a row of data to a file.

Step 7. Image Implementation:

Since we have implemented this model using video input for counting, detecting, and classifying the vehicles from the input. It is not possible to count vehicles that travel through a road in a particular direction on a static image. Because it is a continuous process. But can classify and count the number of vehicles that are present on the road or an image. And later we can analyze the data.

```
image_file = 'vehicle_classification-image02.png'
def from_static_image(image):
    img = cv2.imread(image)

    blob = cv2.dnn.blobFromImage(img, 1 / 255, (input_size, input_size), [0, 0, 0], 1, crop=False)

    # Set the input of the network
    net.setInput(blob)
    layersNames = net.getLayerNames()
    outputNames = [(layersNames[i] - 1) for i in net.getUnconnectedOutLayers()]
    # Feed data to the network
    outputs = net.forward(outputNames)

    # Find the objects from the network output
    postProcess(outputs, img)

    # count the frequency of detected classes
    frequency = collections.Counter(detected_classNames)
    print(frequency)

    # Draw counting texts in the frame
    cv2.putText(img, "Car:      "+str(frequency['car']), (20, 40), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Motorbike:  "+str(frequency['motorbike']), (20, 60), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Bus:      "+str(frequency['bus']), (20, 80), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Truck:    "+str(frequency['truck']), (20, 100), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)

    cv2.imshow("image", img)

    cv2.waitKey(0)
```

Fig16. Image Implementation

- First, we create a function that takes an image file as input.
- Using the cv2.imread() function we read the image.
- After that we repeat the exact same process as the previous step for detecting objects.
- Previously we stored all the detected objects in the 'detected_classNames' list. So, using collections. Counter(detected_classNames) we calculate the frequency of the elements in the list. It returns a dictionary containing the element as the key of the dictionary and the frequency of the element as the value of that particular key.

```

# Draw the crossing lines
cv2.line(img, (0, middle_line_position), (iw, middle_line_position),
(255, 0, 255), 1)
cv2.line(img, (0, up_line_position), (iw, up_line_position), (0, 0, 255), 1)
cv2.line(img, (0, down_line_position), (iw, down_line_position), (0, 0, 255), 1)

# Show the frames
cv2.imshow('Output', img)

# save the data to a csv file
with open("static-data.csv", 'a') as f1:
    cwriter = csv.writer(f1)
    cwriter.writerow([image, frequency['car'], frequency['motorbike'], frequency['bus'], frequency['truck']])
f1.close()

```

Fig17. Drawing cross lines and saving the data

- After that we draw the counting texts on the frame.
- `cv2.imshow()` shows the output image in a new OpenCV window.
- `cv2.waitKey(0)` keeps the window open until any key is pressed.
- And finally, we save the data to a csv file.

D. Results and Demonstration:

Upon successful completion of building and implementing the YOLOv3 model for the vehicle counting, detection and classifications algorithm we got the results for each phase with better accuracy. Here are the results of each phase.

Dataset Implementation:

Input:

```

# Store Coco Names in a list
classesFile = "coco.names"
classNames = open(classesFile).read().strip().split('\n')
print(classNames)
print(len(classNames))

```

Fig18. Input for Dataset Implementation

Output:

```
[ 'person', 'bicycle', 'car', 'motorbike', 'aeroplane', 'bus', 'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'stop sign', 'parking meter',
, 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie', 'suitcase',
, 'frisbee', 'skis', 'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket', 'bottle', 'wi
ne glass', 'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
, 'chair', 'sofa', 'pottedplant', 'bed', 'diningtable', 'toilet', 'tvmonitor', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone', 'microwave', '
oven', 'toaster', 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
80
```

Fig19. Output for Dataset Implementation**Image Implementation:****Input:**

```
image_file = 'vehicle_classification-image02.png'
def from_static_image(image):
    img = cv2.imread(image)

    blob = cv2.dnn.blobFromImage(img, 1 / 255, (input_size, input_size), [0, 0, 0], 1, crop=False)

    # Set the input of the network
    net.setInput(blob)
    layersNames = net.getLayerNames()
    outputNames = [(layersNames[i - 1]) for i in net.getUnconnectedOutLayers()]
    # Feed data to the network
    outputs = net.forward(outputNames)

    # Find the objects from the network output
    postProcess(outputs, img)

    # count the frequency of detected classes
    frequency = collections.Counter(detected_classNames)
    print(frequency)

    # Draw counting texts in the frame
    cv2.putText(img, "Car:      "+str(frequency['car']), (20, 40), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Motorbike: "+str(frequency['motorbike']), (20, 60), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Bus:      "+str(frequency['bus']), (20, 80), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
    cv2.putText(img, "Truck:    "+str(frequency['truck']), (20, 100), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)

    cv2.imshow("image", img)

    cv2.waitKey(0)
```

Fig20. Input for Image Implementation

image

Car: 9
Motorbike: 0
Bus: 0
Truck: 10

TRUCK 94%
TRUCK 85%
TRUCK 94%
TRUCK 91%
TRUCK 94%
TRUCK 79%
CAR 98%
CAR 97%
TRUCK 96%
TRUCK 96%
CAR 98%
CAR 90%
CAR 94%
CAR 77%
TRUCK 94%
CAR 99%
CAR 99%

Video Implementation:

```
def realtime():
    while True:
        success, img = cap.read()
        img = cv2.resize(img, (0,0), None, 0.5, 0.5)
        ih, iw, channels = img.shape
        blob = cv2.dnn.blobFromImage(img, 1 / 255, (input_size, input_size), [0, 0, 0], 1, crop=False)

        # Set the input of the network
        net.setInput(blob)
        layerNames = net.getLayerNames()
        outputNames = [[layerNames[i - 1]] for i in net.getUnconnectedOutLayers()]
        # Feed data to the network
        outputs = net.forward(outputNames)

        # Find the objects from the network output
        postProcess(outputs, img)

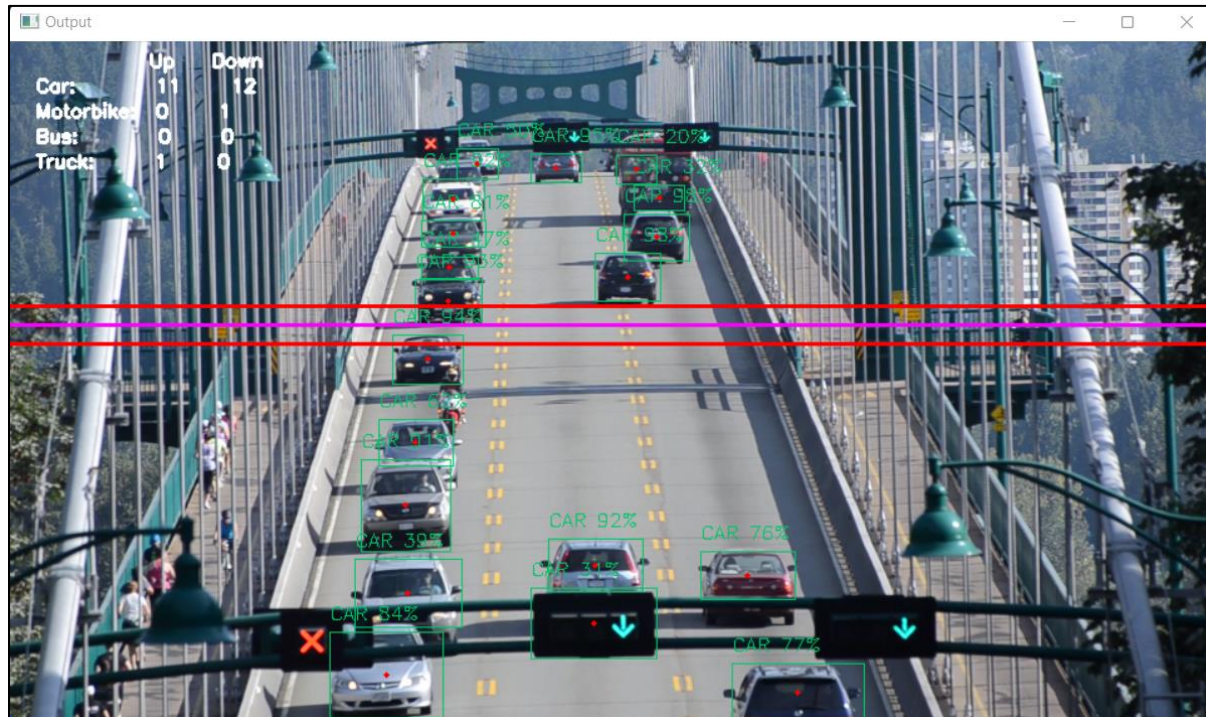
        # Draw the crossing lines

        cv2.line(img, (0, middle_line_position), (iw, middle_line_position), (255, 0, 255), 2)
        cv2.line(img, (0, up_line_position), (iw, up_line_position), (0, 0, 255), 2)
        cv2.line(img, (0, down_line_position), (iw, down_line_position), (0, 0, 255), 2)

        # Draw counting texts in the frame
        cv2.putText(img, "Up", (110, 20), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
        cv2.putText(img, "Down", (160, 20), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
        cv2.putText(img, "Car: " + str(up_list[0]) + " " + str(down_list[0]), (20, 40), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
        cv2.putText(img, "Motorbike: " + str(up_list[1]) + " " + str(down_list[1]), (20, 60), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
        cv2.putText(img, "Bus: " + str(up_list[2]) + " " + str(down_list[2]), (20, 80), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)
        cv2.putText(img, "Truck: " + str(up_list[3]) + " " + str(down_list[3]), (20, 100), cv2.FONT_HERSHEY_SIMPLEX, font_size, font_color, font_thickness)

        # Show the frames
        cv2.imshow('output', img)
```

Fig23. Input for Video Implementation

Output:**GIF1. Output for Video Implementation****VI. Challenges, Recommendations and Solutions:****Challenges: Limitation on YOLO model:**

- YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict.
- Our model struggles with small objects that appear in groups, such as flocks of birds. Since our model learns to predict bounding boxes from data, it struggles to generalize objects in new or unusual aspect ratios or configurations. Our model also uses relatively coarse features for predicting bounding boxes since our architecture has multiple down-sampling layers from the input image.

- Finally, while we train on a loss function that approximates detection performance, our loss function treats errors the same in small bounding boxes versus large bounding boxes. A small error in a large box is generally benign but a small error in a small box has a much greater effect on IOU. Our main source of error is incorrect localizations.

VII. Recommendations and Solutions:

There are several suggested strategies to effectively improve this model for real-time use in order to address the issues with this vehicle counting, detection, and classifier system.

Using different CNN approaches:

In the area of automotive object detection, deep convolutional networks (CNNs) have had incredible success. CNNs are adept at learning image characteristics and are capable of a variety of related tasks, including classification and bounding box regression. Two broad categories can be made for the detecting techniques. The two-stage method creates a candidate box of the object using a variety of algorithms, and then uses a convolutional neural network to classify the object. The one-stage method directly transforms the positioning problem of the object bounding box into a regression problem for processing rather than creating a candidate box.

- **Two Stage Methods of CNN are, R-CNN, R-FCN, FPN, and Mask RCNN.**
 - Region-CNN (R-CNN) employs selective region search in the image during the two-stage process. The convolutional network requires a fixed-size image input, and because of its deeper structure, training takes a long time and uses up a lot of memory. SPP NET, which is based on the concept of spatial pyramid matching, enables the network to accept images of different sizes and produce fixed results.

- Convolutional networks' feature extraction processes, feature selection processes, and classification abilities have all been enhanced in various ways by R-FCN, FPN, and Mask RCNN.
- **The One Stage Methods of CNN are, Single Shot Multi-Box Detector (SSD) and You Only Look Once (YOLO).**
 - The Single Shot Multibox Detector (SSD) [12] and You Only Look Once (YOLO) [13] frameworks are the most significant one-stage techniques. A default collection of anchor boxes with various aspect ratios is utilized in SSD, which employs the Muti Box [14], Region Proposal Network (RPN), and multi-scale representation methods to position the item more precisely. The YOLO [13] network divides the image into a predetermined number of grids, unlike SSD. Each grid oversees forecasting items whose centers are on its grid. The BN (Batch Normalization) layer, which was added by YOLOv2 [15], forces the network to normalize the input of each layer and speeds up network convergence.
 - Every ten batches, YOLOv2 randomly chooses a new image size using a multi-scale training technique. The YOLOv3 [16] network is used for our vehicle object detection. YOLOv3 bases its logistic regression on YOLOv2 for the object category. Two-class cross-entropy loss is the category loss approach, and it may deal with various label issues for the same object. Additionally, the box confidence is regressed using logistic regression to see if the IOU between the a priori box and the actual box is greater than 0.5. Only the largest prior box of the IOU is taken if more than one priority box meets the criterion. YOLOv3 predicts the object in the image using three alternative scales for the final object prediction.

Detection-Based Tracking (DBT) and Detection-Free Tracking (DFT) (Europe):

- The DBT approach first detects moving objects in video frames before tracking them. The tracking object must be initialized by the DFT method, which is unable to accommodate the arrival of fresh objects or the departure of old ones. The Multiple Object Tracking algorithm must consider both the issue of inter-frame objects and the similarity of intra-frame objects. You can use normalized cross-correlation to determine how similar intra-frame objects are (NCC). The distance of the color histogram between the items is calculated using the Bhattacharyya distance, as seen in.
- The Multiple Object Tracking algorithm must consider both the issue of inter-frame objects and the similarity of intra-frame objects. You can use normalized cross-correlation to determine how similar intra-frame objects are (NCC). The distance of the color histogram between the items is calculated using the Bhattacharyya distance, as seen in.
- In order to associate inter-frame objects, it is important to establish that an object can only exist on a single track and that a single object can only relate to a single track. The current solutions to this issue are detection-level exclusion or trajectory-level exclusion. [28] employed SIFT feature points for object tracking, but this is slow, to address the issues brought on the scale changes and illumination changes of moving objects.

ORB Algorithm with YOLO:

- FAST KeyPoint detector and BRIEF descriptor are combined to create ORB, which has some extra characteristics to boost performance. To find features in the given image, FAST, or Features from Accelerated Segment Test, is employed.

- Additionally, a pyramid is used to create multiscale features. Since it no longer computes the orientation and descriptions for the features, BRIEF fills this gap.
- Because the BRIEF performs poorly with rotation, ORB uses BRIEF descriptors. Therefore, ORB rotates the BRIEF in accordance with the positioning of keypoints. The rotation matrix of the patch is found using the patch's orientation, and it rotates the BRIEF to produce the rotated version. Better extraction feature points can be obtained by ORB at a large speed advantage over SIFT.
- Then, for multi-object tracking, the ORB algorithm is applied. To establish correlation between the same object and several video frames, the ORB algorithm extracts the detected box's features and compares them. Finally, traffic statistics are computed. The trajectory produced by object tracking, the direction in which the vehicle is traveling, and traffic data, such as the number of vehicles in each category, are all recorded.
- This system creates a detection tracking and traffic information collecting plan throughout the whole field of the camera view, improving the object detection accuracy from the highway surveillance video perspective.

However, with the advancement of deep learning technology, CNN-based vehicle detection has found success in Europe. In the German city of Karlsruhe, Fast Region-CNN is being employed for vehicle detection in traffic scenes. Fast R-CNN uses a selective search strategy to find all candidate frames, which is notably time-consuming, and the vehicle detection speed is slow.

Conclusion

In addition to proposing an object detection and tracking approach for highway surveillance video sequences, this study created a high-definition vehicle object dataset from the perspective of surveillance cameras. The highway's road surface area was extracted to produce a ROI area that was more productive. The end-to-end highway vehicle recognition model was obtained by the YOLO object detection method using the annotated highway vehicle object dataset. The road surface area was divided into a remote area and a proximal area in order to handle the issue of the identification of small objects and the multi-scale variation of the object. In order to achieve successful vehicle identification outcomes in the monitoring field, the two road sections of each frame were sequentially detected. In order to gather information about the present state of the highway traffic, including driving direction, vehicle type, and vehicle number, the vehicle trajectories were last examined. The experimental findings demonstrated the effectiveness and applicability of the suggested vehicle detection and tracking method for highway surveillance video situations. The method described in this study is low-cost, highly stable, and doesn't call for extensive building work or installation work on existing monitoring equipment, in contrast to the conventional method of monitoring vehicle traffic via hardware. The surveillance camera can be further calibrated to acquire the internal and external parameters of the camera, following the study described in this paper.

Thus, the image coordinate system of the vehicle trajectory's location information is changed to the global coordinate system. Based on the camera's calibration results, the speed of the vehicle can be determined. Abnormal parking and traffic jam occurrences can be detected in conjunction with the vehicle detection and tracking techniques that have been presented in order to get more comprehensive traffic data.

References

- *Feature matching using ORB algorithm in Python-OpenCV*. (2020, April 22). GeeksforGeeks. <https://www.geeksforgeeks.org/feature-matching-using-orb-algorithm-in-python-opencv/>
- Gulati, A. P. (2021, December 31). *Vehicle Detection and Counting System using OpenCV*. Analytics Vidhya. <https://www.analyticsvidhya.com/blog/2021/12/vehicle-detection-and-counting-system-using-opencv/>
- Mathews, H. R. (n.d.). *Closing.Py at main · HimaRaniMathews/Vehicle-Detection-Classification-and-Counting*.
- ODSC-Open Data Science. (2018, September 25). *Overview of the YOLO object detection algorithm*. Medium. <https://odsc.medium.com/overview-of-the-yolo-object-detection-algorithm-7b52a745d3e0>
- Sharma, P. (2018, December 6). *A practical guide to object detection using the popular YOLO framework – part III (with python codes)*. Analytics Vidhya. <https://www.analyticsvidhya.com/blog/2018/12/practical-guide-object-detection-yolo-framework-python/>
- Song, H., Liang, H., Li, H., Dai, Z., & Yun, X. (2019). Vision-based vehicle detection and counting system using deep learning in highway scenes. *European Transport Research Review*, 11(1), 1–16. <https://doi.org/10.1186/s12544-019-0390-4>
- Srishilesh, P. S. (n.d.). *Understanding COCO dataset*. Engineering Education (EngEd) Program | Section. Retrieved October 28, 2022, from <https://www.section.io/engineering-education/understanding-coco-dataset/>

- *What object categories / labels are in COCO dataset?* (2018, April 12). Amikeline | Technology Blog. <https://tech.amikeline.com/node-718/what-object-categories-labels-are-in-coco-dataset/>
- (N.d.). Techvidvan.com. Retrieved October 28, 2022, from <https://techvidvan.com/tutorials/opencv-vehicle-detection-classification-counting/>

Appendix

Group2_EAI6080_Main.py

Group2_EAI6080_FinalPPT.pptx