REAL-TIME VEHICLE TRACKING AND SPEED ESTIMATION USING COMPUTER VISION

MAJOR PROJECT REPORT

Submitted to the Department of Computer Applications, Bharathiar University in partial fulfillment of the requirements for the award of the degree of

MASTER OF SCIENCE IN DATA ANALYTICS

Submitted by

VIJAYALAKSHMI S (22CSEG32)

Under the guidance of
Dr. S. GAVASKAR, MCA., Ph.D.,
Professor,
Department of Computer Applications

DEPARTMENT OF COMPUTER APPLICATIONS

BHARATHIAR UNIVERSITY

COIMBATORE – 641 046

APRIL – 2024

DECLARATION

I hereby declare that this project, titled **"REAL-TIME VEHICLE TRACKING AND SPEED ESTIMATION USING COMPUTER VISION"** submitted to the Department of Computer Applications, Bharathiar University, Coimbatore is a record of original project work done by **VIJAYALAKSMI S (22CSEG32)** under the supervision and guidance of **Dr. S GAVASKAR, MCA., Ph.D.,** Department of Computer Applications, Bharathiar University, Coimbatore and that this project work has not previously formed the basis of the award of the Degree / Diploma / Associate Ship / Fellowship or similar title to any candidate of any university.

Place: Coimbatore

Date:

Signature of Candidate

(VIJAYALAKSHMI S)

Countersigned by

Dr. S. GAVASKAR, MCA., Ph.D.,
Professor,
Department of Computer Applications

CERTIFICATE

This is to certify that the project work title **"REAL-TIME VEHICLE TRACKING AND SPEED ESTIMATION USING COMPUTER VISION"** submitted to the Department of Computer Applications, Bharathiar University, in partial fulfilment of the requirement for the award of a Degree in **Master of Science in Data Analytics,** is a record of the original work done by **VIJAYALAKSHMI S (22CSEG32)** under my supervision and guidance and this project work has not formed the basis of the award of any Degree / Diploma / Associate Ship / Fellowship or similar title to any candidate of any university.

Place: Coimbatore

Date:

Project Guide **Head of the Department**

Submitted for the University Viva-Voce Examination held on ……

Internal Examiner **External Examiner**

ACKNOWLEDGEMENT

I thank the Almighty God who showered His grace to make this project successful. The success and outcome of this study required a lot of guidance and assistance from many people and I am extremely fortunate to have had those people all along the work and completion of the project.

I express my sincere gratitude to our professor and head of the Department **Dr. M PUNITHAVALLI, MCA., Ph.D.,** Department of Computer Applications, Bharathiar University, Coimbatore for allowing me to do this project work.

I deem it a special privilege to **Dr S. GAVASKAR, MCA., Ph.D.,** professor, Department of Computer Application, for valuable guidance and interest in my project work from the beginning to the end. My heartfelt thanks to my dear parents, family, and friends for their support and encouragement toward the successful completion of this project. I am highly obliged to all those who have helped me directly and indirectly in making this project successful.

TABLE OF CONTENTS

ABSTRACT

Efficient traffic monitoring and analysis are essential for improving road safety, optimizing traffic flow, and supporting transportation planning initiatives. This project presents a computer vision-based system for vehicle tracking and speed estimation, leveraging the state-of-the-art YOLO (You Only Look Once) object detection model and a custom vehicle tracking algorithm. The system uses the YOLOv8 object detection model to identify vehicles in video frames, and a custom Tracker class to maintain the identity of each detected vehicle. To estimate vehicle speeds, the system detects the road area, calculates reference lines, and measures the time it takes for vehicles to cross these lines. Experimental results demonstrate the system's ability to accurately detect, track, and estimate the speeds of vehicles in real-time. The project highlights the potential of computer vision techniques for intelligent traffic monitoring and analysis.

CHAPTER – I
INTRODUCTION

Traffic monitoring and management have become increasingly critical in modern cities as vehicular congestion continues to rise. Accurate and real-time information about traffic flow, vehicle speeds, and congestion levels is essential for efficient transportation planning and enforcement of traffic regulations. This project aims to develop a computer vision-based system for vehicle detection, tracking, and speed estimation using video footage captured from road cameras.

The system leverages the power of deep learning object detection models, specifically the YOLOv8 (You Only Look Once) model, to identify and locate vehicles in each video frame. The detected vehicles are then tracked across consecutive frames using a custom tracking algorithm, which assigns unique identifiers to each vehicle. This tracking mechanism enables the system to maintain continuity and follow the movement of individual vehicles throughout the video sequence.

One of the key features of the system is its ability to measure the time taken for a vehicle to travel between two predefined lines on the road. By calculating the elapsed time and knowing the distance between these lines, the system can estimate the speed of the vehicle in kilometres per hour. This speed information is then overlaid on the video feed, providing real-time speed data for vehicles traveling in both directions.

1.1 PROBLEM STATEMENT

Monitoring and enforcing speed limits on roads is crucial for ensuring road safety and preventing accidents. Traditional methods of speed enforcement, such as speed cameras and radar guns, are often costly and labor-intensive. This project aims to develop an automated system that can detect and track vehicles from video footage and accurately estimate their speeds. The system should be able to identify and locate vehicles in each frame, assign unique identifiers to track them across consecutive frames, and measure the time taken for a vehicle to travel between two predefined points on the road. By calculating the elapsed time and distance, the system can determine the vehicle's speed, which can be used for speed limit enforcement and traffic monitoring purposes.

1.2 WORKFLOW:

FLOW CHART

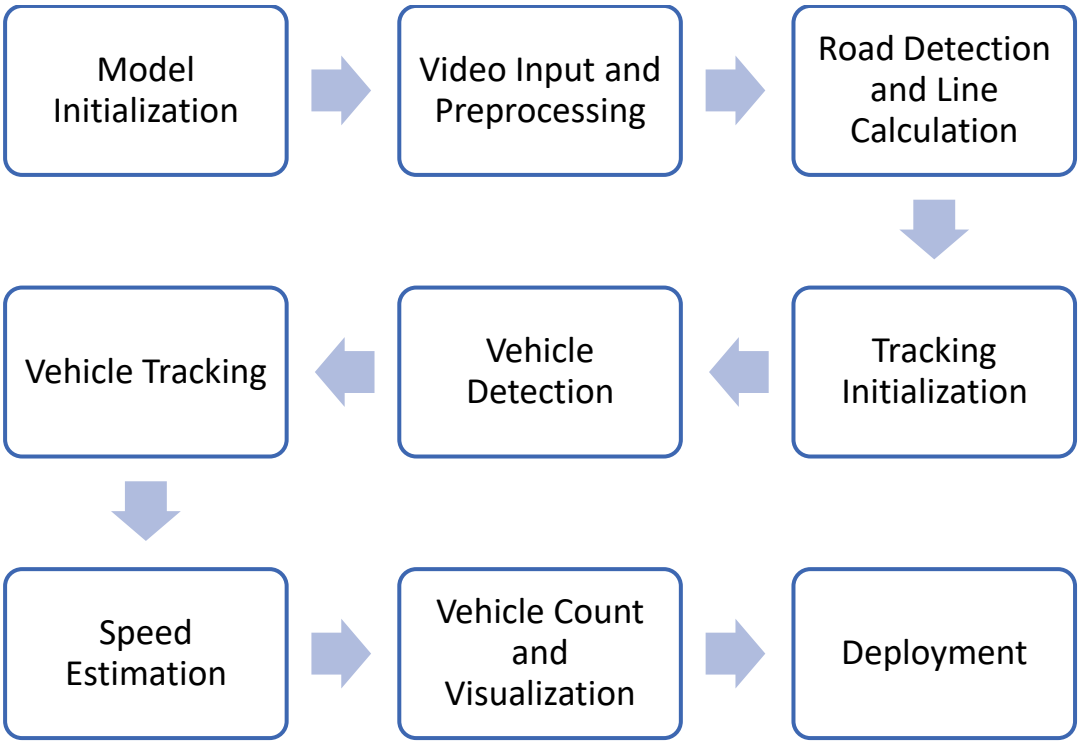

Fig. 1 Workflow of Methodology Used

1.3 SYSTEM REQUIREMENT

HARDWARE REQUIREMENT:

PROCESSOR	: Intel Core i5
RAM	: 8 GB
HARD DISK	: 256 GB
SYSTEM TYPE	: 64-bit operating system
GPU	: NVIDIA GeForce GTX 1050

SOFTWARE REQUIREMENT:

OPERATING SYSTEM	: Windows 10
ENVIRONMENT	: Visual Studio Code
LANGUAGE	: Python 3.8
FRONT END	: Streamlit ==1.10.0
BACK END	: OpenCV, Ultralytics YOLO ==8.0
DEPLOYMENT	: Local Machine Host

CHAPTER – II
METHODOLOGY

2.1 PYTHON:

Python, created by **Guido van Rossum** in the late 1980s, is **a high-level, interpreted programming language** renowned for its simplicity and readability. Its syntax emphasizes code clarity through indentation, making it accessible to programmers. Python excels in **data science and machine learning** with libraries like NumPy, Pandas, and TensorFlow. Its versatility extends across domains, including **automation, scripting, scientific computing, game development, and desktop applications**. With an extensive ecosystem and cross-platform compatibility, Python offers rapid prototyping and seamless integration. Its powerful capabilities and active community have made Python a go-to language for diverse projects, fostering its widespread adoption and continuous growth.

2.1.1 DEPENDENCIES:

Ultralytics:

- This is a Python package that provides pre-trained models and utilities for object detection and instance segmentation tasks, including the YOLOv8 model used in this project.

OpenCV:

- OpenCV is a popular open-source computer vision library that provides a wide range of functions and tools for image and video processing tasks, such as reading and displaying video frames.

Pandas:

- Pandas is a data manipulation and analysis library in Python, which is used in this project to handle the bounding box data returned by the object detection model.

NumPy:

- NumPy is a fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, which are commonly used in computer vision and image processing tasks.

time:

- This is a built-in Python module that provides functions for working with time values, which are used in this project to calculate the elapsed time for vehicle speed estimation.

math:

- Another built-in Python module that provides access to mathematical functions, such as the dist function used for calculating distances.

tracker.py:

- This is a custom Python module created for this project, which contains the implementation of the vehicle tracking algorithm used to assign unique IDs to detected vehicles and track their movements across frames.

2.2 MODEL INITIALIZATION:

The project utilizes **the YOLOv8 object detection model**, which is initialized using the YOLO class from the Ultralytics library. We load the **pre-trained YOLOv8 small (v8s)** model weights from the yolov8s.pt file. This model is a lightweight version of the YOLOv8 architecture, designed for real-time object detection tasks while balancing speed and accuracy. The Ultralytics library provides a convenient interface for loading and using various YOLO models, including YOLOv8, which is based on deep learning techniques. By initializing the model, the code can leverage the pre-trained weights to **detect objects, specifically vehicles**, in the video frames for tracking and speed estimation purposes.

2.2.1 YOLOv8 (You Only Look Once):

YOLOv8 is the latest iteration in the YOLO series of real-time object detectors, offering cutting-edge performance in terms of accuracy and speed. Building upon the advancements of previous YOLO versions, YOLOv8 introduces new features and optimizations that make it an ideal choice for various object detection tasks in a wide range of applications.

Model Training:

The YOLOv8 model is trained on the **COCO (Common Objects in Context)** dataset, which is a large-scale dataset of images containing common objects in real-world scenes. This dataset is used to train the model to recognize and detect a wide range of objects, including people, animals, vehicles, and other common objects. In addition to object detection, the YOLOv8 model can also be used for **image classification and instance segmentation**. This makes it a versatile tool for a wide range of computer vision applications.

YOLOv8s:

YOLO v8s is an evolution of the YOLO algorithm, designed to improve object detection performance by enhancing speed and accuracy. The "s" in YOLO v8s typically denotes a smaller version of the model, optimized for faster inference while maintaining high detection accuracy.

2.3.2 Architecture:

YOLOv8 utilizes a convolutional neural network that can be divided into three main parts:

- the backbone
- the neck
- the head.

Backbone Network:

At the core of YOLOv8 lies the custom **CSPDarknet53** backbone, a variant of Darknet architecture. This backbone is responsible for **feature extraction from input images**. A notable feature of CSPDarknet53 is the introduction **of cross-stage partial connections**, which enhance information flow between different stages of the network and improve gradient flow during training. These connections play a crucial role in boosting the model's ability to understand complex patterns and relationships within images.

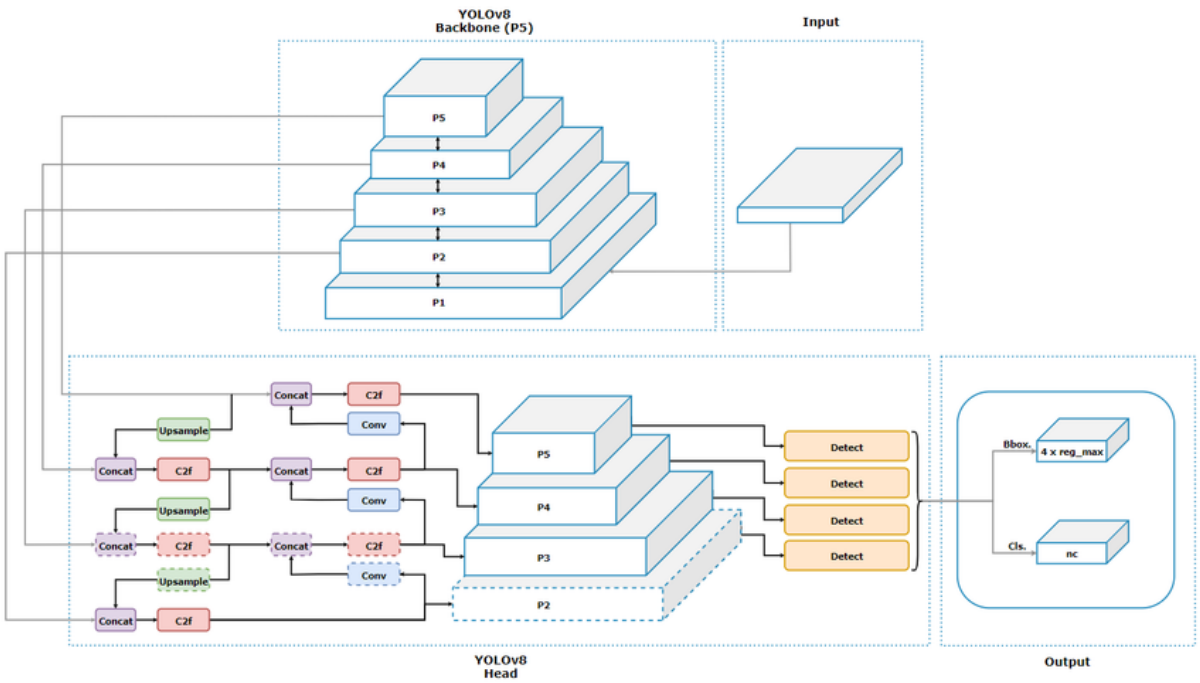

Fig. 2

Darknet Architecture:

The Darknet architecture is a lightweight and fast neural network architecture primarily used for object detection, recognition, and classification tasks in the realm of computer vision. It is characterized by its deep convolutional layers that efficiently process visual data. Darknet's structure involves a **series of convolutional layers grouped into blocks**, each performing feature extraction at different scales. This architecture is known for its efficiency and speed, achieved through **smaller filter sizes** in convolutional layers and techniques like max-pooling for down sampling feature maps.

Cross-stage partial connections:

Cross-stage partial connections are a pivotal component in the YOLOv8 architecture, enhancing information propagation within the network. These connections play a crucial role in facilitating augmented information flow between different stages of the model, thereby improving gradient flow during training. By incorporating cross-stage partial connections, YOLOv8 optimizes the communication between layers, enabling more efficient learning and enhancing the model's ability to understand complex patterns and relationships within images.

CSPDarknet53:

This refers specifically to a variant of the Darknet architecture that incorporates the CSP design element. CSPDarknet53 is composed of multiple layers of **convolutional, pooling, and activation functions**, arranged in a hierarchical manner to extract features from input images. The "53" in the name typically indicates the number of layers in the network. CSPDarknet53 is often used as a backbone network in object detection tasks.

In object detection, the goal is to identify and locate objects within an image. CSPDarknet53 serves as the feature extractor, processing the input image and extracting relevant features that can be used by subsequent layers to detect objects. CSPDarknet53 combines the efficiency of the Darknet architecture with the improved information flow of the CSP design, making it a powerful tool for various computer vision tasks, especially object detection

Neck Structure:

The neck network in YOLOv8 is responsible for **aggregating and combining features** from different layers of the backbone network. This is achieved using **Feature Pyramid Networks** (FPNs) and **Cross-Stage Partial** (CSP) modules. FPNs help capture multi-scale information, allowing the model to detect objects of various sizes, while the CSP modules further enhance the feature extraction and combination process.

Head Structure:

The head network in YOLOv8 is the final part of the architecture, responsible for the actual object detection and instance segmentation tasks. It takes the features from the neck network and outputs the bounding boxes, object classes, and segmentation masks. YOLOv8 employs a novel head design that combines object detection and instance segmentation in a single, end-to-end network, improving the model's efficiency and overall performance.

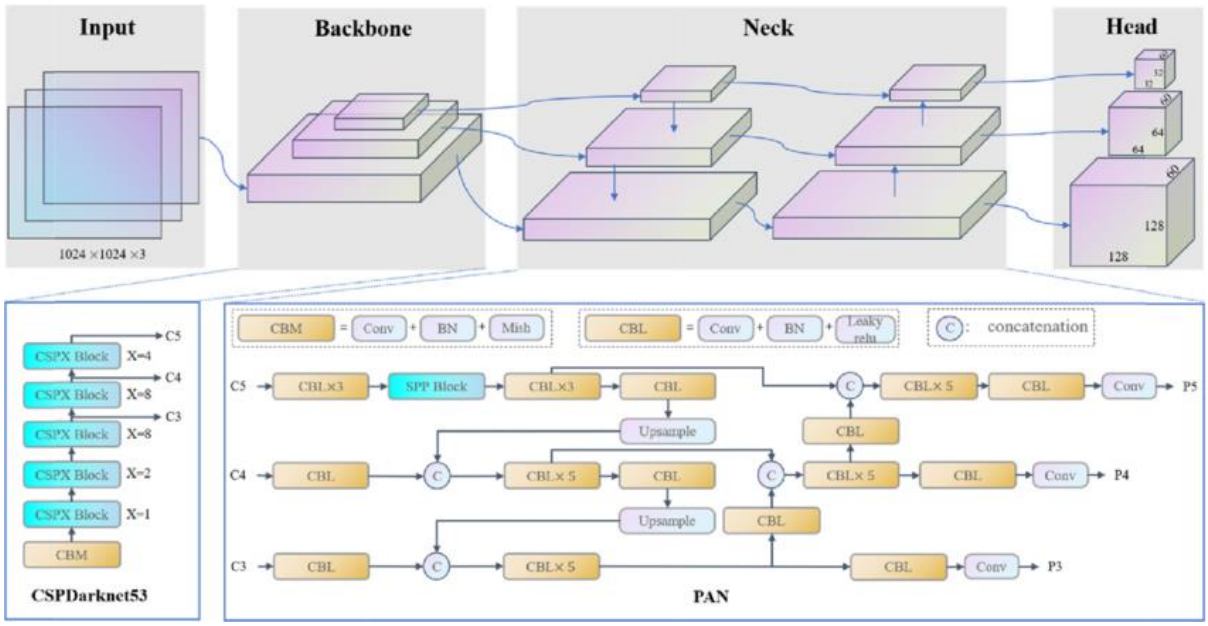

Fig. 3

Feature Pyramid Networks:

Feature Pyramid Networks (FPN) are neural networks used in computer vision for object detection, specifically designed to **combine features from different levels of a convolutional network** to enhance object detection at various scales. FPNs create a feature pyramid by extracting features from different levels of the network, allowing for the creation of strong semantic feature maps at each scale.

This approach enables the network to effectively detect objects of different sizes by combining low-level detail-aware features with high-level context-aware features. FPNs have been shown to significantly improve object detection performance and have been integrated into various object detection architectures, surpassing previous state-of-the-art models in terms of accuracy and efficiency.

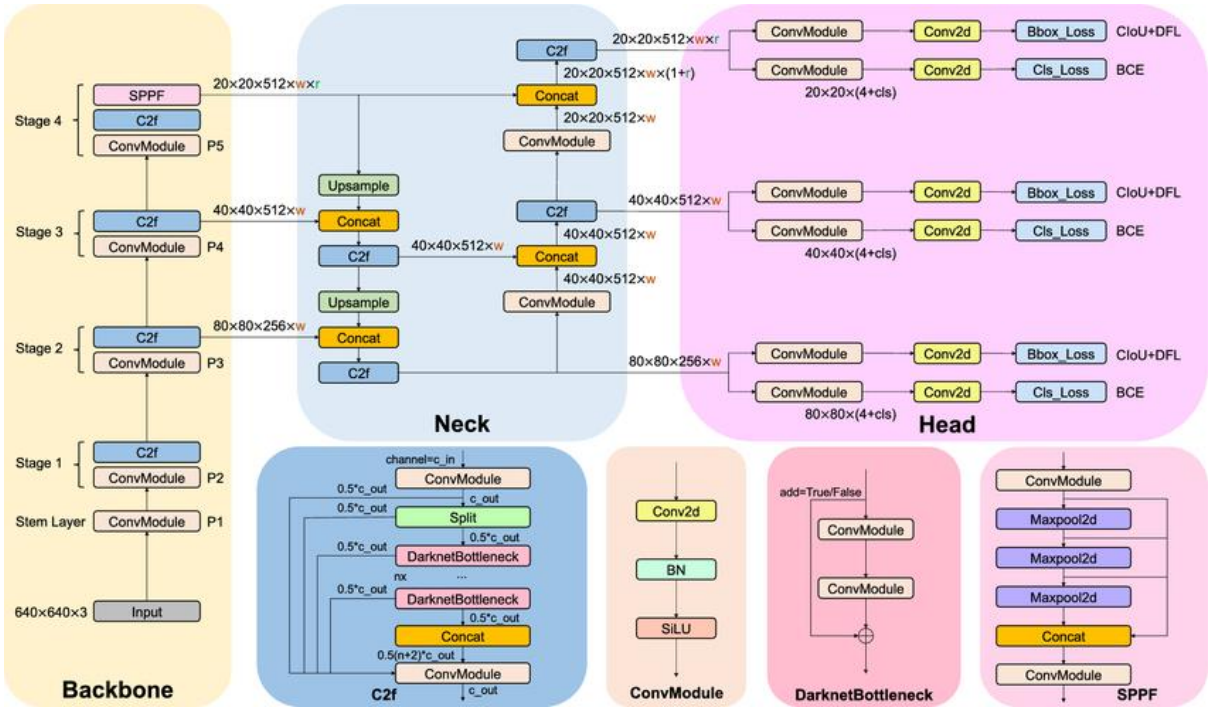

Fig. 4

YOLOv8 architecture stands out for its innovative design, incorporating advanced features like **cross-stage partial connections, PANet, and dynamic anchor assignment** to deliver exceptional object detection capabilities. By combining these elements effectively, YOLOv8 sets new standards in real-time object detection, making it a versatile and powerful tool for a wide range of applications.

Key Features of YOLO v8s:

1. **Single Shot Detection:** YOLO v8s follows the single-shot detection paradigm, where it processes the entire image in a single forward pass to predict bounding boxes and class probabilities simultaneously.
2. **Anchor Boxes:** YOLO v8s uses anchor boxes to predict object locations and sizes. These anchor boxes are predefined shapes that help the model better localize objects of different scales and aspect ratios.
3. **Feature Extraction:** YOLO v8s employs a deep neural network architecture to extract features from the input image, enabling the model to learn discriminative features for object detection.

4. **Class Prediction:** The model predicts the class probabilities for each bounding box, indicating the likelihood of the detected object belonging to a specific class from a predefined list.
5. **Efficient Inference:** YOLO v8s is optimized for efficient inference on various hardware platforms, making it suitable for real-time applications like video analysis and surveillance.

Performance and Applications:

- YOLO v8s offers a balance between **speed and accuracy**, making it suitable for applications requiring real-time object detection.
- Common applications of YOLO v8s include **surveillance systems, autonomous vehicles, traffic monitoring, and industrial automation**.

Model	Filenames	Task	Inference	Validation	Training	Export
YOLOv8	`yolov8n.pt` `yolov8s.pt` `yolov8m.pt` `yolov8l.pt` `yolov8x.pt`	Detection	✅	✅	✅	✅

Fig. 5 YOLOv8 Supported tasks and modes

2.3 VIDEO INPUT AND PREPROCESSING:

The code captures the video using **the cv2.VideoCapture** function. This function initializes the video capture device, which can be a file or a live video stream. Once the video capture is set up, the project enters a loop that iterates through each frame of the video. Before processing the frame, a frame skipping mechanism is implemented, where only every third frame is processed. After skipping the unnecessary frames, the current frame is resized to a fixed resolution of **1020x500 pixels**. This step is crucial for ensuring consistent processing across all frames and maintaining a reasonable aspect ratio for the output video. By resizing the frames, the project prepares the data for the subsequent steps, such as road detection, vehicle detection, and speed estimation.

```
#Model - Yolo Version 8s
model=YOLO('yolov8s.pt')

#Video Input
cap=cv2.VideoCapture('veh2.mp4')
```

Fig. 6 Video Input

```
while True:
    ret, frame = cap.read()
    if not ret:
        break

    count += 1
    if count % 3 != 0:
        continue

    frame = cv2.resize(frame, (1020, 500))
```

Fig. 7 Video capture and Preprocessing

2.4 ROAD DETECTION AND LINE CALCULATION:

This step aims to identify the road area within the video frames and determine the coordinates of the upper and lower lines that will be used to estimate the vehicles' speeds. It starts by defining a **region of interest (ROI)** within the frame, which is a subset of the entire frame. In this case, the ROI is the lower portion of the frame, specifically the area between the 300th and 500th rows.

This region is chosen based on the assumption that the road is located in the lower part of the frame. Once the ROI is defined, the function applies **Canny edge detection** to the ROI. Canny edge detection is a widely used algorithm for detecting sharp changes in pixel intensity, which often correspond to the edges of objects in an image. The result of the Canny edge detection is a binary image where the edges are highlighted.

Next, the function uses the **OpenCV cv2.HoughLinesP** function to detect the lines within the Canny edge image. The HoughLinesP function is a probabilistic version of the Hough transform, which is a technique for detecting lines in an image. The function returns a set of line segments that represent the detected lines in the image. The function then analyses the detected lines to find the centre position of the road. It does this by finding the minimum and maximum y-coordinates of the detected lines, and then calculating the average of these values to determine the centre position of the road. This centre position is then converted back to the full frame coordinates by adding the offset used for the ROI.

Once the centre position of the road is determined, the function calculates the positions of the upper and lower lines based on a predefined distance. These lines are then drawn on the frame using OpenCV's cv2.line function, with the upper line in red and the lower line in blue.

If no lines are detected in the ROI, the function returns None for the road centre position, and the process returns to the road detection step. This allows the system to handle cases where the road is not clearly visible or when the video quality is poor.

```python
# Function to detect road area and calculate line coordinates
def detect_road_and_lines(frame):
    # Define region of interest (ROI) for road detection
    roi = frame[300:500, :]

    # Apply lane detection algorithm to detect road area
    edges = cv2.Canny(roi, 50, 150)
    lines = cv2.HoughLinesP(edges, 1, np.pi / 180, 50, minLineLength=50, maxLineGap=5)

    # Check if any lines are detected
    if lines is None or len(lines) == 0:
        return None, None

    # Calculate center position of the road
    line1_y = np.min([lines[:, :, 1].min(), lines[:, :, 3].min()])
    line2_y = np.max([lines[:, :, 1].max(), lines[:, :, 3].max()])
    road_center_y = (line1_y + line2_y) // 4

    # Convert road center position to full frame coordinates
    road_center_y_full_frame = road_center_y + 300

    return int(road_center_y_full_frame)
```

Fig. 8 Road Detection and Line calculation

2.5 TRACKING INITIALIZATION:

Tracking initialization involves setting up the tracker to monitor and follow objects, specifically vehicles, in a video stream. The Tracker class is responsible for this task. We define a Tracker class that tracks objects based on their center points.

Initialization:

- The Tracker class initializes with two attributes:
  - center_points: a dictionary to store the centre positions of detected objects.
  - id_count: an integer to keep track of the count of detected object IDs.

Update Method:

- The update method takes a list of object rectangles (objects_rect) as input.
- It iterates through each rectangle to extract the centre point (cx, cy) of the object.
- For each object, it checks if it has been detected before by comparing its centre point with existing ones in center_points.
  - If a similar object is found (based on a distance threshold of 35), it updates the centre point and appends the object's bounding box and ID to objects_bbs_ids.

- If it's a new object, it assigns a new ID, updates center_points, and adds the bounding box and ID to objects_bbs_ids.

- After processing all objects, it cleans up center_points by removing IDs that are no longer in use.
- Finally, it updates center_points with the remaining IDs and returns objects_bbs_ids, which contains bounding boxes and corresponding IDs for all detected objects.

This implementation efficiently tracks objects by their center points, assigning unique IDs to new objects and updating existing ones based on proximity. It ensures accurate tracking while managing the assignment and removal of object IDs effectively.

```python
class Tracker:
    def __init__(self):
        # Store the center positions of the objects
        self.center_points = {}
        # Keep the count of the IDs
        # each time a new object id detected, the count will increase by one
        self.id_count = 0


    def update(self, objects_rect):
        # Objects boxes and ids
        objects_bbs_ids = []

        # Get center point of new object
        for rect in objects_rect:
            x, y, w, h = 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 < 35:
                    self.center_points[id] = (cx, cy)
#                    print(self.center_points)
                    objects_bbs_ids.append([x, y, w, h, id])
                    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])
                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 = 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
```

Fig. 9 Tracker Initialization

CHAPTER – III
VEHICLE TRACKING AND SPEED ESTIMATION

3.1 VEHICLE DETECTION:

Object detection plays a crucial role in various applications, including surveillance, autonomous vehicles, image analysis, and more. By accurately identifying and localizing objects in images or videos, object detection enables automated systems to understand and interact with their environments effectively. In this project object detection is achieved using YOLO (You Only Look Once), a popular deep learning algorithm known for its speed and accuracy in detecting objects in real-time.

1. **Model Selection:** The code utilizes YOLO version 8s (yolov8s.pt) for object detection, which is a pre-trained model capable of recognizing a wide range of objects.
2. **Detection Process:** The model processes video frames from a source and predicts objects present in each frame.
3. **Bounding Box Generation:** Detected objects are represented by bounding boxes that outline their location within the frame.
4. **Class Labelling:** Each detected object is assigned a class label from a predefined list, such as 'car', 'person', 'bicycle', etc., based on the trained model's classification capabilities.
5. **Visualization:** The code visualizes the detected objects by drawing bounding boxes around them and displaying relevant information like detected class.

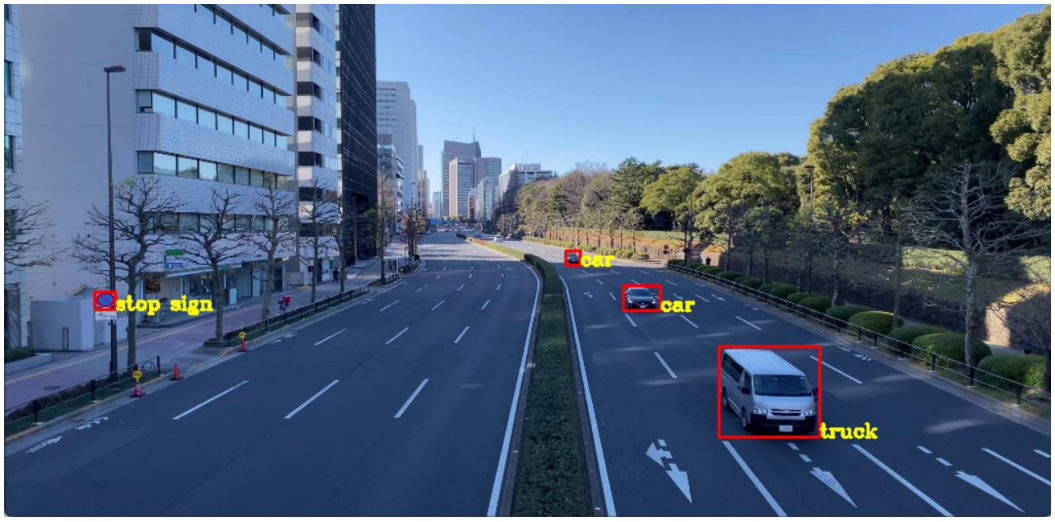

Fig. 10 Object Detection

3.2 VEHICLE TRACKING:

Vehicle tracking is a crucial component of the overall system designed for vehicle speed detection and counting. Vehicle tracking is accomplished through a combination of object detection and tracking techniques applied to video streams. Initially, vehicles are detected in each frame using the YOLO (You Only Look Once) object detection algorithm, which identifies vehicles based on predefined features. Upon initializing the tracker, it creates dictionaries vh_down and vh_up to track vehicles moving in different directions across defined lines.

- **vh_down Dictionary:** This dictionary stores the time when a vehicle crosses a lower line (cy1) in the video frame.
- **vh_up Dictionary:** Similarly, this dictionary records the time when a vehicle crosses an upper line (cy2) in the video frame.

When a vehicle is detected crossing the lower line (cy1), its ID and the current time are stored in vh_down. Likewise, when a vehicle crosses the upper line (cy2), its ID and the current time are saved in vh_up. These dictionaries facilitate the calculation of the vehicle's speed based on the time taken to travel a predefined distance. By utilizing these dictionaries and the tracker, the code effectively initializes and manages the tracking process. The tracking process involves continuously updating the positions of vehicles as they move within the video stream. By accurately tracking vehicles, the system can determine when they cross specific points of interest, enabling the calculation of their speeds and counting their movements in different directions.

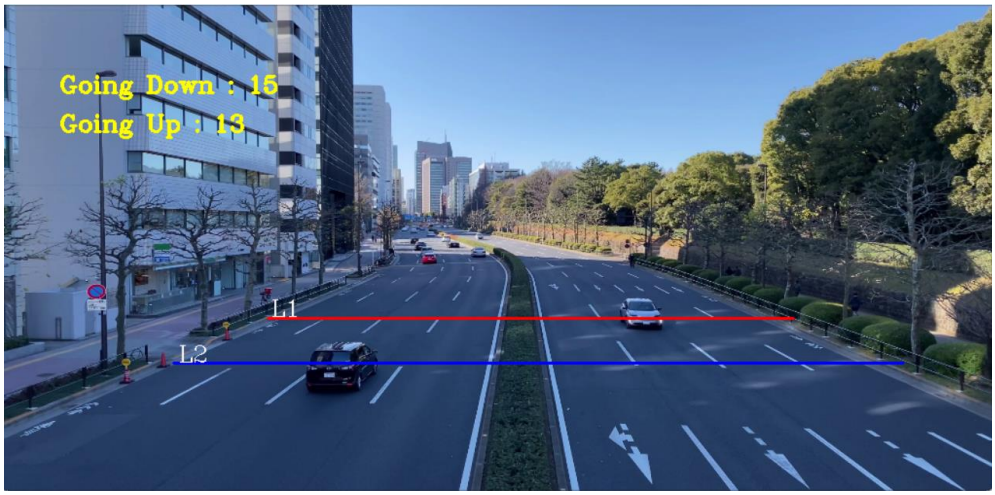

Fig. 11 Vehicle Tracking

3.3 SPEED ESTIMATION:

Vehicle speed estimation is a fundamental aspect of the system designed for vehicle speed estimation and tracking. The speed estimation process involves analysing the movement of vehicles detected in the video stream and calculating their speeds based on the time taken to traverse predefined distances. After vehicles are detected and tracked using object detection and tracking techniques, the system monitors their movements as they cross predefined lines within the video frame. By tracking the time each vehicle takes to travel between these lines, the system can calculate their speeds using the basic formula:

speed = distance / time

To ensure accurate speed estimation, the system requires a predefined distance between the lines and precise timing measurements. The distance between the lines serves as the reference for calculating vehicle speeds, while the timing measurements are obtained by monitoring the passage of vehicles across these lines. Once the time taken by each vehicle to traverse the distance between the lines is recorded, the system calculates their speeds in kilometres per hour (Km/h) using simple mathematical calculations.

```
bbox_id = tracker.update(list)
for bbox in bbox_id:
    x3, y3, x4, y4, id = bbox
    cx = int(x3 + x4) // 2
    cy = int(y3 + y4) // 2

    bbox_height = y4 - y3  # Height of the bounding box
    offset = int(bbox_height * 0.2)
    cy1 = line1_y
    cy2 = line2_y

    # Going DOWN
    if cy1 < (cy + offset) and cy1 > (cy - offset):
        vh_down[id] = time.time()
    if id in vh_down:
        if cy2 < (cy + offset) and cy2 > (cy - offset):
            elapsed_time = time.time() - vh_down[id]
            if counter.count(id) == 0:
                counter.append(id)
                distance = 10  # meters
                a_speed_ms = distance / elapsed_time
                a_speed_kh = a_speed_ms * 3.6
                cv2.rectangle(frame, (x3, y3), (x4, y4), green_color, 2)
                cv2.circle(frame, (cx, cy), 4, yellow_color, -1)
                cv2.putText(frame, str(id), (x3, y3), cv2.FONT_HERSHEY_COMPLEX, 0.6, yellow_color, 1)
                cv2.putText(frame, str(int(a_speed_kh)) + 'Km/h', (x4, y4), cv2.FONT_HERSHEY_COMPLEX, 0.8,
                            yellow_color, 2)
```

Fig. 12 Speed Estimation

CHAPTER – IV
RESULT AND DISCUSSION

The vehicle tracking and speed estimation system developed in this project has demonstrated promising results in accurately detecting and monitoring the speeds of vehicles in a video feed. The combination of YOLO-based vehicle detection, the custom Tracker class, and the speed estimation logic has enabled the system to provide valuable insights into the traffic dynamics.

4.1 VEHICLE COUNT AND VISUALIZATION:

Visualization plays a crucial role in presenting the results of vehicle detection, tracking, speed estimation, and counting in a clear and understandable manner. The visualization components are designed to provide real-time insights into the movement and behaviour of vehicles within the video stream.

The visualization includes several key elements:

- **Bounding Boxes**: Detected vehicles are enclosed within bounding boxes, allowing users to visually identify their locations and sizes within each frame of the video stream.
- **Line Markers**: Predefined lines are drawn on the video frame to indicate specific points of interest, such as the beginning and end of the speed calculation zone. These lines serve as reference points for tracking vehicle movements and estimating their speeds.
- **Text Annotations**: Text annotations are added to the video frame to display relevant information, including vehicle IDs, calculated speeds, and counts of vehicles moving in different directions. These annotations provide additional context and insights into the detected vehicles' characteristics and behaviours.
- **Color-coded Visuals**: Different colours are used to differentiate between various visualization elements, such as bounding boxes, lines, and text annotations. This colour coding enhances the clarity and readability of the visualization, making it easier for users to interpret the information presented.

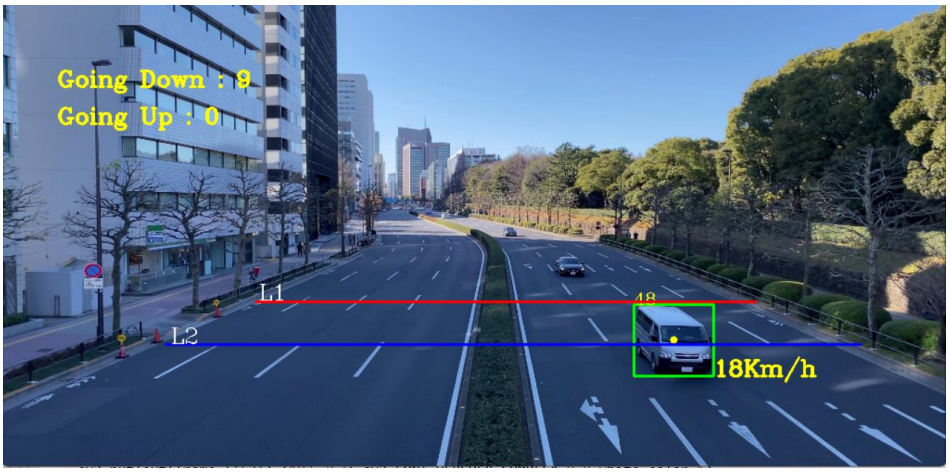

Fig. 13

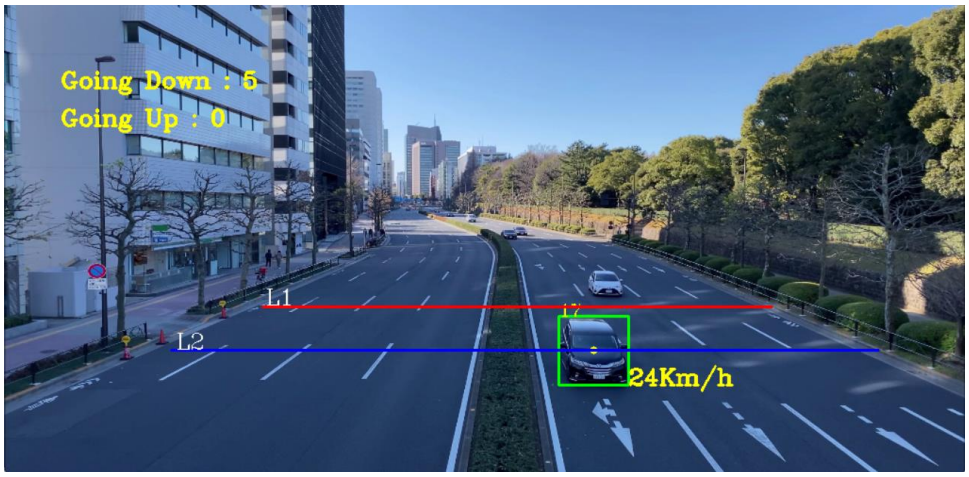

Fig. 14

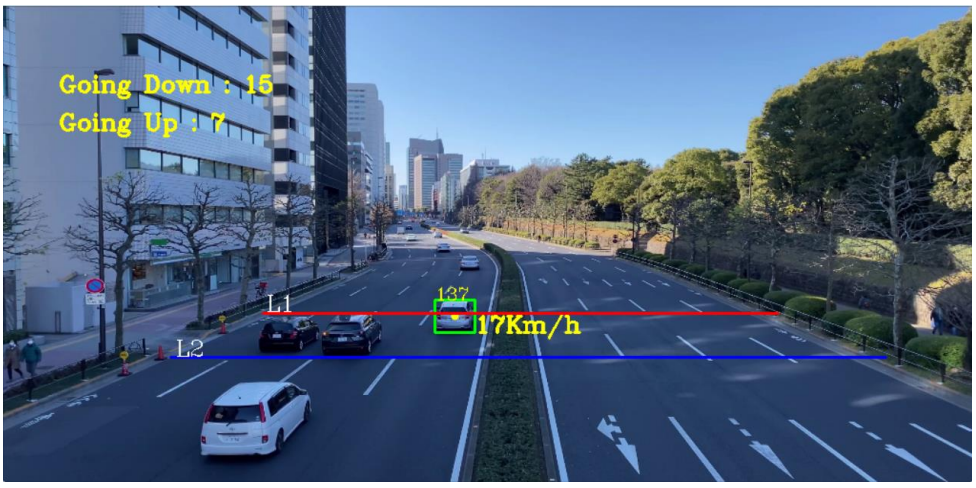

Fig. 15

4.2 FACTORS INFLUENCE THE PERFORMANCE OF VEHICLE TRACKING AND SPEED ESTIMATION:

- Video Quality and Resolution
- Camera Placement and Angle
- Traffic Density and Complexity
- Computational Resources

4.3 LIMITATIONS:

- Sensitivity to Occlusions and Complex Interactions
- Dependency on Road Detection Accuracy
- Scalability and Integration Challenges

4.4 FUTURE WORK:

- Incorporating more advanced tracking algorithms, such as those based on Kalman filters, particle filters, or deep learning-based methods, can enhance the system's ability to handle occlusions and complex vehicle interactions.
- Exploring more sophisticated road detection and line calculation techniques, potentially leveraging machine learning or computer vision advancements, can improve the reliability and accuracy of the reference lines used for speed estimation.
- Exploring scalable system architectures, such as multi-camera setups or distributed processing frameworks, can enable the deployment of the system in larger-scale traffic monitoring applications.
- Integrating the vehicle tracking and speed estimation system with existing traffic management infrastructure, such as traffic signals, variable message signs, and intelligent transportation systems, can unlock synergies and enable more comprehensive traffic monitoring and optimization solutions.

CHAPTER – V

MODEL DEPLOYMENT

Deploying a machine learning model, known as model deployment, means integrating a machine learning model and integrate it into an existing production environment where it can take in an input and return an output. The purpose of deploying your model is so that you can make the predictions from a trained ML model available to others, whether that be users, management, or other systems.

There are various sources to deploy the model that we developed. Some of the easiest deployment sources from Python are **Streamlit, Flask API** etc. ML models will usually be deployed in an offline or local environment, so will need to be deployed with live data.

STREAMLIT:

Streamlit is an open-source Python library that simplifies the process of turning data scripts into interactive web applications. It is designed to be easy to use and allows data scientists and developers to create web applications for data analysis, machine learning, and visualization with minimal effort. Streamlit provides a high-level API that enables the creation of web applications using a few lines of Python code. With Streamlit, you can quickly prototype and iterate on your data applications. The library abstracts away much of the complexity involved in web development, allowing users to focus on the logic and visualization aspects of their applications.

Run the Python file in the terminal using the Streamlit run command. This will generate a link. You can click the link which will take you to the web app.

You can now view your Streamlit app in your browser.

Local URL: http://localhost:8502

Network URL: http://192.168.1.2:8502

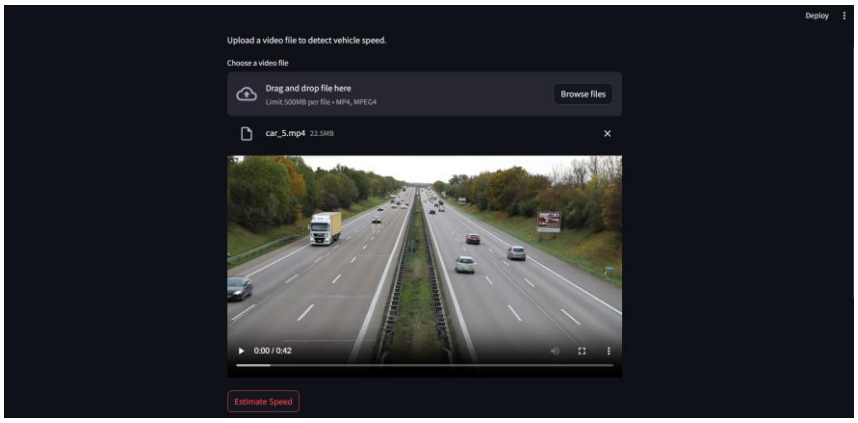

Fig. 16 Web Interface

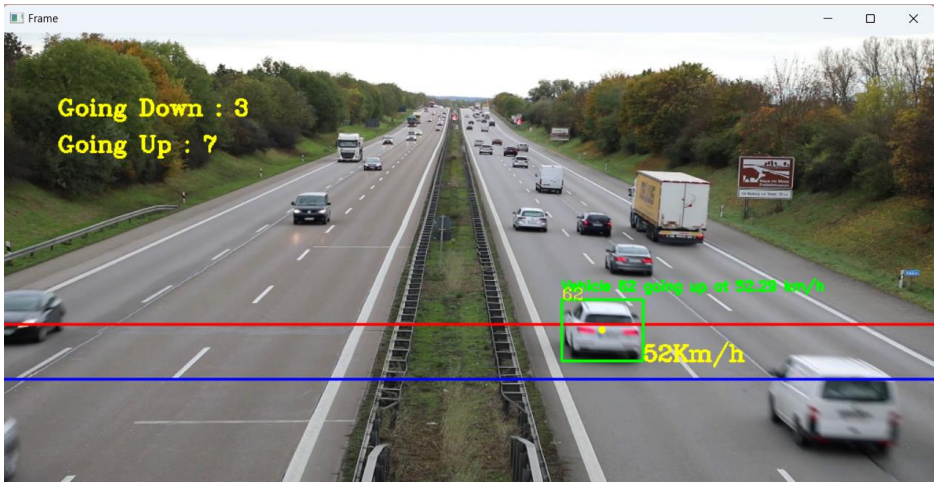

Fig. 17

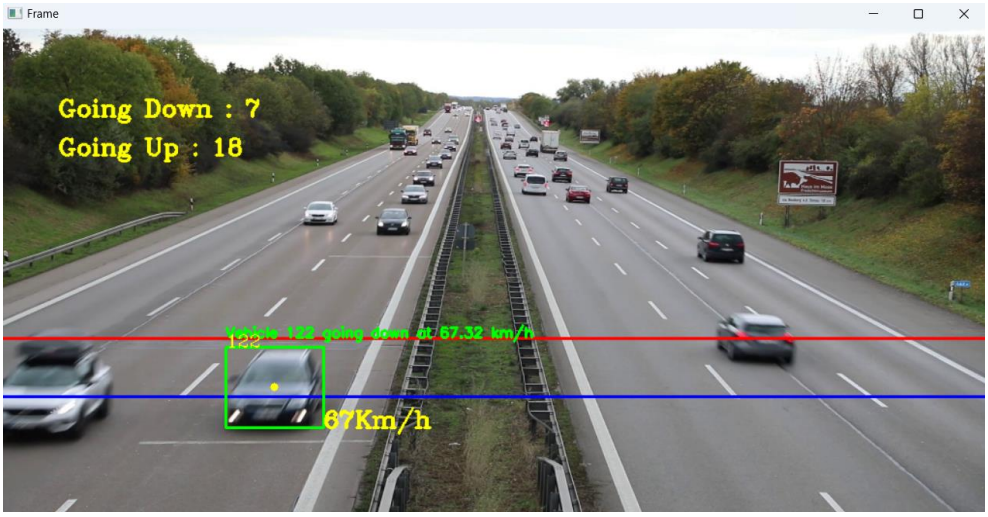

Fig. 18

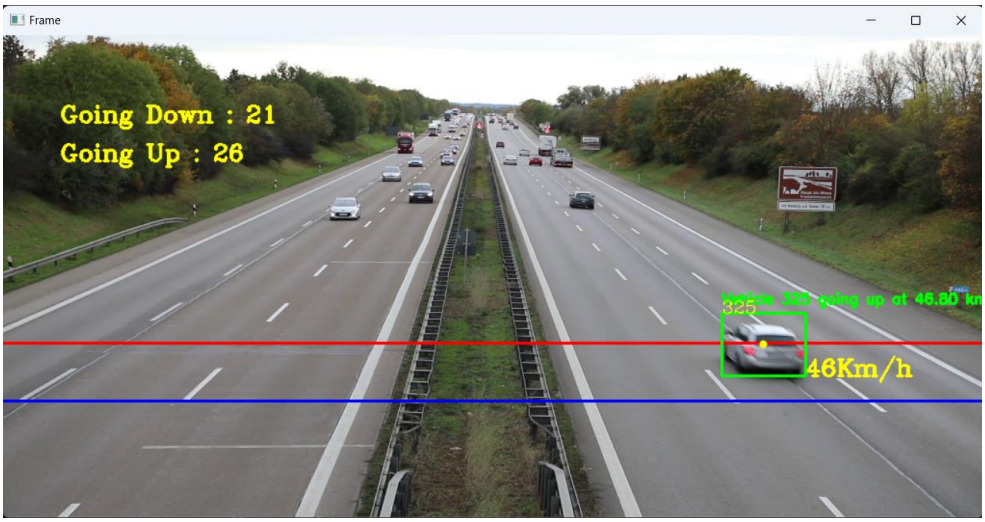

Fig. 19

CHAPTER - VI
CONCLUSION

The vehicle tracking and speed estimation system developed in this project has demonstrated its potential to enhance traffic monitoring and analysis capabilities through the application of advanced computer vision techniques. By combining YOLO-based vehicle detection, a custom Tracker class for object tracking, and speed estimation logic, the system is able to accurately detect, track, and calculate the speeds of vehicles in a video feed. The results obtained from the system showcase its ability to provide valuable insights into traffic dynamics, such as the number of vehicles traveling in different directions and their corresponding speeds. This information can be leveraged to support a wide range of traffic management, transportation planning, and road safety initiatives. As technology continues to advance, the integration of such innovative solutions into transportation infrastructure will play a crucial role in shaping the future of smart and sustainable cities.

BIBLIOGRAPHY

1. https://pyimagesearch.com/2019/12/02/opencv-vehicle-detection-tracking-and-speed-estimation/
2. https://blog.roboflow.com/estimate-speed-computer-vision/
3. https://paperswithcode.com/paper/detection-of-3d-bounding-boxes-of-vehicles
4. https://paperswithcode.com/paper/vehicle-speed-estimation-using-computer
5. https://www.labellerr.com/blog/understanding-yolov8-architecture-applications-features/
6. https://medium.com/@juanpedro.bc22/detailed-explanation-of-yolov8-architecture-part-1-6da9296b954e
7. https://github.com/AarohiSingla/Speed-detection-of-vehicles
8. https://github.com/freedomwebtech/yolov8counting-trackingvehicles