Vehicle Monitoring System In A Gated Community

A report submitted in partial fulfillment of the requirements for the Degree of

Bachelor of Technology

in

Computer Science and Engineering

By

1	Sura Ashwanth Reddy	2111CS010073
2	Bhavana Anusha	2111CS010052
3	Mohammed Arbaz Ali	2111CS010057
4	Shaik Azeed	2111CS010077

Under the esteemed guidance of

Dr. Rambabu.P
Associate Professor

Department of Computer Science & Engineering

School of Engineering

MALLA REDDY UNIVERSITY

Maisammaguda, Dulapally, Hyderabad, Telangana 500100

2024-25

MALLA REDDY UNIVERSITY
(Telangana State Private Universities Act No.13 of 2020 and G.O.Ms.No.14, Higher Education (UE) Department)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the project report entitled "**Vehicle Monitoring System In A Gated Community**", submitted by **Sura Ashwanth Reddy (2111CS010073), Bhavana Anusha (2111CS010052), Mohammad Arbaz ali (2111CS010057), Shaik Azeed (2111CS010077),** towards the partial fulfillment for the award of Bachelor's Degree in Technology from the Department of Computer Science and Engineering, Malla Reddy University, Hyderabad, is a record of bonafide work done by him/ her. The results embodied inthe work are not submitted to any other University or Institute for award of any degree or diploma.

Internal Guide
Dr. Rambabu.P
Associate Professor

Head of the Department
Dr. Shaik Meeravali

External Examiner

DECLARATION

We hereby declare that the project report entitled "**Vehicle Monitoring System In A Gated Community.**" Has been carried out by us and this work has been submitted to the Department of Computer Science and Engineering, Malla Reddy University, Hyderabad in partial fulfillment of the requirements for the award of degree of Bachelor of Technology. We further declare that this project work has not been submitted in full or part for the award of any other degree in any other educational institutions.

Place: Hyderabad

Date:

Sura Ashwanth Reddy	2111CS010073
Bhavana Anusha	2111CS010052
Mohammad Arbaz Ali	2111CS010057
Shaik Azeed	2111CS010077

ACKNOWLEDGEMENT

We extend our sincere gratitude to all those who have contributed to the completion of this project report. Firstly, we would like to extend our gratitude to **Dr. V. S. K Reddy**, Vice-Chancellor, for his visionary leadership and unwavering commitment to academic excellence.

We would also like to express my deepest appreciation to our project guide **Dr. Rambabu.P**, Associate Professor, whose invaluable guidance, insightful feedback, and unwavering support have been instrumental throughout the course of this project for successful outcomes.

We are also grateful to **Dr. Shaik Meeravali**, Head of the Department of Computer Science and Engineering, for providing us with the necessary resources and facilities to carry out this project.

My heartfelt thanks also go to **Dr. Harikrishna Kamatham**, Dean ,School of Engineering for his guidance and encouragement.

We are deeply indebted to all of them for their support, encouragement, and guidance, without which this project would not have been possible.

Sura Ashwanth Reddy	2111CS010073
Bhavana Anusha	2111CS010052
Mohammad Arbaz Ali	2111CS010057
Shaik Azeed	2111CS010077

ABSTRACT

This project aims to develop a deep learning-based system for monitoring vehicle movement in gated communities to enhance security and operational efficiency. Utilizing the YOLO (You Only Look Once) algorithm for real-time object detection and tracking, the system can accurately identify and monitor vehicles at entry and exit points, as well as within community spaces. The implementation leverages tools like TensorFlow and OpenCV for model training and image processing, with an interactive Streamlit interface for a seamless monitoring experience.

Key features include object detection to identify vehicles, image classification to categorize them (e.g., cars, bikes, trucks), and vehicle tracking to count entries and exits. This data provides valuable insights for applications such as security surveillance, traffic management, and resource optimization. By integrating YOLO's high-speed detection capabilities with a user-friendly Streamlit dashboard, the system enables rapid response to potential security threats and contributes to safer residential environments.

By combining the efficiency of YOLO's detection algorithms with a streamlined interface, the system offers a high level of responsiveness and accuracy. This solution is ideal for applications in community security, parking management, and emergency response, empowering gated communities to proactively manage potential security threats and ensure a safer, well-regulated environment for residents.

CONTENTS

LIST OF FIGURES

CHAPTER - 1

INTRODUCTION

1.1 Introduction

- In modern gated communities, ensuring the security of residents and managing the flow of vehicles are top priorities. Traditional manual or semi-automated monitoring methods often struggle to provide accurate and real-time tracking, making it difficult to respond to security threats or optimize traffic management. This project, "Vehicle Movement Monitoring in a Gated Community Using Deep Learning," leverages the power of machine learning and real-time data processing to provide an efficient and highly accurate vehicle surveillance system.

- Utilizing the YOLO (You Only Look Once) algorithm, renowned for its high-speed object detection capabilities, the system detects and tracks vehicles as they enter, exit, or move within the community. The project employs a deep learning-based approach, with YOLO responsible for vehicle detection and classification based on type, including cars, bikes, and trucks. The system processes video feeds from cameras strategically positioned at key points within the community and displays real-time results on an intuitive Streamlit dashboard, enabling quick access to vehicle logs and detection alerts.

- By automating vehicle monitoring, this system not only enhances security but also provides valuable insights into vehicle flow, traffic patterns, and resource usage, supporting both immediate response to security incidents and data-driven decision-making for long-term community management.

1.2 Objective

- Develop an automated vehicle monitoring system that ensures the security and operational efficiency of gated communities.
- Accurately detect and classify vehicles in real time as they move within the community, using the YOLO algorithm for high-speed and reliable object detection.
- Track entry and exit times and log vehicle movement to identify any unusual patterns or unauthorized entries.
- Provide a user-friendly, real-time interface via Streamlit, displaying logs, vehicle categories, and alerts for quick and effective monitoring.
- Collect and analyze data on vehicle movement patterns to assist community managers with traffic management, resource optimization, and security insights.

1.3 Overview

This project, "Vehicle Movement Monitoring in a Gated Community Using Deep Learning," offers a sophisticated, automated solution for managing and analyzing vehicle movements within residential communities. By combining YOLO-based real-time detection with a user-friendly interface on Streamlit, the system brings both security and data insights to community administrators, enhancing the quality of monitoring while minimizing manual intervention.

The system operates through a series of interdependent components, each optimized to process data swiftly and accurately.
Here's a high-level breakdown of the system's components and their functions:

1. **Data Collection**: Video feeds from surveillance cameras capture the movement of vehicles within the community. These feeds are continuously processed to detect and identify vehicles in real time.

2. **Vehicle Detection and Classification**: Using YOLO, the system quickly identifies and classifies different types of vehicles (cars, bikes, trucks, etc.), logging essential details such as entry and exit times.

3. **Data Processing and Storage**: Each detection event is processed and stored, building a database of vehicle movements that community managers can reference for tracking trends, generating reports, and implementing policies.

4. **Real-Time Monitoring Interface**: Through a Streamlit-based dashboard, users can monitor vehicle activity as it happens. The interface displays categorized logs, traffic patterns, and alert notifications, providing an easy-to-navigate visual of vehicle data within the community.

5. **Analysis and Insights**: The stored data allows for comprehensive analysis, uncovering valuable insights such as peak traffic times, vehicle density, and frequency of specific vehicle types, which can aid in predictive modeling for security and operational needs.

CHAPTER–2
LITERATURE SURVEY

In recent years, gated communities and urban residential complexes have increasingly adopted technological advancements for enhanced security and operational efficiency. Deep learning, computer vision, and real-time data processing have proven to be effective in addressing the challenges of vehicle monitoring in such environments. This literature survey covers key studies, methodologies, and technologies that have informed the development of automated vehicle monitoring systems, with a focus on object detection and tracking techniques, security applications, and real-time implementation.

1. Object Detection in Surveillance Systems

1.1 Evolution of Object Detection Algorithms

Object detection has progressed significantly with advancements in deep learning models. Early methods relied on traditional computer vision techniques, such as feature matching and edge detection (Viola-Jones, 2001; HOG-SVM, 2005), which were effective but limited in their ability to generalize across complex environments. However, with the advent of deep learning, algorithms such as CNNs and YOLO have improved detection accuracy and processing speed, making them ideal for real-time applications.

YOLO (You Only Look Once): The YOLO family of algorithms (YOLOv1 to YOLOv5) represents a significant leap in object detection. Developed by Redmon et al., YOLO uses a single neural network to process an entire image, making detections in one forward pass. Studies demonstrate that YOLOv5, in particular, offers a superior balance between detection accuracy and speed, making it highly suitable for vehicle detection in real-time surveillance (Redmon & Farhadi, 2018).

Application to Vehicle Detection: Several research works have validated the effectiveness of YOLO for vehicle detection in live video feeds. For instance, Krishna et al. (2020) successfully employed YOLOv3 to detect and classify vehicles in highway surveillance footage. The study highlights YOLO's ability to handle high frame rates while accurately identifying multiple classes of vehicles, which is highly relevant to gated community

security where both accuracy and real-time performance are critical.

1.2 DEEP SORT for Object Tracking

While YOLO performs well in detection tasks, tracking objects across frames requires specialized algorithms. DeepSORT (Simple Online and Realtime Tracking) has gained popularity for its robustness in tracking multiple objects in dynamic environments. DeepSORT combines Kalman filtering with YOLO-based detection to maintain an ID for each detected object, enabling continuous tracking as vehicles move across camera views. Research by Bewley et al. (2016) demonstrated DeepSORT's effectiveness in urban traffic monitoring, where it successfully maintained object identities over time, even in complex scenarios with overlapping vehicles.

1.3 Comparisons of Object Detection Models

A comparative study by Wu et al. (2021) evaluated the accuracy and speed of YOLOv5 against Faster R-CNN, SSD, and RetinaNet in vehicle detection tasks. The study found that YOLOv5 outperformed these models in real-time applications, achieving higher accuracy at faster processing speeds. This supports the choice of YOLOv5 for gated community monitoring, where real-time performance is essential for immediate security response.

2. Real-Time Vehicle Tracking for Security Applications

2.1 The Role of Computer Vision in Gated Communities

The application of computer vision in gated communities has evolved from basic motion detection to sophisticated tracking and classification of vehicles. Studies like those by Zhang et al. (2019) emphasize that real-time monitoring systems based on deep learning can significantly reduce security risks. By analyzing entry and exit patterns, these systems provide insights into unusual or unauthorized vehicle movement, which is crucial for enhancing security.

2.2 Security Benefits of Vehicle Classification and Pattern Analysis

Research in vehicle classification has shown that identifying vehicle types can aid in predictive analytics for security (Lee et al., 2020). For instance, the system could flag a large delivery truck entering a residential area during late hours as suspicious. Additionally, analysis of vehicle movement patterns allows administrators to detect

abnormal behavior, such as repeated entries within a short period, which could indicate a potential security threat. Studies by Wang et al. (2020) further show that deep learning-based monitoring systems improve responsiveness to security incidents by automatically alerting security personnel to irregularities.

3. Technology and Tools in Real-Time Monitoring Systems

3.1 YOLO and OpenCV for Real-Time Performance

OpenCV is widely adopted in surveillance projects for its versatility in handling video processing tasks and its compatibility with various deep learning frameworks. By integrating YOLO with OpenCV, research has shown that real-time performance can be significantly enhanced. For example, a study by Reddy et al. (2019) on automated toll gate systems demonstrated that the YOLO-OpenCV combination could effectively detect and log vehicle details in real time, achieving minimal latency even on mid-range hardware.

3.2 Streamlit for User-Friendly Interfaces

Streamlit has emerged as a popular tool for developing quick, interactive, and user-friendly dashboards in machine learning projects. For example, Kim et al. (2021) successfully deployed a Streamlit-based interface for real-time monitoring of traffic in smart cities. The study highlights Streamlit's ease of use for developing interfaces that allow administrators to view logs, receive alerts, and generate reports without extensive technical knowledge. This aligns well with the requirements of gated communities, where security personnel can monitor vehicle activity without specialized training.

4. Gaps and Challenges in Existing Literature

While significant progress has been made in vehicle detection and tracking, several challenges remain. Existing systems often struggle with:

Occlusion and Crowded Scenes： Detecting vehicles in densely populated areas where objects partially overlap is still a challenge, though techniques like DeepSORT help address this to some extent (Zhu et al., 2020).

Environmental Conditions: Weather conditions like rain, fog, and low lighting impact the accuracy of vehicle detection, a challenge that most gated community solutions need to overcome. Current research suggests that multi-sensor data fusion could improve accuracy under adverse conditions.

Scalability and Data Privacy: Scaling the system for larger communities or multi-gate setups remains challenging, especially when data privacy and security concerns are paramount. Studies by Gupta et al. (2022) suggest cloud deployment with end-to-end encryption to address scalability and privacy, though this approach requires further testing in residential applications.

5. Summary and Relevance to the Project

The literature supports the use of YOLO for vehicle detection due to its high accuracy and real-time capabilities, making it suitable for gated community monitoring. The combination of YOLO with DeepSORT for tracking, OpenCV for efficient processing, and Streamlit for an accessible interface aligns with the needs of this project, as validated by numerous studies. Integrating these technologies provides a scalable, real-time solution that enhances security, optimizes traffic flow, and supports decision-making with detailed data analytics.

This project builds upon these findings, leveraging the strengths of deep learning-based object detection and tracking while addressing challenges such as occlusion and interface accessibility through a tailored Streamlit dashboard. By implementing insights from prior research and using YOLO and Streamlit as foundational tools, the project aims to provide a holistic, secure, and efficient vehicle monitoring system for gated communities.

CHAPTER–3
SYSTEM ANALYSIS

3.1Existing System

In many gated communities, vehicle monitoring is typically achieved through various conventional systems, each with its own advantages and limitations. The most basic form of vehicle monitoring is manual surveillance, where security personnel manually track vehicle entries and exits. This system requires constant human oversight, making it prone to human error and subjectivity. Although it is low-cost and suitable for smaller communities, it lacks efficiency and scalability, particularly for communities with high traffic. In the event of a security threat, manual systems may delay response times, as they rely heavily on the vigilance of security staff.

Another commonly used solution is the CCTV-based monitoring system, where cameras are installed at strategic points within the community, mainly at entry and exit gates. This setup allows for basic record-keeping through video footage, which can be reviewed if an incident occurs. However, CCTV systems still require human monitoring to detect suspicious activity in real time, leading to potential oversight and delayed response. While CCTV provides some level of security deterrence, it does not provide data analytics, vehicle classification, or tracking capabilities. Additionally, video footage review is often time-intensive and may not provide the immediate insights needed for proactive security management.

In recent years, some communities have adopted Automatic Number Plate Recognition (ANPR) systems with limited AI capabilities to improve security. ANPR systems combine license plate recognition with basic AI algorithms to identify vehicle types and colors, providing some level of tracking and classification. Although more advanced than traditional LPR, ANPR systems still focus primarily on entry and exit points, lacking the capability to monitor vehicle movement patterns within the community itself. They are also costly and may require additional hardware and regular updates, making scalability an issue for larger gated communities or those with multiple gates.

3.2Proposed System

The proposed system for monitoring vehicle movement in a gated community combines advanced deep learning algorithms with real-time data processing to enhance security and operational efficiency. By leveraging the YOLO (You Only Look Once) object detection algorithm, the system can accurately and quickly detect vehicles entering and exiting the community. YOLO's real-time detection capabilities allow for instant identification of vehicle types, such as cars, bikes, and trucks, as well as detailed tracking of their movement within the premises. The use of deep learning ensures high accuracy, even in challenging conditions like varying lighting or partial obstructions. Vehicle movement is tracked using the DeepSORT (Simple Online and Realtime Tracking) algorithm, which assigns unique identifiers to vehicles, enabling continuous tracking across multiple camera views.

For the frontend, the system integrates a user-friendly interface built using Streamlit, which provides real-time visualization and analytics of vehicle movements. Security personnel or community administrators can view entry/exit logs, receive alerts about unusual behavior, and monitor vehicle patterns through a comprehensive dashboard. This interface allows for quick decision-making and enhances the system's operational efficiency by providing actionable insights into peak traffic times, security threats, and overall community traffic management. The proposed system is scalable, capable of being adapted for different community sizes, and can be integrated with existing security measures for a more holistic approach to community safety.

The system uses YOLO for vehicle detection and DeepSORT for tracking, offering efficient and real-time monitoring. By utilizing Streamlit, the system provides an intuitive interface for security teams to visualize and analyze data.

This deep learning-based solution offers accurate vehicle categorization, automatic tracking, and real-time alerts.

It can efficiently monitor large communities with multiple entry points.The system's scalability and adaptability make it ideal for diverse gated communities, ensuring enhanced security and operational oversight with minimal human intervention.

CHAPTER 4

SYSTEM REQUIREMENTS & SPECIFICATIONS

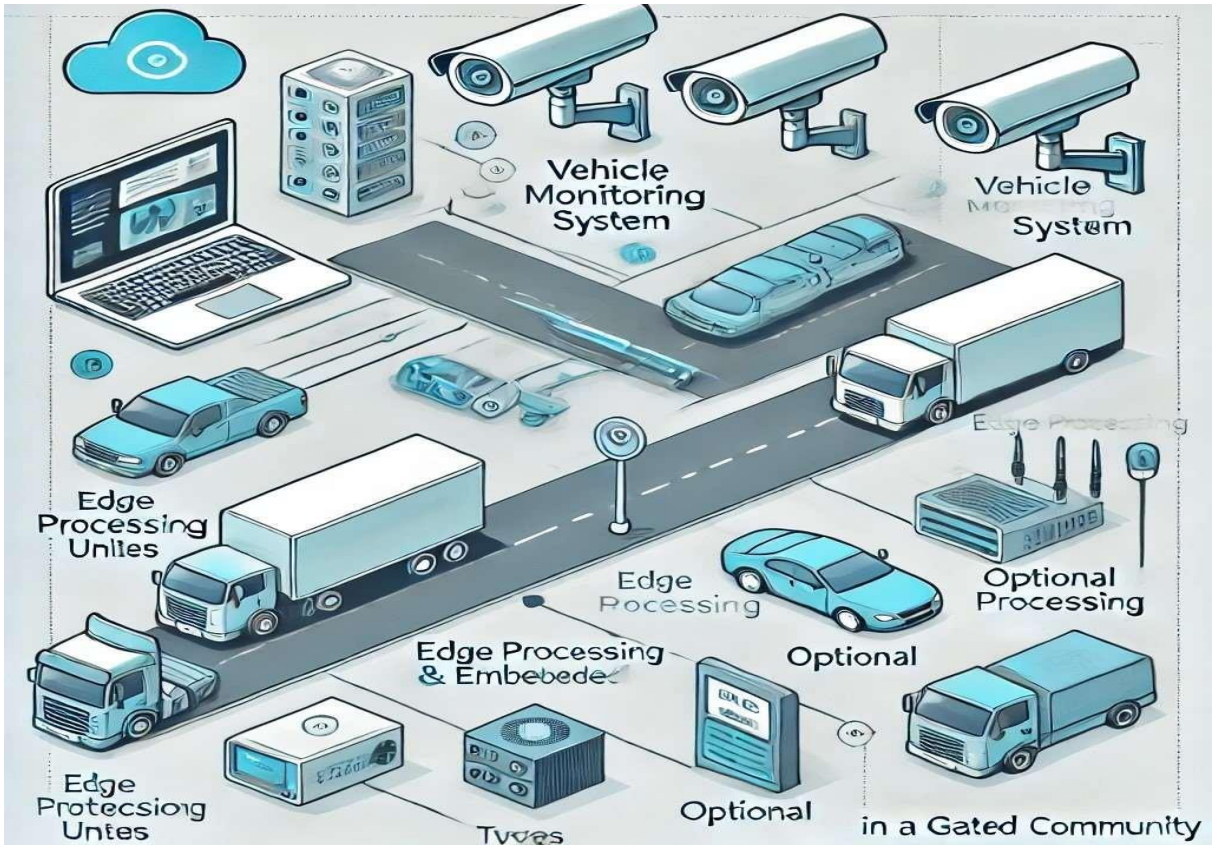
4.1 Database

For this project, PostgreSQL is an ideal database choice due to its ability to handle structured data efficiently and support complex queries. The database will consist of key tables such as vehicle_ counts, which tracks the number of vehicles entering and exiting the gated community by vehicle type, and vehicle_entry_exit_logs, which records detailed logs of each vehicle's entry and exit events, including timestamps. PostgreSQL's ACID compliance ensures that transactions, such as vehicle count updates and logging of vehicle movements, are completed reliably, preventing data inconsistencies and ensuring accurate real-time reporting.

Additionally, PostgreSQL's scalability and indexing capabilities make it well-suited to handle large amounts of data generated by the vehicle detection system. The database will allow for efficient querying of vehicle counts and movement history, enabling the system to provide real-time vehicle statistics for security monitoring and traffic management. With its robust data handling, PostgreSQL ensures that the system can scale as needed, providing reliable data for tracking vehicle movements and generating insightful reports for the community

4.2 Design

4.2.1 System Architecture



1.Input Source-Video Capture:

- **Camera Network:** Installed at the entrance of the gated community,continuously capturing video footage of incoming and outgoing vehicles.
- **Components:** High-resolution IP cameras with night vision for 24/7 monitoring.
- **Data Format:** Real-time video streams.

2.Edge device/Local Processing:

- **Frame Detection:** Breaks video into frames for vehicle detection.
- **Object Detection Algorithm:** Applies a Deep Learning or computer vision model(e.g.,YOLO,SSD) to identify vehicles such as cars,bikes and buses.

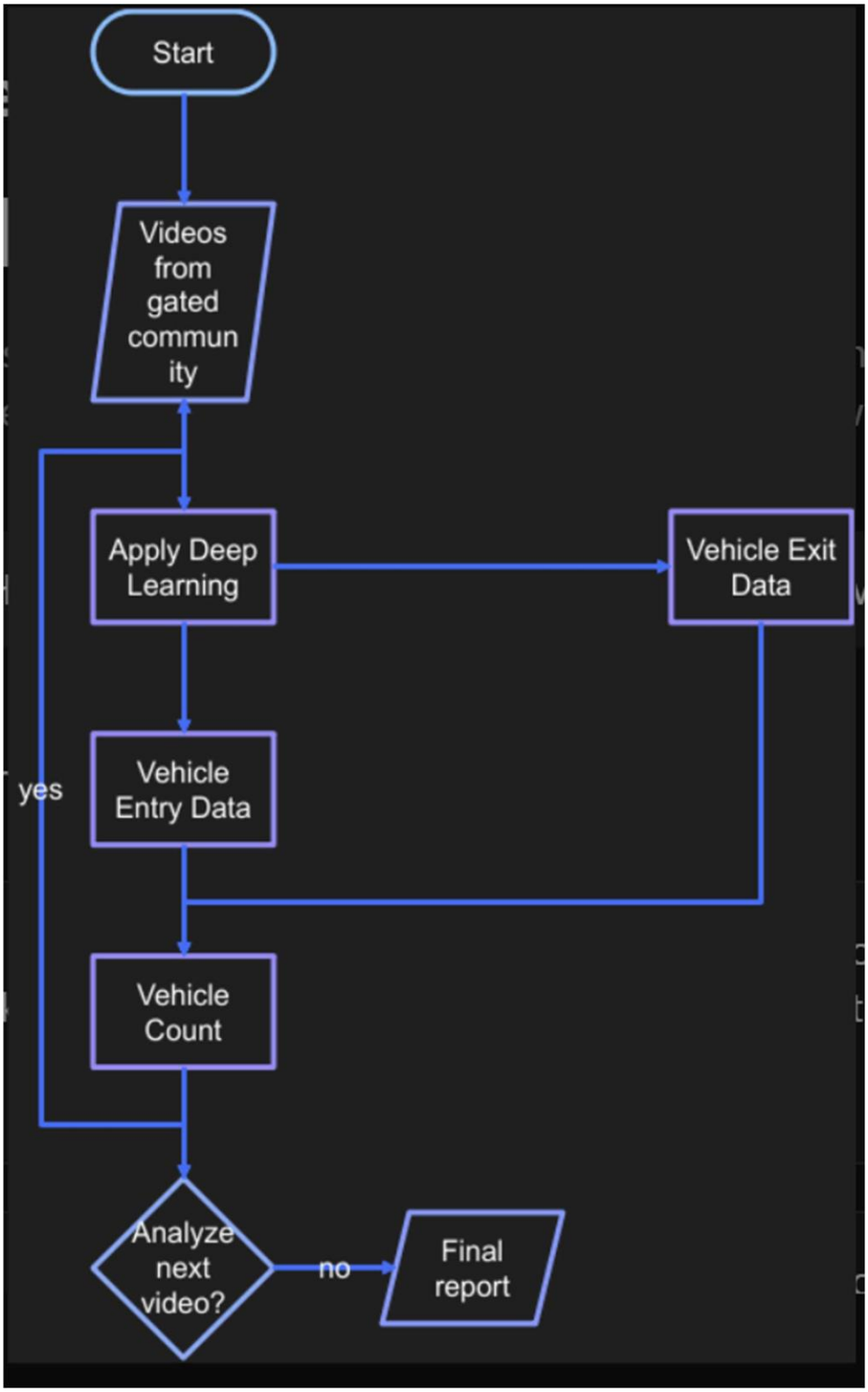
3.Vehicle Classification:

- **Vehicle Detection Model:** Identifies different types of vehicles from the video stream.
- **Object Tracking:** Tracks the movement of vehicles across frames to count them accurately without duplication.

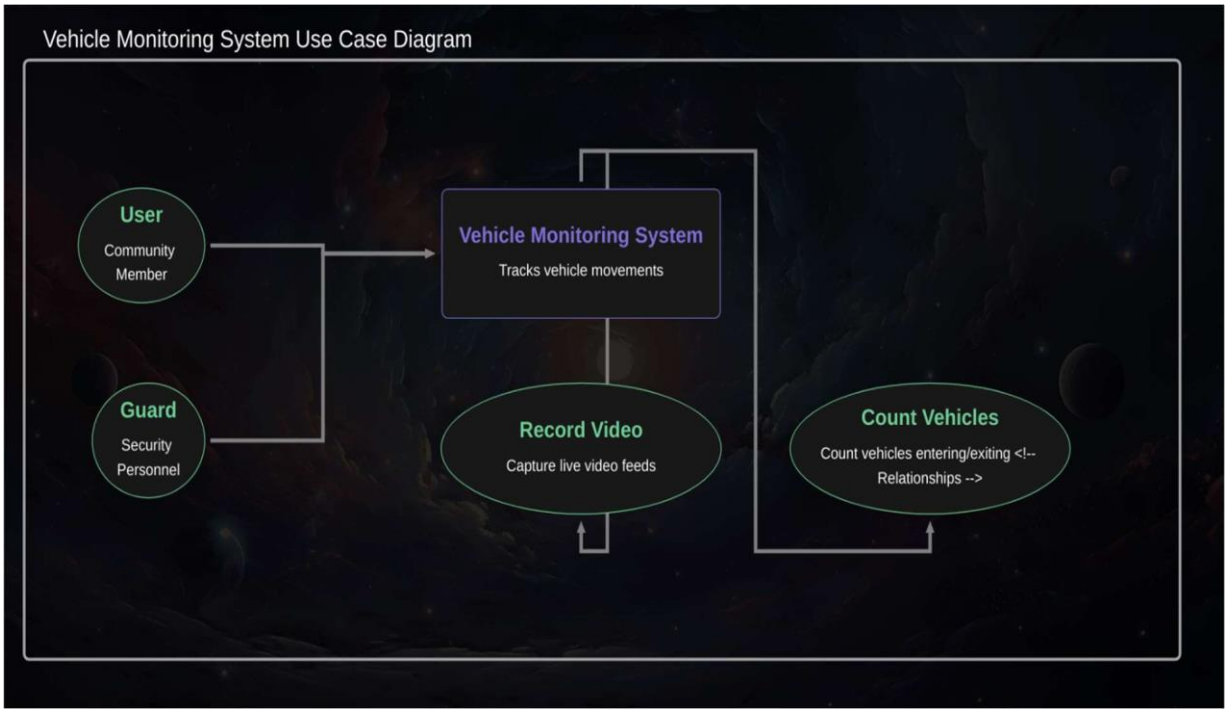
4. Data Aggregation and Counting:

- **Vehicle Entry/Exit Counting:** Records the number of vehicles entering and exiting the gated community.

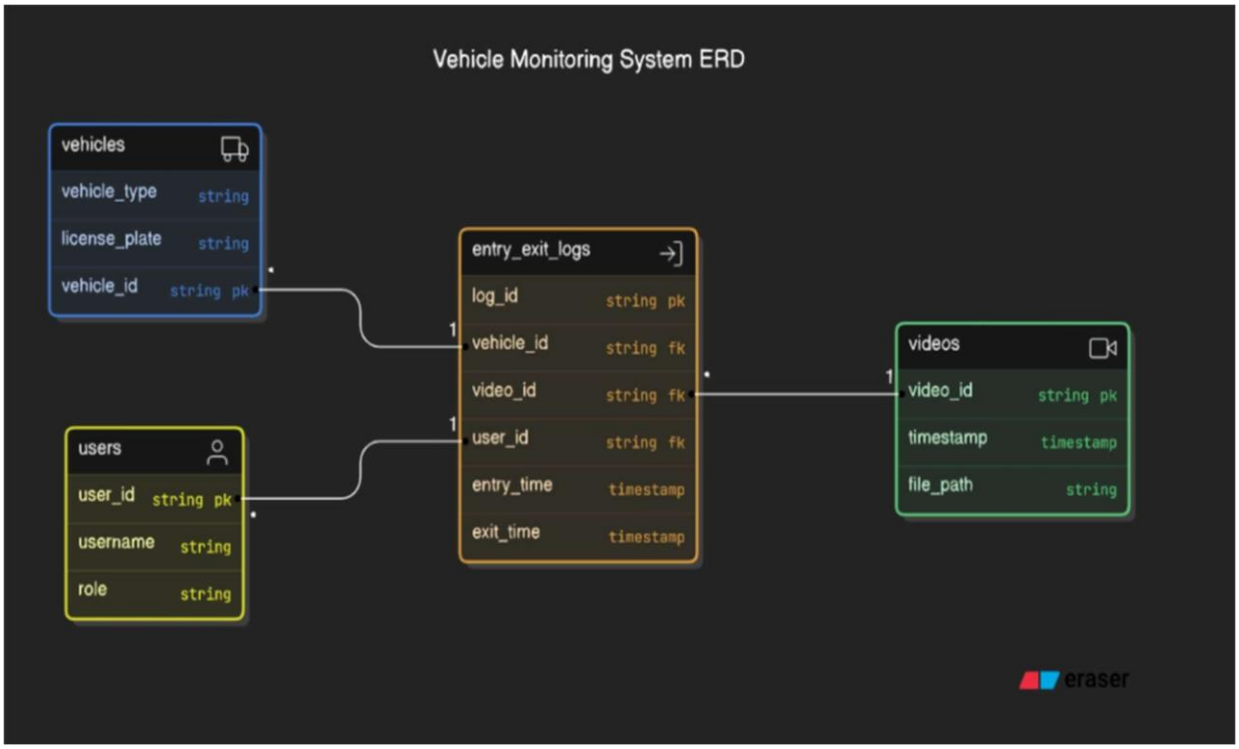
4.2.2 Data Flow Diagram



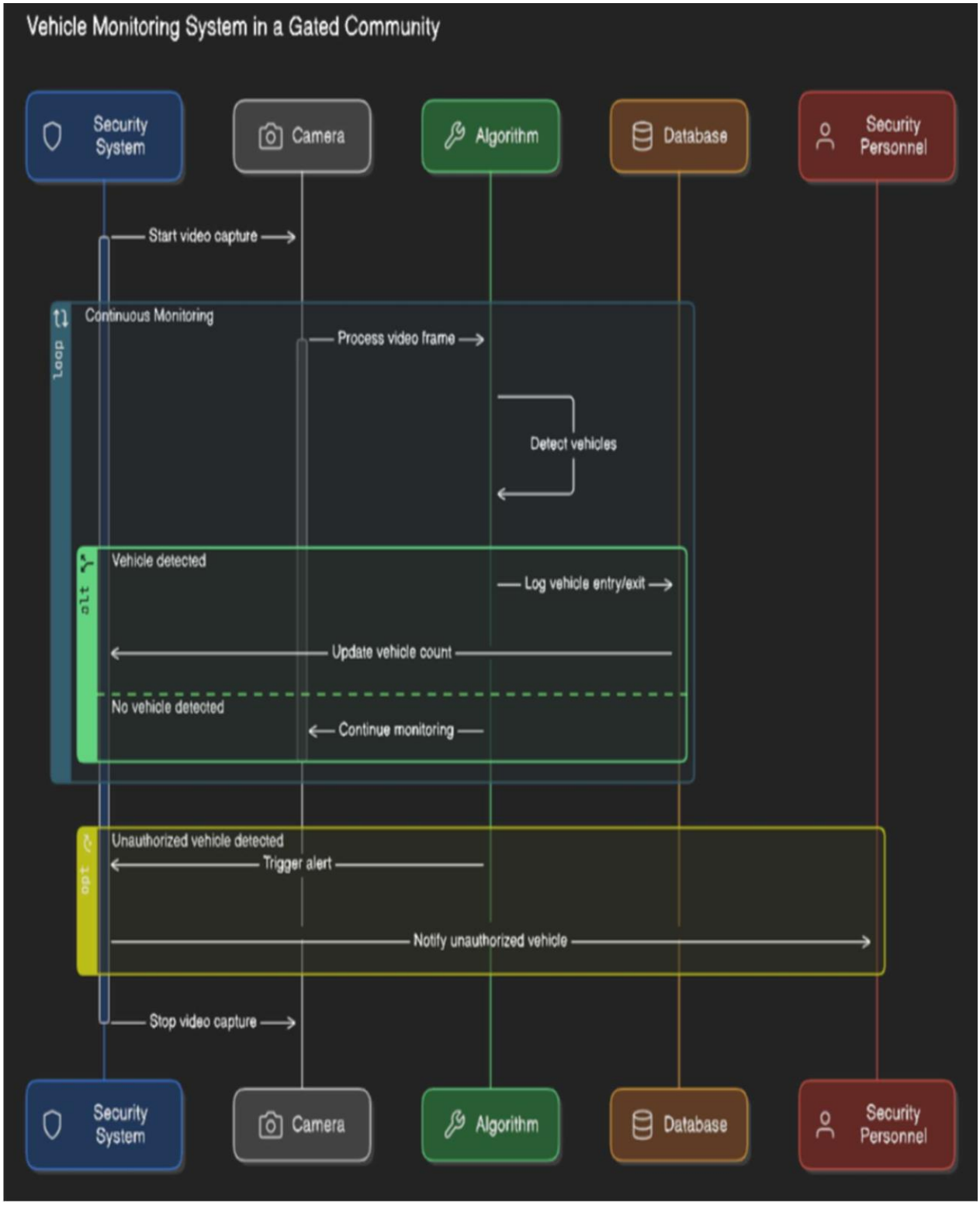
4.2.3 Use Case Diagram



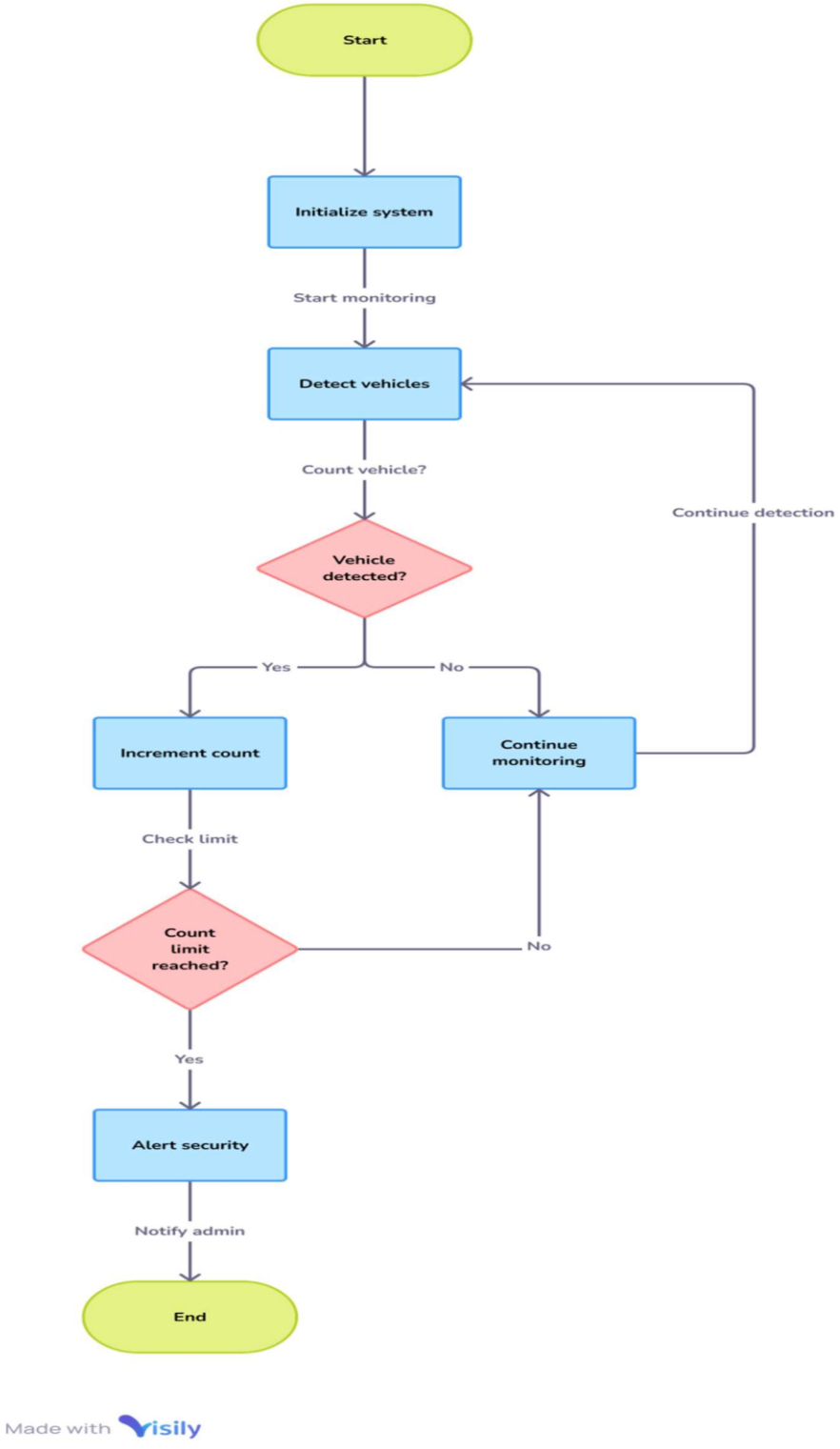
4.2.4 Class Diagram



4.2.5 Sequence Diagram



4.2.6 Activity Diagram



4.3 Modules

4.3.1 Modules Description

1.Video Input and Preprocessing Module: This module captures high-definition video streams from cameras at entry and exit points within the gated community, preprocesses frames for clarity, and converts them into frames for further processing. Tools: OpenCV, image preprocessing libraries.

2.Vehicle Detection and Classification Module: Using the YOLO object detection algorithm, this module identifies and classifies vehicles in each frame, categorizing detected objects (e.g., car, truck, motorcycle) and passing data to the tracking module. Tools: YOLO, TensorFlow or PyTorch.

3.Vehicle Tracking and Counting Module: This module tracks vehicles across frames, assigns unique identifiers, monitors vehicle movement through entry and exit points, and counts each vehicle type based on these events. Tools: YOLO-based tracking, OpenCV.

4.Data Storage Module: This module stores real-time data on vehicle counts, types, and timestamps of entry and exit events in PostgreSQL, ensuring efficient data retrieval and updates for new entries. Tools: PostgreSQL.

5.Real-time Monitoring and Visualization Module: It displays vehicle movements, counts, and logs in real-time on a user-friendly dashboard, updates counts by type, and provides insights into historical movement patterns. Tools: Streamlit, Plotly.

6.Reporting and Analytics Module: Generates periodic reports and analyzes vehicle movement data for security and management, identifying patterns, peak times, and trends for future planning. Tools: SQL queries, Pandas.

7.Security and Access Control Module: Ensures secure access to the monitoring system by authenticating users, authorizing access, and logging user interactions to maintain secure data access. Tools: Streamlit or Flask authentication, database access control.

4.4 System Requirements

4.4.1 Hardware Requirements

1. Camera Setup

- **HD Surveillance Cameras:** High-definition IP cameras for capturing clear images of vehicles at entry and exit points. Suggested resolution is 1080p or higher.
- **Night Vision Capability:** Infrared or low-light cameras to ensure quality footage in dark conditions.
- **Wide Field of View:** Cameras with at least a 90° field of view to cover broad entry and exit zones.

2. Processing Hardware

- **High-Performance Computer (Edge Server):** A local server or powerful computer is required for real-time video processing and object detection tasks.
- **CPU:** Multi-core processor (e.g., Intel i7 or higher).
- **GPU:** Dedicated GPU for running YOLO-based object detection models (e.g., NVIDIA GTX 1080 or higher, or NVIDIA Jetson devices for smaller installations).
- **RAM:** At least 16GB, ideally 32GB or more, to handle multiple high-resolution video streams.
- **Storage:** SSD (512GB or higher) for fast read/write speeds and handling large video data files, plus additional HDD storage for long-term data (1TB or more).

3.Display and Monitoring Equipment

- **Monitor for Real-time Monitoring:** At least a 1080p monitor (24 inches or larger) for live streaming and monitoring dashboard.

4.Access Control and Security

- **Access Control Devices:** Card readers or RFID devices at entry/exit points for security personnel or authorized individuals.
- **Secure Authentication Device:** Hardware security modules (HSM) or multi-factor authentication tokens for access control and security.

4.4.2 Software Requirements

1. Video Processing and Preprocessing

- **OpenCV:** For capturing, preprocessing, and handling video frames in real time.
- **Image Preprocessing Libraries:** Libraries such as Pillow or scikit-image for image enhancement (e.g., noise reduction, sharpening) to improve vehicle detection accuracy.

2.Vehicle Detection and Classification

- **YOLO (You Only Look Once):** Pre-trained YOLO models (e.g., YOLOv4, YOLOv5, or YOLOv8) for object detection and classification of vehicles in video frames.

  Deep Learning Frameworks:

  - **TensorFlow or PyTorch:** Required to load, run, and manage YOLO models and potentially fine-tune them on specific vehicle classes.
  - **CUDA and cuDNN (optional):** GPU acceleration libraries from NVIDIA for speeding up model inference on compatible hardware.

3. Vehicle Tracking and Counting

- **DeepSORT or ByteTrack:** Algorithms for multi-object tracking that work with YOLO outputs to assign unique IDs to detected vehicles and track them across frames.
- **OpenCV (Extended):** For additional video frame processing and vehicle counting logic.

4. Data Storage and Management

- **PostgreSQL:** A relational database for storing vehicle data, including types, entry and exit timestamps, and counts. It supports complex queries and scalability.
- **SQLAlchemy:** For connecting Python applications to PostgreSQL, handling database interactions programmatically.
- **Data Backup Solutions:** Database backup and recovery tools, possibly through PostgreSQL's built-in backup options or third-party solutions.

5. Real-time Monitoring and Visualization

- **Streamlit:** A user-friendly Python-based web framework to create a real-time monitoring dashboard, allowing easy data presentation and interactivity.
- **Plotly or Matplotlib:** For visualizing vehicle counts, movement patterns, and generating graphs for the dashboard.

6. Reporting and Analytics

- **Pandas:** For data manipulation, analysis, and report generation based on historical vehicle movement and traffic patterns.
- **Jupyter Notebook :** For exploratory data analysis and generating reports.
- **SQL Queries:** Predefined SQL queries for aggregating and extracting data from PostgreSQL, supporting analytics and reporting functions.

7. Security and Access Control

- **Streamlit Authentication or Flask Authentication:** For user authentication and role-based access to the monitoring system, limiting who can view and manipulate data.
- **Database Access Control:** Configured in PostgreSQL to set permissions and ensure secure access to data logs and historical records.

4.5 Testing

Testing is essential to ensure that the Vehicle Movement Monitoring System functions correctly, providing accurate detection, tracking, and data logging for vehicles within the gated community. The testing phase is divided into three primary levels: **Unit Testing**, **Integration Testing**, and **System Testing**.

4.5.1 Unit Testing

- **Objective**: To verify the functionality of individual components and methods within the system.
- **Focus Areas**:
    - **Video Frame Preprocessing**: Test each function involved in capturing and converting video frames for processing, ensuring correct resolution, clarity, and frame rate adjustments.
    - **YOLO Vehicle Detection**: Verify the YOLO model's ability to detect vehicles in each frame accurately, and validate that the classification logic works across various vehicle types.
    - **Vehicle Tracking**: Test tracking functions to ensure each vehicle receives a unique identifier and is accurately tracked across frames without errors.
    - **Database Operations**: Ensure that database queries (inserts, updates, deletions) work correctly in PostgreSQL, specifically for logging entry/exit events and updating vehicle counts.
- **Expected Outcome**: Each function performs as expected, handling its specific task accurately, with all edge cases considered.

4.5.2 Integration Testing

- **Objective**: To test the interaction between multiple modules to ensure they work together as a cohesive system.
- **Focus Areas**:
    - **Vehicle Detection and Tracking**: Validate that the detection and tracking modules work together smoothly, accurately detecting and assigning unique IDs to vehicles and maintaining their continuity across frames.
    - **Data Flow from Detection to Database**: Test the flow of data from the detection/tracking modules to the database, ensuring that vehicle counts and logs are recorded accurately in real-time.
    - **Database and Frontend Interaction**: Ensure that data stored in PostgreSQL (vehicle counts, entry/exit logs) is correctly retrieved and displayed in the Streamlit dashboard, reflecting real-time updates.
- **Expected Outcome**: Modules interact seamlessly without data loss or inaccuracies,

maintaining consistent information flow between components.

4.5.3 System Testing

- **Objective**: To validate the entire system's functionality under real-world conditions, ensuring that the complete system meets the specified requirements.
- **Focus Areas**:
    - **End-to-End Vehicle Monitoring**: Verify the system's ability to capture video, detect and track vehicles, log entry/exit events, and display accurate counts on the dashboard.
    - **Performance under Load**: Test the system’s response and accuracy under high traffic scenarios, where multiple vehicles are entering and exiting simultaneously, to ensure stability and speed.
    - **Data Accuracy and Real-Time Display**: Ensure the Streamlit frontend accurately reflects vehicle counts and logs in real time, providing users with a reliable monitoring experience.
    - **Security Testing**: Evaluate access control mechanisms to ensure only authorized users can access and modify data, ensuring data protection and user privacy.
- **Expected Outcome**: The complete system performs as required, accurately monitoring, logging, and displaying vehicle movement with minimal latency and high reliability in diverse operational conditions.

CHAPTER 5

SOURCE CODE

```
# YOLOv5  by Ultralytics, GPL-3.0 license

"""YOLO-specific modules Usage: $ python models/yolo.py --cfg yolov5s.yaml """

import argparse

import contextlib

import os

import platform

import sys
```

```
from copy import deepcopy

from pathlib import Path

FILE = Path(__file__).resolve()

ROOT = FILE.parents[1] # YOLOv5 root directory

if str(ROOT) not in sys.path:

            sys.path.append(str(ROOT)) # add ROOT to PATH

            if platform.system() != 'Windows':

          ROOT = Path(os.path.relpath(ROOT, Path.cwd()))  # relative



from models.common import *

from models.experimental import *

from utils.autoanchor import check_anchor_order

from utils.general import LOGGER, check_version, check_yaml, make_divisible, print_args

from utils.plots import feature_visualization

from utils.torch_utils import (fuse_conv_and_bn, initialize_weights, model_info, profile, scale_img, time_sync)

try import thop # for FLOPs computation

except Import Error:

 thop = None

class Detect(nn.Module):# YOLOv5 Detect head for detection models

stride = None  # strides computed during build

dynamic = False # force grid reconstruction

 export = False # export model
```

```
def __init_(self, nc=80, anchors=(), ch=(), inplace=True): # detection layer
            super()._init_()
            self.nc = nc  # number of classes
           self.no = nc + 5 # number of outputs per anchor
         self.nl = len(anchors)  # number of detection layers
  self.na = len(anchors[0]) // 2  # number of anchors
self.grid = [torch.empty(0) for _ in range(self.nl)]  # init grid
self.anchor_grid = [torch.empty(0) for _ in range(self.nl)] # init anchor grid
self.register_buffer('anchors', torch.tensor(anchors).float().view(self.nl, -1, 2)) # shape(nl,na,2)
self.m = nn.ModuleList(nn.Conv2d(x, self.no * self.na, 1) for x in ch) # output conv
self.inplace = inplace # use inplace ops (e.g. slice assignment)
def forward(self, x):
z = [] # inference output
 for i in range(self.nl):
x[i] = self.m[i](x[i])  # conv
 bs, _, ny, nx = x[i].shape  # x(bs,255,20,20) to x(bs,3,20,20,85)
x[i] = x[i].view(bs, self.na, self.no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()
  if not self.training:  # inference
  if self.dynamic or self.grid[i].shape[2:4] != x[i].shape[2:4]
self.grid[i], self.anchor_grid[i] = self._make_grid(nx, ny, i)
if isinstance(self, Segment): # (boxes + masks)
xy, wh, conf, mask = x[i].split((2, 2, self.nc + 1, self.no - self.nc - 5), 4)
```

```
xy = (xy.sigmoid() * 2 + self.grid[i]) * self.stride[i] # xy
wh = (wh.sigmoid() * 2) ** 2 * self.anchor_grid[i] # wh
y = torch.cat((xy, wh, conf.sigmoid(), mask), 4)
else: # Detect (boxes only)
xy, wh, conf = x[i].sigmoid().split((2, 2, self.nc + 1), 4)
xy = (xy * 2 + self.grid[i]) * self.stride[i] # xy
wh = (wh * 2) ** 2 * self.anchor_grid[i]  # wh
 y = torch.cat((xy, wh, conf), 4)
 z.append(y.view(bs, self.na * nx * ny, self.no))
return x if self.training else (torch.cat(z, 1),) if self.export else (torch.cat(z, 1), x)
def _make_grid(self, nx=20, ny=20, i=0, torch_1_10=check_version(torch.__version__, '1.10.0')):
d = self.anchors[i].device
t = self.anchors[i].dtype
 shape = 1, self.na, ny, nx, 2 # grid shape
y, x = torch.arange(ny, device=d, dtype=t), torch.arange(nx, device=d, dtype=t)
   yv, xv = torch.meshgrid(y, x, indexing='ij') if torch_1_10 else torch.meshgrid(y, x) # torch>=0.7 compatibility
grid = torch.stack((xv, yv), 2).expand(shape) - 0.5  # add grid offset, i.e. y = 2.0 * x - 0.5
anchor_grid = (self.anchors[i] * self.stride[i]).view((1, self.na, 1, 1, 2)).expand(shape)
 return grid, anchor_grid
```

Demo.Py:

```
from track import *
import tempfile
```

```
import cv2
import torch
import streamlit as st
import os
def display_footer():
    footer_text = """
     <div style="background-color:#F5F5DC; padding: 10px; border-radius: 5px;">
        <center><h2>Vehicle Monitoring System in Community</h2><br>
        <h3>Done By:</h3><br><h3>Azeed.Sk</h3> <br>
        <h3>Anusha.B </h3><br>
        <h3>Arbaz Ali.Mohammad </h3><br>
        <h3>Ashwanth Reddy.S</h3> <br>
        <h3>Guided By: DR.Ramababu.P Prof. Mruh </h3><br>
        <h3>All Copyrights @ Reserved 2024</h3></center>
         </div>
    """
    st.markdown("<hr>", unsafe_allow_html=True)
            st.markdown(f'<p style="text-align: center; color: black;">{footer_text}</p>', unsafe_allow_html=True)
if __name__== '__main__':
    st.title('vehicle monitoring system in a gated community')
        st.markdown('<h3 style="color: red"> with Yolov5 and Deep Learning </h3', unsafe_allow_html=True)
    # upload video
    video_file_buffer = st.sidebar.file_uploader("Upload a video", type=['mp4', 'mov', 'avi']
    if video_file_buffer:
        st.sidebar.text('Input video')
        st.sidebar.video(video_file_buffer)
```

```
    # save video from streamlit into "videos" folder for future detect
    with open(os.path.join('videos', video_file_buffer.name), 'wb') as f:
        f.write(video_file_buffer.getbuffer())
st.sidebar.markdown('---')
st.sidebar.title('Settings')
# custom class
custom_class = st.sidebar.checkbox('Custom classes')
assigned_class_id = [0, 1, 2, 3]
names = ['car', 'motorcycle', 'truck', 'bus']
if custom_class:
    assigned_class_id = []
    assigned_class = st.sidebar.multiselect('Select custom classes', list(names))
    for each in assigned_class:
        assigned_class_id.append(names.index(each))
# st.write(assigned_class_id)
# setting hyperparameter
confidence = st.sidebar.slider('Confidence', min_value=0.0, max_value=1.0, value=0.5)
line = st.sidebar.number_input('Line position', min_value=0.0, max_value=1.0, value=0.6, step=0.1)
st.sidebar.markdown('---')
status = st.empty()
stframe = st.empty()
if video_file_buffer is None:
status.markdown('<font size= "4"> **Status:** Waiting for input </font>', unsafe_allow_html=True)
else:
    status.markdown('<font size= "4"> **Status:** Ready </font>', unsafe_allow_html=True)
car, bus, truck, motor = st.columns(4)
with car:
```

```
    st.markdown('**Car**')
    car_text = st.markdown('__')
with bus:
    st.markdown('**Bus**')
    bus_text = st.markdown('__')
with truck:
    st.markdown('**Truck**')
    truck_text = st.markdown('__')
with motor:
    st.markdown('**Motorcycle**')
    motor_text = st.markdown('__')
fps, _, _, _ = st.columns(4)
with fps:
    st.markdown('**FPS**')
    fps_text = st.markdown('_')
track_button = st.sidebar.button('START')
# reset_button = st.sidebar.button('RESET ID')
if track_button:
    # reset ID and count from 0
    reset()
    opt = parse_opt()
    opt.conf_thres = confidence
    opt.source = f'videos/{video_file_buffer.name}'
    status.markdown('<font size= "4"> **Status:** Running... </font>', unsafe_allow_html=True)
    with torch.no_grad():
        detect(opt, stframe, car_text, bus_text, truck_text, motor_text, line, fps_text, assigned_class_id)
    status.markdown('<font size= "4"> **Status:** Finished ! </font>', unsafe_allow_html=True)
```

```
            # end_noti = st.markdown('<center style="color: blue"> FINISH </center>', unsafe_allow_html=True)

 # Display the footer

   display_footer()

   # if reset_button:

      # reset()

   #     st.markdown('<h3 style="color: blue"> Reseted ID </h3>', unsafe_allow_html=True)
```

Track.Py

```
# limit the number of cpus used by high performance libraries

import os

os.environ["OMP_NUM_THREADS"] = "1"

os.environ["OPENBLAS_NUM_THREADS"] = "1"

os.environ["MKL_NUM_THREADS"] = "1"

os.environ["VECLIB_MAXIMUM_THREADS"] = "1"

os.environ["NUMEXPR_NUM_THREADS"] = "1"

import  sys

sys.path.insert(0, './yolov5')

import streamlit as st

import time

import IPython

import argparse

import os

import platform

import shutil

import time

from pathlib import Path

import cv2
```

```
import torch
import torch.backends.cudnn as cudnn
from yolov5.models.experimental import attempt_load
from yolov5.utils.downloads import attempt_download
from yolov5.models.common import DetectMultiBackend
from yolov5.utils.dataloaders import LoadImages, LoadStreams
from yolov5.utils.general import (LOGGER, check_img_size, non_max_suppression, scale_boxes,
 check_imshow, xyxy2xywh, increment_path)
from yolov5.utils.torch_utils import select_device, time_sync
from yolov5.utils.plots import Annotator, colors
from deep_sort.utils.parser import get_config
from deep_sort.deep_sort import DeepSort
FILE = Path(_file_).resolve()
ROOT = FILE.parents[0] # yolov5 deepsort root directory
if str(ROOT) not in sys.path:
    sys.path.append(str(ROOT))  # add ROOT to PATH
ROOT = Path(os.path.relpath(ROOT, Path.cwd())) # relative
# count_car, count_bus, count_truck = 0, 0, 0
data_car = []
data_bus = []
data_truck = []
data_motor = []
already = []
line_pos = 0.6
def detect(opt, stframe, car, bus, truck, motor, line, fps_rate, class_id):
      out, source, yolo_model, deep_sort_model, show_vid, save_vid, save_txt, imgsz, evaluate, half,
project, name, exist_ok= \
```

```
    opt.output, opt.source, opt.yolo_model, opt.deep_sort_model, opt.show_vid, opt.save_vid, \
    opt.save_txt, opt.imgsz, opt.evaluate, opt.half, opt.project, opt.name, opt.exist_ok
  # choose custom class from streamlit
  opt.classes = class_id
  webcam = source == '0' or source.startswith(
    'rtsp') or source.startswith('http') or source.endswith('.txt')
  sum_fps = 0
  line_pos = line
  save_vid = True
  # initialize deepsort
  cfg = get_config()
  cfg.merge_from_file(opt.config_deepsort)
  deepsort = DeepSort(deep_sort_model,
                      max_dist=cfg.DEEPSORT.MAX_DIST,
                      max_iou_distance=cfg.DEEPSORT.MAX_IOU_DISTANCE,
                                  max_age=cfg.DEEPSORT.MAX_AGE, n_init=cfg.DEEPSORT.N_INIT,
nn_budget=cfg.DEEPSORT.NN_BUDGET,
                      use_cuda=True)
  # Initialize
  device = select_device(opt.device)
  half &= device.type != 'cpu' # half precision only supported on CUDA
  # The MOT16 evaluation runs multiple inference streams in parallel, each one writing to
  # its own .txt file. Hence, in that case, the output folder is not restored
  if not evaluate:
    if os.path.exists(out):
      pass
      shutil.rmtree(out) # delete output folder
```

```
    os.makedirs(out) # make new output folder
# Directories
save_dir = increment_path(Path(project) / name, exist_ok=exist_ok) # increment run
save_dir.mkdir(parents=True, exist_ok=True) # make dir
# Load model
device = select_device(device)
model = DetectMultiBackend(yolo_model, device=device, dnn=opt.dnn)
stride, names, pt, jit, _ = model.stride, model.names, model.pt, model.jit, model.onnx
imgsz = check_img_size(imgsz, s=stride)  # check image size
# Half
half &= pt and device.type != 'cpu' # half precision only supported by PyTorch on CUDA
if pt:
    model.model.half() if half else model.model.float()
# Set Dataloader
vid_path, vid_writer = None, None
# Check if environment supports image displays
if show_vid:
    show_vid = check_imshow()
# Dataloader
if webcam:
    show_vid = check_imshow()
    cudnn.benchmark = True # set True to speed up constant image size inference
    dataset = LoadStreams(source, img_size=imgsz, stride=stride, auto=pt and not jit)
    bs = len(dataset)  # batch_size
else:
    dataset = LoadImages(source, img_size=imgsz, stride=stride, auto=pt and not jit)
    bs = 1  # batch_size
```

```
vid_path, vid_writer = [None] * bs, [None] * bs
# Get names and colors
names = model.module.names if hasattr(model, 'module') else model.names
# extract what is in between the last '/' and last '.'
txt_file_name = source.split('/')[-1].split('.')[0]
txt_path = str(Path(save_dir)) + '/' + txt_file_name + '.txt'
if pt and device.type != 'cpu':
    model(torch.zeros(1, 3, *imgsz).to(device).type_as(next(model.model.parameters()))) # warmup
dt, seen = [0.0, 0.0, 0.0, 0.0], 0
for frame_idx, (path, img, im0s, vid_cap, s) in enumerate(dataset):
    t1 = time_sync()
    img = torch.from_numpy(img).to(device)
    img = img.half() if half else img.float() # uint8 to fp16/32
    img /= 255.0 # 0 - 255 to 0.0 - 1.0
    if img.ndimension() == 3:
        img = img.unsqueeze(0)
    t2 = time_sync()
    dt[0] += t2 - t1
    prev_time = time.time()
    # Inference
    visualize = increment_path(save_dir / Path(path).stem, mkdir=True) if opt.visualize else False
    pred = model(img, augment=opt.augment, visualize=visualize)
    t3 = time_sync()
    dt[1] += t3 - t2
    # Apply NMS
      pred = non_max_suppression(pred, opt.conf_thres, opt.iou_thres, opt.classes, opt.agnostic_nms,
max_det=opt.max_det)
```

```
dt[2] += time_sync() - t3
# Process detections
for i, det in enumerate(pred): # detections per image
    seen += 1
    if webcam:  # batch_size >= 1
        p, im0, _ = path[i], im0s[i].copy(), dataset.count
        s += f'{i}: '
    else:
        p, im0, _ = path, im0s.copy(), getattr(dataset, 'frame', 0)
    p = Path(p)  # to Path
    save_path = str(save_dir / p.name) # im.jpg, vid.mp4, ...
    s += '%gx%g ' % img.shape[2:]  # print string
    annotator = Annotator(im0, line_width=2, pil=not ascii)
    w, h = im0.shape[1],im0.shape[0]
    if det is not None and len(det):
        # Rescale boxes from img_size to im0 size
        det[:, :4] = scale_boxes(
            img.shape[2:], det[:, :4], im0.shape).round()
        # Print results
        for c in det[:, -1].unique():
            n = (det[:, -1] == c).sum()  # detections per class
            s += f"{n} {names[int(c)]}{'s' * (n > 1)}, " # add to string
        xywhs = xyxy2xywh(det[:, 0:4])
        confs = det[:, 4]
        clss = det[:, 5]
        # pass detections to deepsort
        t4 = time_sync()
```

```
        outputs = deepsort.update(xywhs.cpu(), confs.cpu(), clss.cpu(), im0)
        t5 = time_sync()
        dt[3] += t5 - t4
        # draw boxes for visualization
        if len(outputs) > 0:
            for j, (output, conf) in enumerate(zip(outputs, confs)):
                bboxes = output[0:4]
                id = output[4]
                cls = output[5]
                #count
                c = int(cls) # integer class
                label = f'{id} {names[c]} {conf:.2f}'
                annotator.box_label(bboxes, label, color=colors(c, True))
                # count_obj(bboxes,w,h,id, names[c], data_car, data_bus, data_truck, data_motor)
                count_obj(bboxes,w,h,id, names[c], line_pos)
                if save_txt:
                    # to MOT format
                    bbox_left = output[0]
                    bbox_top = output[1]
                    bbox_w = output[2] - output[0]
                    bbox_h = output[3] - output[1]
                    # Write MOT compliant results to file
                    with open(txt_path, 'a') as f:
                        f.write(('%g ' * 10 + '\n') % (frame_idx + 1, id, bbox_left, # MOT format
                                                       bbox_top, bbox_w, bbox_h, -1, -1, -1, -1))
LOGGER.info(f'{s}Done. YOLO:({t3 - t2:.3f}s), DeepSort:({t5 - t4:.3f}s)')
        else:
```

```
            deepsort.increment_ages()
            LOGGER.info('No detections')
        # Stream results
        im0 = annotator.result()
        if show_vid:
            # count vehicle
            color = (0,255,0)
            color_car = (0,150,255)
            color_bus = (0,255,0)
            color_truck = (255,0,0)
            color_motor = (255,255,0)
            start_point = (0, int(line_pos*h))
            end_point = (w, int(line_pos*h))
            cv2.line(im0, start_point, end_point, color, thickness=2)
            thickness = 3
            org = (20, 70)
            distance_height = 100
            font = cv2.FONT_HERSHEY_SIMPLEX
            fontScale = 2
                # cv2.putText(im0, 'car: ' + str(len(data_car)), org, font, fontScale, color_car, thickness, cv2.LINE_AA)
                # cv2.putText(im0, 'bus: ' + str(len(data_bus)), (org[0], org[1] + distance_height), font, fontScale, color_bus, thickness, cv2.LINE_AA)
                # cv2.putText(im0, 'truck: ' + str(len(data_truck)), (org[0], org[1] + distance_height*2), font, fontScale, color_truck, thickness, cv2.LINE_AA)
                # cv2.putText(im0, 'motor: ' + str(len(data_motor)), (org[0], org[1] + distance_height*3), font, fontScale, color_motor, thickness, cv2.LINE_AA)
            # cv2.imshow(str(p), im0)
            if cv2.waitKey(1) == ord('q'):  # q to quit
```

```
        raise StopIteration
# Save results (image with detections)
if save_vid:
    if vid_path != save_path: # new video
        vid_path = save_path
        if isinstance(vid_writer, cv2.VideoWriter):
            vid_writer.release()  # release previous video writer
        if vid_cap: # video
            fps = vid_cap.get(cv2.CAP_PROP_FPS)
            w = int(vid_cap.get(cv2.CAP_PROP_FRAME_WIDTH))
            h = int(vid_cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        else: # stream
            fps, w, h = 60, im0.shape[1], im0.shape[0]
        vid_writer = cv2.VideoWriter(save_path, cv2.VideoWriter_fourcc(*'mp4v'), fps, (w, h))
    vid_writer.write(im0)
    # show fps
    curr_time = time.time()
    fps_ = curr_time - prev_time
    fps_ = round(1/round(fps_, 3),1)
    prev_time = curr_time
    sum_fps += fps_
    stframe.image(im0, channels="BGR", use_column_width=True)
    car.markdown(f"<h3> {str(len(data_car))} </h3>", unsafe_allow_html=True)
    bus.write(f"<h3> {str(len(data_bus))} </h3>", unsafe_allow_html=True)
    truck.write(f"<h3> {str(len(data_truck))} </h3>", unsafe_allow_html=True)
    motor.write(f"<h3> {str(len(data_motor))} </h3>", unsafe_allow_html=True)
    fps_rate.markdown(f"<h3> {fps_} </h3>", unsafe_allow_html=True)
```

```
    # Print results
    t = tuple(x / seen * 1E3 for x in dt) # speeds per image
    print("Average FPS", round(1 / (sum(list(t)) / 1000), 1))
     LOGGER.info(f'Speed: %.1fms pre-process, %.1fms inference, %.1fms NMS, %.1fms deep sort update \
        per image at shape {(1, 3, *imgsz)}' % t)
    if save_txt or save_vid:
        print('Results saved to %s' % save_path)
        if platform == 'darwin': # MacOS
            os.system('open ' + save_path)
def count_obj(box, w, h, id, label, line_pos):
    global data_car, data_bus, data_truck, data_motor, already
    center_coordinates = (int(box[0]+(box[2]-box[0])/2) , int(box[1]+(box[3]-box[1])/2))
    # classify one time per id
    if center_coordinates[1] > (h*line_pos):
        if id not in already:
   already.append(id)
            if label == 'car' and id not in data_car:
                data_car.append(id)
            elif label == 'bus' and id not in data_bus:
                data_bus.append(id)
            elif label == 'truck' and id not in data_truck:
                data_truck.append(id)
            elif label == 'motorcycle' and id not in data_motor:
                data_motor.append(id)
# reset id in data
def reset():
```

```
    global data_car, data_bus, data_truck, data_motor, already

    data_car = []

    data_bus = []

    data_truck = []

    data_motor = []

    already = []

def parse_opt():

    parser = argparse.ArgumentParser()

        parser.add_argument('--yolo_model', nargs='+', type=str, default='best_new.pt', help='model.pt path(s)')

    parser.add_argument('--deep_sort_model', type=str, default='osnet_x0_25')

    parser.add_argument('--source', type=str, default='videos/motor.mp4', help='source') # file/folder, 0 for webcam

      parser.add_argument('--output', type=str, default='inference/output', help='output folder') # output folder

     parser.add_argument('--imgsz', '--img', '--img-size', nargs='+', type=int, default=[480], help='inference size h,w')

    parser.add_argument('--conf-thres', type=float, default=0.5, help='object confidence threshold')

    parser.add_argument('--iou-thres', type=float, default=0.5, help='IOU threshold for NMS')

    parser.add_argument('--fourcc', type=str, default='mp4v', help='output video codec (verify ffmpeg support)')

    parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')

    parser.add_argument('--show-vid', action='store_false', help='display tracking video results')

    parser.add_argument('--save-vid', action='store_true', help='save video tracking results')

    parser.add_argument('--save-txt', action='store_true', help='save MOT compliant results to *.txt')

    # class 0 is person, 1 is bycicle, 2 is car... 79 is oven

    parser.add_argument('--classes', nargs='+', type=int, help='filter by class: --class 0, or --class 16 17')

    parser.add_argument('--agnostic-nms', action='store_true', help='class-agnostic NMS')

    parser.add_argument('--augment', action='store_true', help='augmented inference')
```

```
    parser.add_argument('--evaluate', action='store_true', help='augmented inference')
    parser.add_argument("--config_deepsort", type=str, default="deep_sort/configs/deep_sort.yaml")
    parser.add_argument("--half", action="store_true", help="use FP16 half-precision inference")
    parser.add_argument('--visualize', action='store_true', help='visualize features')
    parser.add_argument('--max-det', type=int, default=1000, help='maximum detection per image')
    parser.add_argument('--dnn', action='store_true', help='use OpenCV DNN for ONNX inference')
    parser.add_argument('--project', default=ROOT / 'runs/track', help='save results to project/name')
    parser.add_argument('--name', default='exp', help='save results to project/name')
    parser.add_argument('--exist-ok', action='store_true', help='existing project/name ok, do not increment')
    opt = parser.parse_args()
    opt.img *= 2 if len(opt.imgsz) == 1 else 1 # expand
    return opt
if __name__ == '_main_':
    opt = parse_opt()
    with torch.no_grad():
        detect(opt)
```

CHAPTER 6

EXPERIMENTAL RESULTS

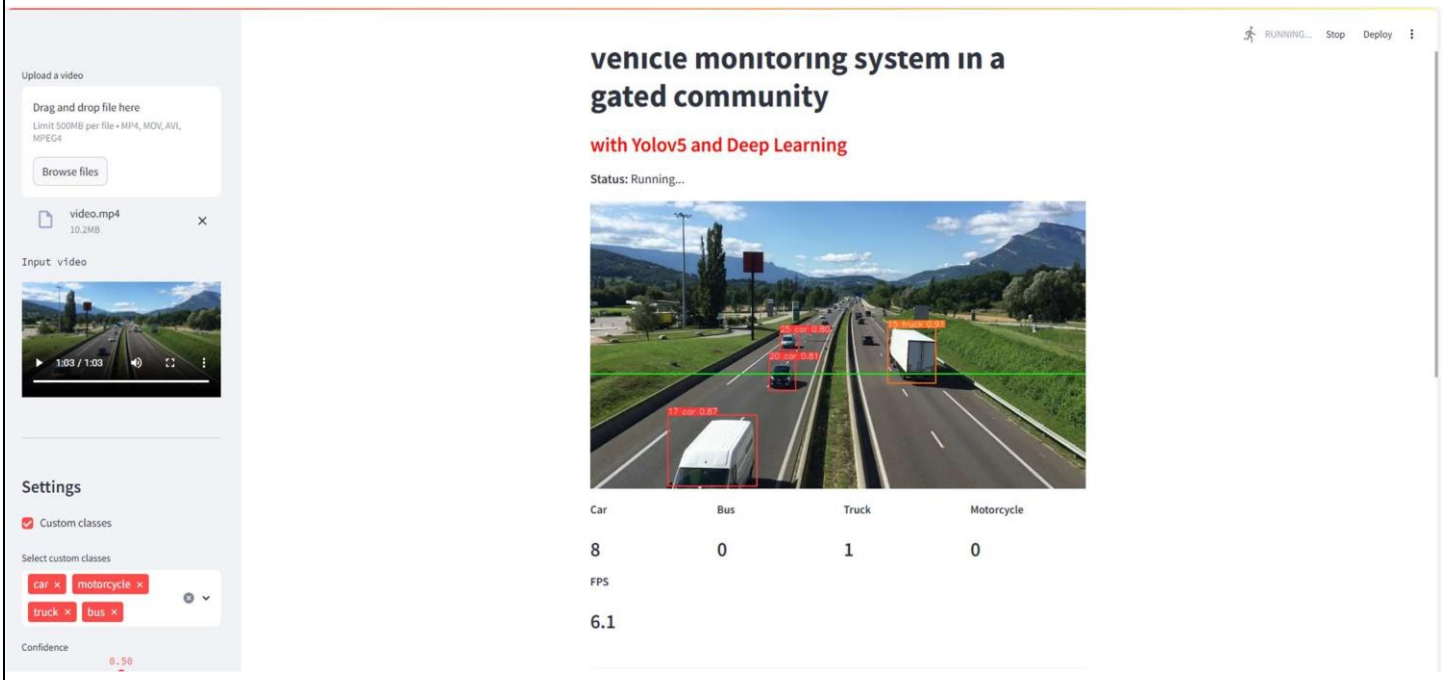


Figure:6.1 Dashboard

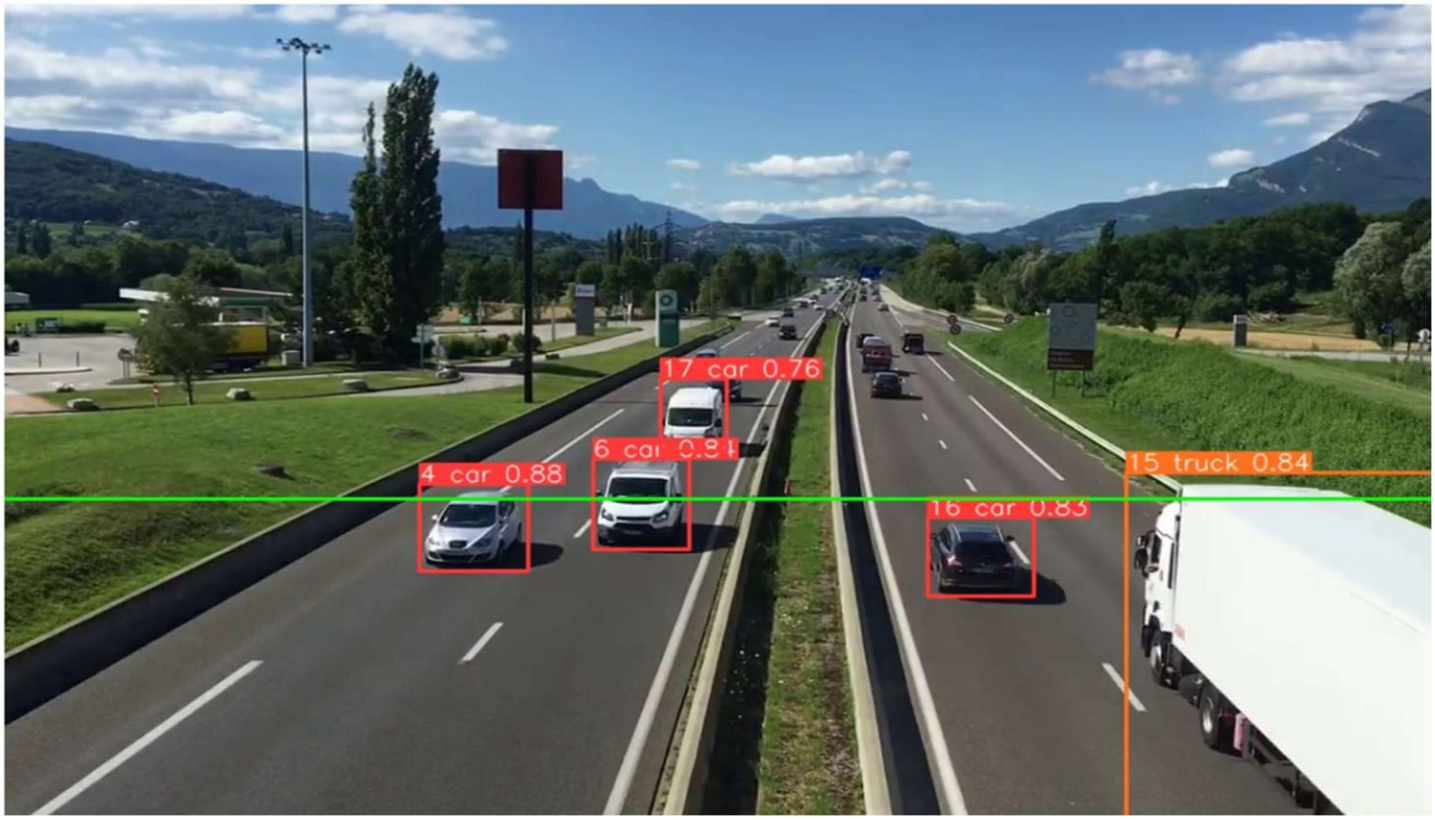


Figure: 6.2 detection of Vehicles

CHAPTER 7

CONCLUSION & FUTURE ENHANCEMENT

7.1 Conclusion

This deep learning-based vehicle monitoring system significantly improves security in gated communities by providing automated, real-time tracking and analysis of vehicle movements. Leveraging the YOLO (You Only Look Once) model for object detection enhances the efficiency of surveillance by accurately identifying and tracking vehicles, which minimizes the need for manual monitoring and ensures prompt detection of suspicious activities. The successful deployment of this system contributes to a safer and more secure environment, showcasing the valuable application of advanced technology in residential security.

7.2 Future Enhancement

Here are five focused future enhancements for your vehicle movement monitoring system:

- **Multi-camera Integration**
  Expanding the system to include multiple cameras at different entry and exit points or other critical areas would improve the accuracy of vehicle tracking and provide a more comprehensive view of vehicle movements.

- **Real-time Alert System with Notification Integration**
  Adding a real-time alert system that sends notifications via SMS, email, or a dedicated app for any suspicious activity would allow for quick responses from security personnel or residents.

- **Vehicle Type and License Plate Recognition**
  Integrating vehicle type classification (e.g., car, motorcycle, truck) and license plate recognition (ANPR) would allow automated access control for authorized vehicles and flag unauthorized ones more effectively.

- **Predictive Analytics for Traffic Patterns**
  Analyzing historical data to predict peak traffic times could help optimize security resource allocation, such as increasing personnel during high-traffic periods.

- **Night Vision and Low-Light Detection**
  Incorporating infrared cameras or algorithms for low-light detection would enhance vehicle tracking and monitoring effectiveness at night or in poor lighting conditions.

These targeted improvements would enhance the system's overall accuracy, responsiveness, and usability in a gated community.

REFERENCES

1. S. Ren, K. He, R. Girshick, and C. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, June 2017.

2. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779-788, 2016.

3. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Advances in Neural Information Processing Systems (NeurIPS), pp. 1097-1105, 2012.

4. G. Bradski and A. Kaehler, Learning OpenCV: Computer Vision with the OpenCV Library, O'Reilly Media, 2008.

5. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. "SSD: Single Shot Multibox Detector," *European Conference on Computer Vision (ECCV)*, pp. 21-37, 2016.

6. Ge, Z., Liu, S., Wang, F., Li, Z., & Sun, J. "YOLOX: Exceeding YOLO Series in 2021," *arXiv preprint arXiv:2107.08430*, 2021.

7. Zhu, X., Lyu, S., Wang, X., & Zhao, Q. "TMM: Deep Structured Models for Multi-Object Tracking," *IEEE Transactions on Multimedia*, vol. 19, no. 4, pp. 750-762, 2017.

8. Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... & Zhang, X. "End to End Learning for Self-Driving Cars," *arXiv preprint arXiv:1604.07316*, 2016.

9. Goodfellow, I., Bengio, Y., & Courville, A. *Deep Learning*, MIT Press, 2016.