ROAD SIGN DETECTION USING MAMBA TRANSFORMER

A PROJECT REPORT

Submitted by

B. DHANASEKAREN (412321104005)

B. HEMANTH (412321104019)

R. KARTHI (412321104027)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SRI RAMANUJAR ENGINEERING COLLEGE

ANNA UNIVERSITY: CHENNAI 600 025

MAY 2025

ROAD SIGN DETECTION USING MAMBA TRANSFORMER

A PROJECT REPORT

Submitted by

B. DHANASEKAREN (412321104005)

B. HEMANTH (412321104019)

R. KARTHI (412321104027)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SRI RAMANUJAR ENGINEERING COLLEGE

ANNA UNIVERSITY: CHENNAI 600 025

MAY 2025

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this Project Report "**ROAD SIGN DETECTION USING MAMBA TRANSFORMER**" is the bonafide work of

B. DHANASEKAREN	**(412321104005)**
B. HEMANTH	**(412321104019)**
R. KARTHI	**(412321104027)**

Who carried out the project work under my supervision.

SIGNATURE
Mr. P. SIVA KUMAR, M.E.,
HEAD OF THE DEPARTMENT
Associate Professor,
Computer Science and Engineering,
Sri Ramanujar Engineering College,
Kolapakkam, Vandalur,
Chennai - 600127.

SIGNATURE
Mrs. K. LAKSHMI, M.E.,
SUPERVISOR
Assistant Professor,
Computer Science and Engineering,
Sri Ramanujar Engineering College,
Kolapakkam, Vandalur,
Chennai - 600127.

Submitted the project work and viva Examination held on …….
at Sri Ramanujar Engineering College, Chennai-127

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

At the juncture of presenting this project, We have a immense pleasure in expressing our heart-felt thanks to various personnel who have helped us at various levels to complete this project successfully.

At the outset, We express our respectful and sincere thanks with deep sense of gratitude to our beloved correspondent **THIRU.M.NITHYASUNDAR, M.COM., M.SC.,M.PHIL.,** and our beloved secretary **THIRU.G.KAMARAJ, B.A.** for providing all the necessary facility & guiding us in a right path of life with their enlightened wisdom.

Words are inadequate to express our feelings of gratitude to thank our beloved Principal **PROF.DR.A.DHANAPAL, M.E.,Ph.D.,** for his constant motivation and encouragement.

We feel greatly indebted to our H.O.D **Mr.P.SIVAKUMAR, M.E.,** for his propellant guidance, kind advice and encouragement. We would like to express our sincere thanks to our Project Coordinator **Mrs.K.LAKSHMI, M.E.,** they always provide us with their scholarly ideas, guidance and suggestions for monitoring our performance throughout the Project work.

Our heartfelt thanks to Project guide **Mrs.K.LAKSHMI, M.E.,** for her continual guidance & support with her suggestion for the successful completion of the project work.

One personal note, We would like to use this opportunity to extend our heartiest and respectable thanks to all teaching and non-teaching faculty members of our department, friends and all well wishers, who helped us in completing this project work successfully.

Sincerely We thank our Parents who stand behind us in each and every steps with abundant blessings of the Almighty.

ABSTRACT

With the continuous evolution of intelligent transportation systems, ensuring road safety through advanced technologies has become a key priority. This project focuses on the development of a smart desktop application capable of identifying traffic signs using deep learning techniques. At the core of the system lies the MambaTransformerClassifier, a powerful model architecture designed to deliver highly accurate predictions from traffic sign images. The application is built using PyTorch for deep learning implementation and PyQt6 for creating a user-friendly and professional graphical interface. Users can upload images of traffic signs and instantly receive classification results along with spoken feedback. To further enhance user understanding, the system includes interactive analytics features that display classification accuracy and frequency trends using Matplotlib visualizations. Additional functionality includes CSV export of results, dark mode support, and real-time predictions optimized using mixed precision computation for efficiency. Looking ahead, this solution has the potential to be adapted for integration into embedded systems within vehicles. By providing reliable recognition of traffic signs, it supports safer driving environments and contributes to the future of semi-autonomous and autonomous navigation. The project stands as a strong example of how artificial intelligence can be applied to address critical challenges in modern transportation.

TABLE OF CONTENTS

CHAPTER	TITLE	PAGE NO
	ABSTRACT	IV
	TABLE OF CONTENTS	V
	LIST OF FIGURES	X
	LIST OF ABBREVIATIONS	XI
1	INTRODUCTION	1
2	LITERATURE SURVEY	2
3	SYSTEM ANALYSIS	3
	3.1 EXISTING SYSTEM	3
	3.1.1 DISADVANTAGES	3
	3.2 PROPOSED SYSTEM	3
	3.3 TYPES OF MODULES	3
	3.3.1 CORE DEEP LEARNING MODULE (TRAFFIC SIGN CLASSIFIER)	4
	3.3.2 GRAPHICAL USER INTERFACE(GUI)	4
	3.3.3 PERFORAMCE ANALYTICS & VISUALIZATION	4

	3.3.4 REAL-TIME AUDIO FEEDBACK	4
	3.3.5 IMAGE PREPROCESSING FOR CLASSIFICATION	4
	3.3.6 DATA EXPORT FOR ANALYSIS	4
	3.3.7 MOBILE AND EMBEDDED PLATFORM COMPATIBILITY	4
	3.3.8 OPTIMIZATION FOR EFFICIENCY	5
	3.3.9 USER INTERFACE ELEMENTS	5
	3.3.10 DATA MANAGEMENT AND ANALYSIS	5
4	SYSTEM SPECIFICATIONS	6
	4.1 HARDWARE REQUIREMENTS	6
	4.2 SOFTWARE REQUIREMENTS	6
5	SYSTEM DESIGN	7
	5.1 INTRODUCTION	7
	5.1.1 LOGICAL DESIGN	7
	5.1.2 PHYSICAL DESIGN	8
	5.2 ARCHITECTURE DIAGRAM	9

5.3 USECASE DIAGRAM **10**

5.4 CLASS DIAGRAM **11**

5.5 DEPLOYMENT DIAGRAM **12**

5.6 PACKAGE DIAGRAM **13**

6 DEVELOPMENT TOOLS **14**

6.1 INTRODUCTION **14**

6.2 SOFTWARE DEVELOPMENT TOOLS **14**

6.2.1 PYTHON 3.8+ **14**

6.2.2 PyTorch **14**

6.2.3 PyQt6 **14**

6.2.4 OpenCV **14**

6.2.5 Matplotlib **14**

6.2.6 Pandas & Numpy **14**

6.2.7 Pyttsx3 (Text-to-Speech Engine) **14**

6.2.8 Py-Installer **15**

6.2.9 Yolov5 **15**

6.2.10 DroidCam **15**

	6.3 HARDWARE DEVELOPMENT TOOLS	**15**
	6.3.1 Mobile Camera	**15**
	6.4 TEXT EDITORS AND IDEs	**15**
	6.4.1 VISUAL STUDIO CODE (VS CODE)	**15**
	6.5 OTHER UTILITIES	**15**
	6.5.1 JUPYTER NOTEBOOK	**15**
7	**TESTING**	**16**
	7.1 ACCEPTANCE TESTING	**16**
	7.2 INTEGRATION TESTING	**16**
	7.3 UNIT TESTING	**16**
	7.4 FUNCTIONAL TESTING	**16**
	CONCLUSION	**17**
	FUTURE ENHANCEMENTS	**18**

APPENDICES	TITLE	PAGE NO
I	SOURCE CODE	19
	MAMBA MODEL	19
	DATASET LOADER	22
	DATA TRAINER	24
	TRAFFIC SIGN CLASSIFIER	28
	CONVERT TO YOLO	41
	REAL TIME CLASSIFICATION (FOR LAPTOP CAM)	42
	REAL TIME CLASSIFICATION (FOR MOBILE CAM)	49
II	SNAPSHOTS	53
	REFERENCES	57

LIST OF FIGURES

FIGURE NO	NAME OF THE FIGURE	PAGE NO
5.2	ARCHITECTURE DIAGRAM	9
5.3	USECASE DIAGRAM	10
5.4	CLASS DIAGRAM	11
5.5	DEPLOYMENT DIAGRAM	12
5.6	PACKAGE DIAGRAM	12
1	INITIALIZATION OF TRAFFIC SIGN CLASSIFIER	53
1.1	INDEX OF TRAFFIC SIGN CLASSIFIER	53
1.2	IMAGE UPLOADATION	54
1.3	OUTPUT OF TRAFFIC SIGN CLASSIFIER	54
2	INITIALIZATION OF REAL TIME CLASSIFICATION LAPTOP CAM	55
2.1	OUTPUT OF REAL TIME CLASSIFICATION LAPTOP CAM	55
3	INITIALIZATION OF REAL TIME CALSSIFICATION MOBILE CAM	56
3.1	OUTPUT OF REAL TIME CLASSIFICATION MOBILE CAM	56

LIST OF ABBREVIATIONS

ONNX	**O**pen **N**eural **N**etwork **E**xchange
CSV	**C**omma-**S**eparated **V**alues
GUI	**G**raphical **U**ser **I**nterface
GTSRB	**G**erman **T**raffic **S**ign **R**ecognition
AMP	**A**utomatic **M**ixed **P**recision
AI	**A**rtificial **I**ntelligence
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
CNN	**C**onvolutional **N**eural **N**etwork
IJCNN	**I**nternational **J**oint **C**onference on **N**eural **N**etworks
TSC	**T**ime **S**eries **C**lassification
ACM	**A**ssociation for **C**omputing **M**achinery
TIST	ACM **T**ransactions on **I**ntelligent **S**ystems and **T**echnology
ADAS	**A**dvanced **D**river **A**ssistance **S**ystems
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
GPU	**G**raphics **P**rocessing **U**nit
CPU	**C**entral **P**rocessing **U**nit
RAM	**R**andom **A**ccess **M**emory
GB	**G**iga **B**yte
SSD	**S**olid **S**tate **D**rives
HD	**H**ard **D**rive
AMD	**A**dvanced **M**icro **D**evices
TB	**T**era **B**yte
HDD	**H**ard **D**isk **D**rives
USB	**U**niversal **S**erial **B**us

TTS	**T**ext-**T**o-**S**peech
VS	**V**isual **S**tudio
UI	**U**ser **I**nterface
PIL	**P**ython **I**maging **L**ibrary
DPI	**D**ots **P**er **I**nch
YOLOv5	**Y**ou **O**nly **L**ook **O**nce version 5

CHAPTER 1

INTRODUCTION

The transportation sector has seen rapid advancements due to artificial intelligence, computer vision, and embedded systems. These technologies have enabled the rise of intelligent traffic management and driver assistance systems, essential for enhancing road safety. With the increasing number of vehicles, there's a growing demand for solutions that can minimize human error and improve driver awareness. One critical function in such systems is the real-time detection and classification of traffic signs, supporting both human drivers and autonomous vehicles. This project focuses on developing a desktop-based traffic sign classification system using a deep learning model known as the MambaTransformerClassifier. It is designed to accurately identify German traffic signs from images and deliver both visual and audio feedback to users. Acting as a driver assistance tool, the application mimics real-world scenarios of sign detection from vehicle-mounted cameras. Though currently developed for offline use, the system is built to scale for embedded automotive platforms in the future.

The motivation behind this work stems from the global emphasis on road safety and smart mobility. Reports by the World Health Organization highlight that many road accidents occur due to driver inattention or violation of traffic rules. By automating traffic sign recognition and providing alerts, the system can assist drivers in real time, reducing dependence on their visual memory and potentially preventing accidents. Model training leverages the German Traffic Sign Recognition Benchmark (GTSRB) dataset, which includes over 50,000 labeled images with diverse real-world conditions. The MambaTransformerClassifier combines convolutional layers with transformer-based attention to extract meaningful features efficiently. This hybrid structure ensures robust learning while maintaining computational performance, making it ideal for future hardware-constrained deployment.

The user interface, built using PyQt6, allows users to upload images, receive predictions with confidence levels, and hear spoken sign names via integrated text to speech. Additional features include dark mode, batch prediction, CSV export, and analytics visualization using Matplotlib. The system supports responsive performance using multi-threading and optimizations like mixed-precision inference with PyTorch AMP, ensuring efficient operation even on low-resource systems. Looking ahead, the application could be integrated into smart helmets, vehicle dashboards, or edge devices like Raspberry Pi and Jetson Nano for real-time road monitoring. Its modular design supports future upgrades such as video stream processing, multilingual audio alerts, and dynamic rule enforcement. This project illustrates how deep learning and thoughtful design can deliver practical, life-saving solutions, contributing to the broader vision of safe, intelligent transportation.

CHAPTER 2

LITERATURE SURVEY

1. **Real-Time Traffic Sign Recognition System Based on Efficient Deep Learning Models**
   Authors: Zhe Liu, Xiangyang Ji, Fei Gao
   IEEE Transactions on Intelligent Transportation Systems, 2020
   Developed a real-time system using CNN and data augmentation for robust recognition under challenging lighting and occlusion conditions.

2. **German Traffic Sign Recognition Benchmark (GTSRB): A Multi-class Classification Dataset**
   Authors: J. Stallkamp, M. Schlipsing, J. Salmen, C. Igel
   IEEE International Joint Conference on Neural Networks (IJCNN), 2011
   Provided the GTSRB dataset used widely for benchmarking traffic sign classification models.

3. **Traffic Sign Recognition Using Convolutional Neural Networks**
   Authors: Cireşan D.C., Meier U., Masci J., Schmidhuber J.
   Neural Networks Journal, 2012
   Demonstrated high classification accuracy using deep CNNs, influencing the design of future recognition architectures.

4. **An Embedded Real-Time Speed Limit Sign Recognition System**
   Authors: Tim Fingscheidt, Daniel Kienle
   IEEE Intelligent Vehicles Symposium, 2014
   Focused on embedded real-time traffic sign recognition, optimized for in-vehicle deployment.

5. **Lightweight Transformer Architecture for Efficient Traffic Sign Classification**
   Authors: Li Yang, Zhang Wei
   ACM Transactions on Intelligent Systems and Technology (TIST), 2022
   Proposed a transformer-based TSC system optimized for lightweight performance suitable for ADAS and embedded systems.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

Traffic sign recognition has traditionally relied on manual feature extraction methods like edge detection, color segmentation, and shape analysis. While somewhat effective in controlled environments, these techniques struggle with real-world challenges such as lighting, occlusion, and motion blur. Basic CNN-based models offer improvements but often lack robustness, generalization, and real-time performance. Many systems also lack desktop-friendly deployment, graphical or voice interfaces, performance visualization, and integration with embedded platforms—limiting their practical use in intelligent transportation systems.

3.1.1 DISADVANTAGES

- Manual feature extraction causes sensitivity to environmental changes and poor real-world performance.
- Basic CNNs lack robustness and generalization across diverse datasets.
- No real-time feedback or user-friendly interface limits practical usability.

3.2 PROPOSED SYSTEM

The proposed system overcomes existing limitations by introducing a deep learning-based desktop application for accurate traffic sign recognition. It uses the Mamba Transformer Classifier, built with PyTorch, and features a user-friendly GUI developed in PyQt6. Users can upload images, get real-time predictions with audio alerts, and interact via mouse or voice. The system includes performance analytics, CSV export, and dark mode. Designed as a standalone application, it also supports future integration with embedded platforms like Raspberry Pi and car infotainment systems, offering both high accuracy and adaptability for smart vehicle environments.

3.3 TYPES OF MODULES

This project contains Ten modules for our application to work correctly. They are,

1. Core Deep Learning Module (Traffic Sign Classifier)

2. Graphical User Interface (GUI)

3. Performance Analytics and Visualization

4. Real-Time Audio Feedback

5. Image Preprocessing for Classification

6. Data Export for Analysis

7. Mobile and Embedded Platform Compatibility

8. Optimization for Efficiency

9. User Interface Elements

10. Data Management and Analysis

This traffic sign classification system is built using a combination of powerful tools and libraries, each contributing to the overall functionality of the application. Below is an overview of the key modules integrated into the systems.

3.3.1 Core Deep Learning Module (Traffic Sign Classifier)

The MambaTransformerClassifier, built with PyTorch, accurately classifies traffic signs and handles challenges like lighting and occlusion for reliable recognition.

3.3.2 Graphical User Interface (GUI)

The PyQt6-based GUI provides a user-friendly interface with image upload, predictions, dark mode, and a clear, intuitive layout.

3.3.3 Performance Analytics and Visualization

Matplotlib visualizes training metrics like accuracy and loss, aiding in performance tracking and model improvement.

3.3.4 Real-Time Audio Feedback

pyttsx3 enables audio feedback by reading out detected traffic signs, offering hands-free interaction and improved accessibility.

3.3.5 Image Preprocessing for Classification

OpenCV handles image preprocessing like resizing and normalization, ensuring traffic sign images are optimized for accurate classification.

3.3.6 Data Export for Analysis

The system can export classification results to CSV, allowing users to log predictions and accuracy scores for performance analysis.

3.3.7 Mobile and Embedded Platform Compatibility

The system is designed for easy deployment on mobile and embedded platforms like Raspberry Pi, enabling use in vehicles and smart city applications.

3.3.8 Optimization for Efficiency

The system uses mixed-precision training and optimized preprocessing to reduce resource usage, ensuring smooth performance even on low-end devices.

3.3.9 User Interface Elements

The PyQt6 interface features image upload buttons, result display areas, and a clean layout, making the application visually appealing and easy to use.

3.3.10 Data Management and Analysis

Pandas is used to manage Train.csv and Test.csv files, enabling efficient data handling for model training and evaluation.

CHAPTER 4

SYSTEM SPECIFICATION

This system specification provides an overview of both the **software** and **hardware** requirements needed to successfully develop, run, and deploy the traffic sign classification system. Depending on the scale of deployment and the platform (desktop or embedded), the hardware and software specifications can be adjusted accordingly. The recommended specifications ensure smooth operation during the training phase and real-time image classification, along with a user-friendly interface for optimal performance.

4.1 Hardware Requirements

The hardware requirements for the traffic sign recognition system are optimized for efficient training and real-time inference. A multi-core processor like Intel i5/i7 or AMD Ryzen 5/7 is recommended for smooth GUI and data processing. A dedicated GPU such as NVIDIA GTX 1660 or higher ensures fast model training and inference. At least 8 GB of RAM is required, with 16 GB preferred for stable multitasking. A minimum of 10 GB free disk space is necessary, and an SSD is recommended for faster data access. The system runs on Windows 10/11 (64-bit) and is adaptable to Linux for embedded use. Audio hardware (speakers/headphones) supports text-to-speech, and standard input/output peripherals improve usability. The hardware specifications determine the system's capacity to handle deep learning computations, real-time image classification, and GUI rendering. These requirements vary based on whether the system is being run for training, inference, or both.

4.2 Software Requriements

The software requirements for the Traffic Sign Classification System are tailored to ensure smooth development, execution, and user interaction. The application is developed using Python 3.9 or higher, which serves as the core programming environment. PyTorch is used for building and deploying the MambaTransformerClassifier deep learning model, providing flexibility and efficient performance for training and inference. PyQt6 is employed for designing the graphical user interface (GUI), offering a modern and user-friendly experience. Additional Python libraries such as Matplotlib are required for visualizing performance metrics, while pyttsx3 is used for text-to-speech functionality. The system is best run on Windows 10/11 (64-bit), though it is adaptable for Linux environments, particularly when deploying to embedded platforms like Raspberry Pi. Other essential tools include NumPy, OpenCV for image preprocessing, and pandas for CSV operations. Installing all dependencies via pip and managing the environment with tools like virtualenv or Anaconda ensures compatibility and ease of setup.

CHAPTER 5

SYSTEM DESIGN

5.1 INTRODUCTION

System design forms the architectural foundation of the traffic sign recognition application, ensuring seamless integration of the deep learning model and GUI. This chapter details the system's logical structure, data flow, and physical setup, emphasizing flexibility, accuracy, scalability, and user experience. A well-planned design is crucial for smooth development, efficient performance, and future enhancements.

5.1.1 Logical Design

Logical design explains how the system works conceptually, focusing on module connections and data flow from input to output, without involving hardware details. It ensures that each component functions cohesively to achieve accurate and efficient traffic sign recognition.

1. **User Input Module**
   Allows users to upload traffic sign images via GUI or voice, supporting formats like .jpg, .jpeg, and .png.
2. **Preprocessing Module**
   Uses OpenCV to resize, normalize, and format images for model input, ensuring compatibility with model requirements.
3. **Classification Engine**
   Uses the MambaTransformerClassifier (PyTorch) to classify images and return the traffic sign label with a confidence score.
4. **Output Display Module**
   Shows predictions and confidence scores on the GUI with audio feedback (pyttsx3) and a visual confidence progress bar.
5. **Performance Analytics Module**
   Displays training graphs (accuracy/loss per epoch) using Matplotlib to monitor and evaluate model performance.
6. **Data Export & Logging Module**
   Enables exporting results to CSV and logs classification history for tracking and analysis.
7. **GUI/Voice Control Module**
   Built with PyQt6, features dark mode, shortcuts, tooltips, and voice control for a smooth and accessible user experience.

5.1.2 Physical Design

While logical design focuses on "how things work," physical design is about **"How things are implemented and deployed"** in the real world. It details the structure of files, software environments, hardware configurations, and execution workflow.

Hardware Requirements

- **Processor:** Intel i5 or higher (recommended for model inference)
- **RAM:** Minimum 8 GB (16 GB preferred)
- **GPU:** NVIDIA GeForce RTX 2050 or above with CUDA support (for faster training and inference)
- **Disk Space:** Minimum 10 GB (for datasets, models, and environment)
- **Optional:** Microphone (for voice commands), speakers (for audio feedback)

Software Requirements

- **Operating System:** Windows 10/11 (64-bit)
- **Programming Language:** Python 3.8+
- **Libraries & Frameworks:**
    - PyTorch – Deep learning model training and inference
    - PyQt6 – GUI development
    - OpenCV – Image processing
    - pyttsx3 – Text-to-speech
    - Matplotlib – Performance visualization
    - pandas, numpy – Data handling
    - pyinstaller – For building standalone .exe applications

Execution Workflow

1. User launches the application (main.py or .exe built using PyInstaller).
2. The GUI opens, and the user uploads an image.
3. The image is pre-processed and passed to the deep learning classifier.
4. The classifier returns predictions.
5. Results are displayed and spoken using the TTS engine.
6. Performance graphs can be viewed, and predictions exported to CSV.

5.2 ARCHITECTURE DIAGRAM

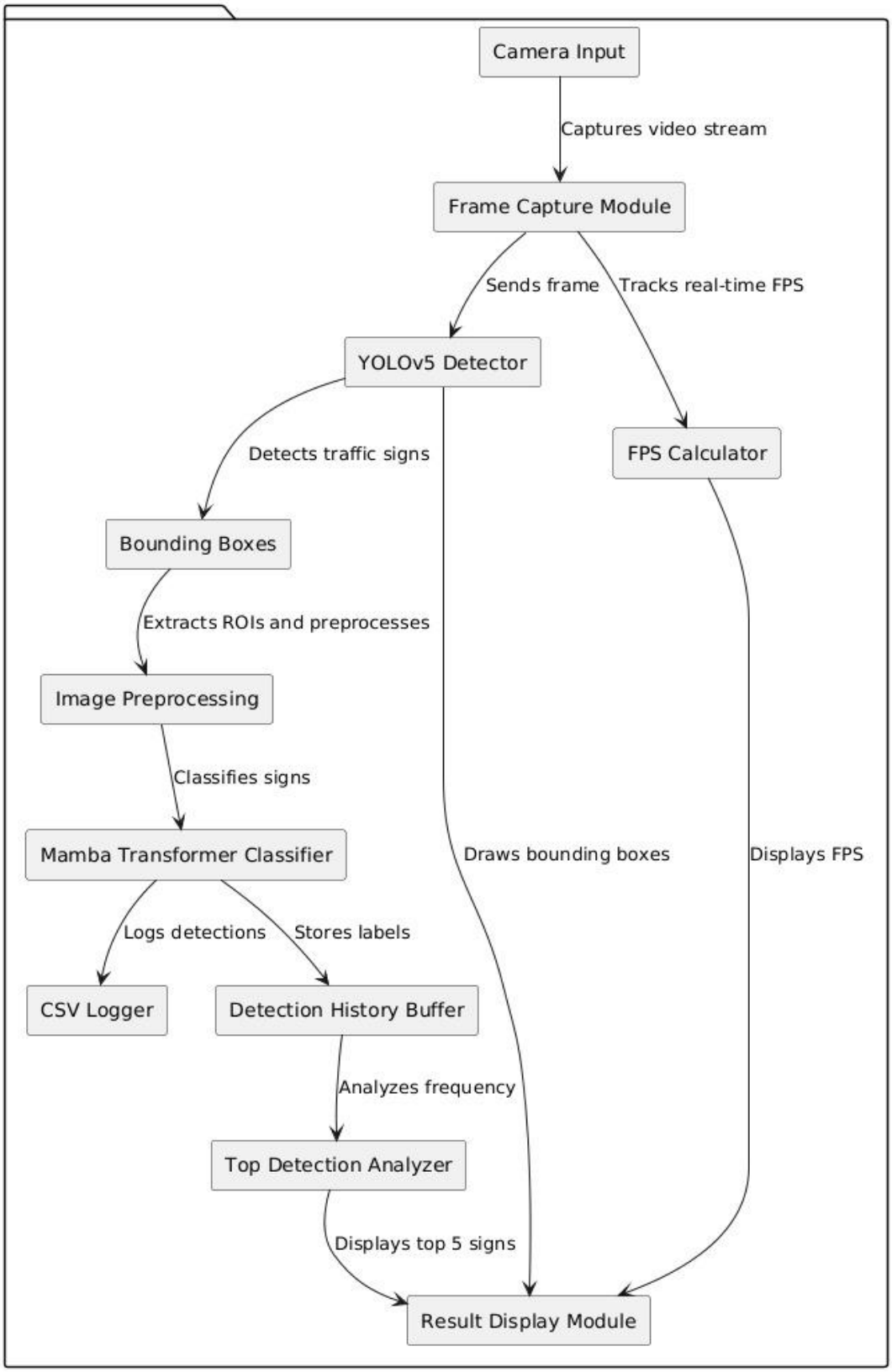


FIGURE 1: 5.2 ARCHITECTURE DIAGRAM

5.3 USECASE DIAGRAM

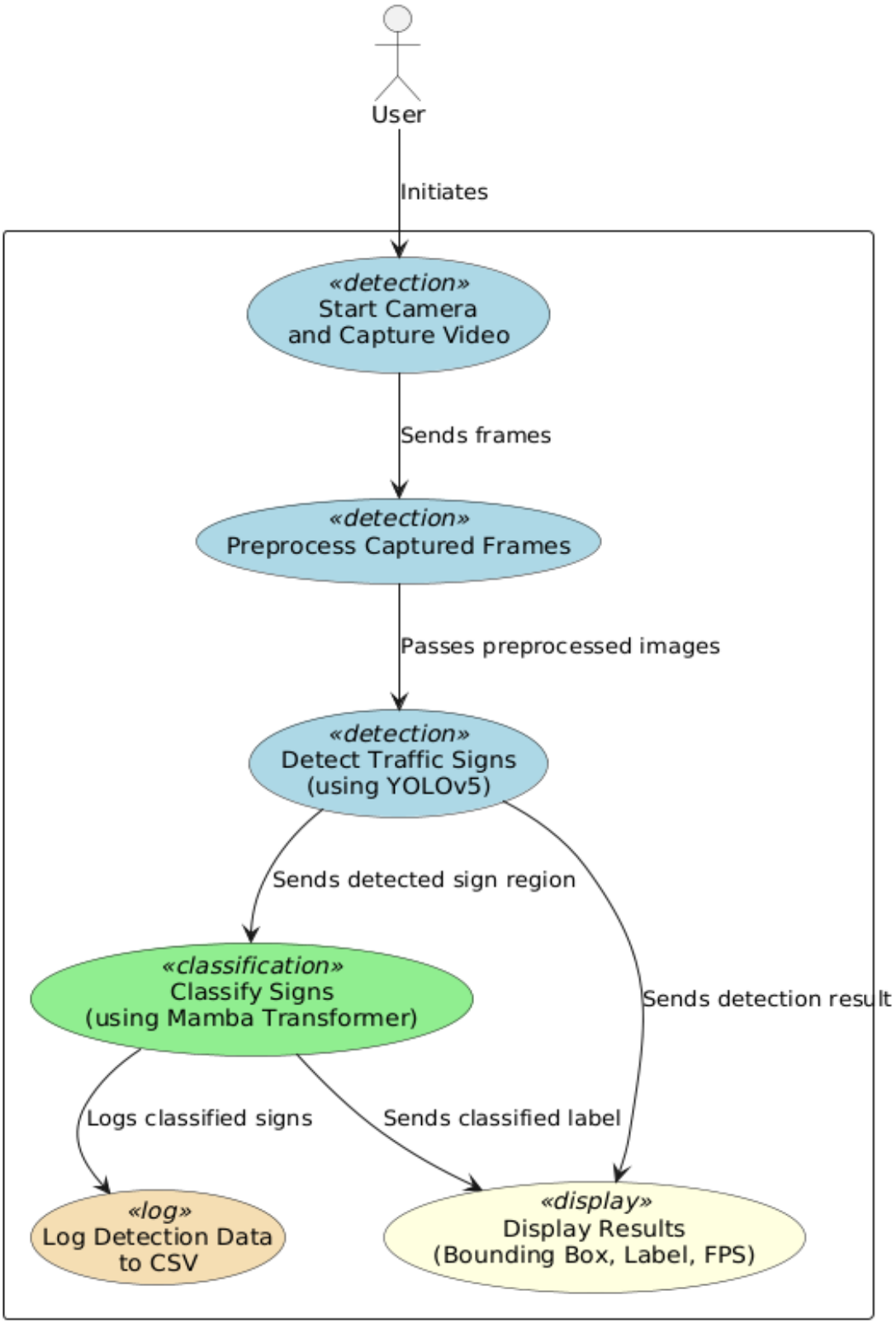


FIGURE 2: 5.3 USECASE DIAGRAM

5.4 CLASS DIAGRAM

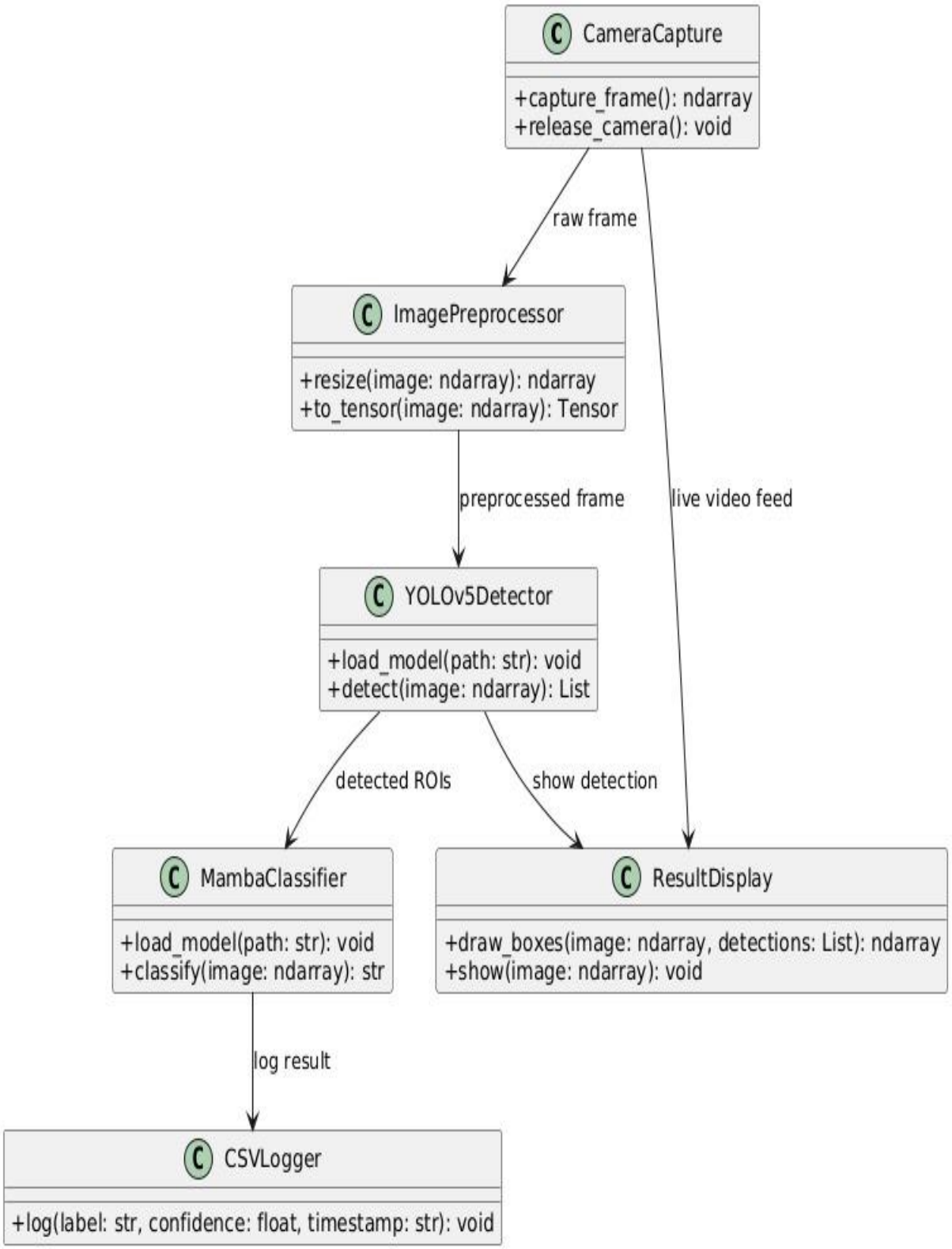


FIGURE 3: 5.4 CLASS DIAGRAM

5.5 DEPLOYMENT DIAGRAM

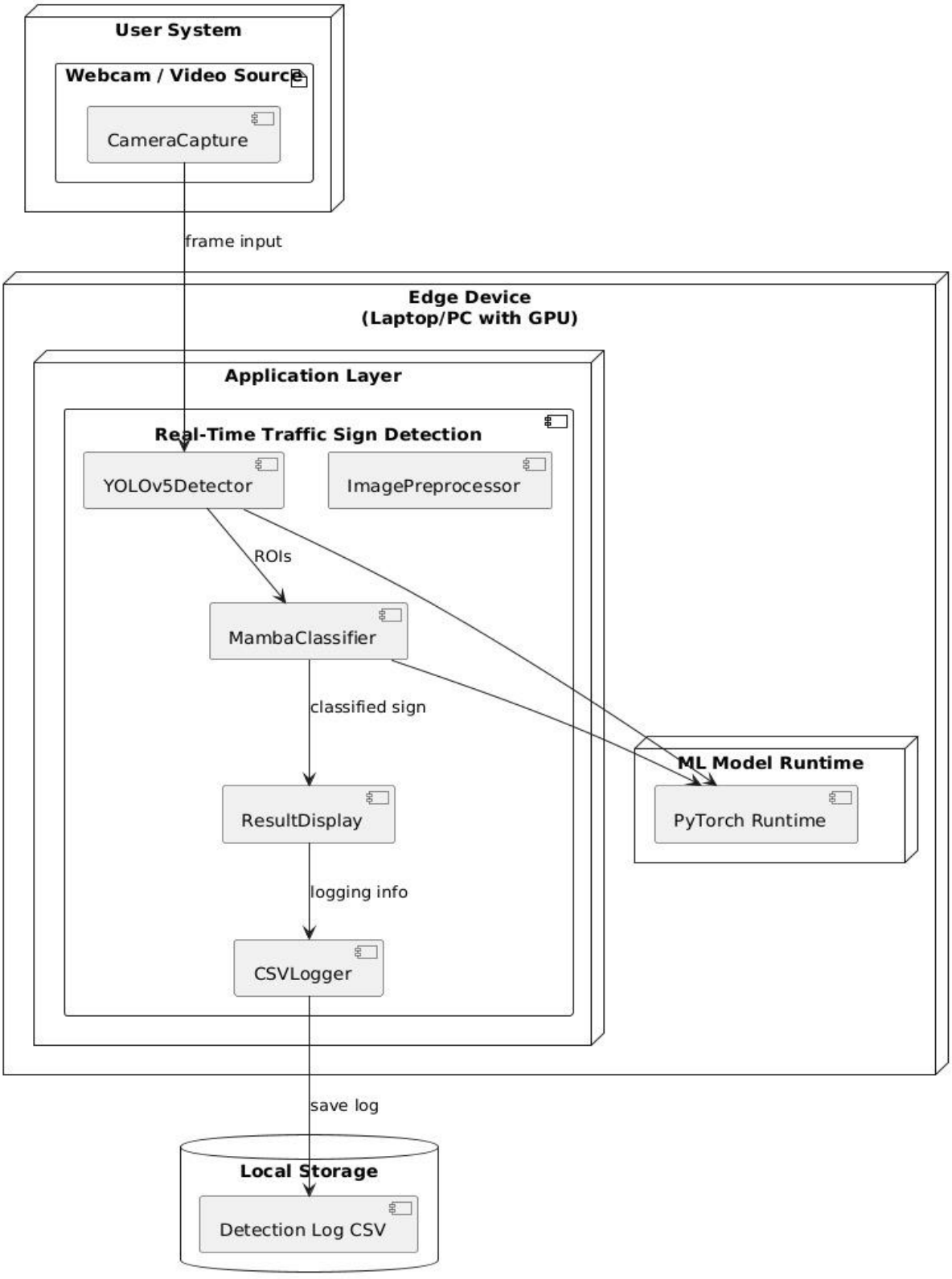


FIGURE 4: 5.5 DEPLOYMENT DIAGRAM

5.6 PACKAGE DIAGRAM

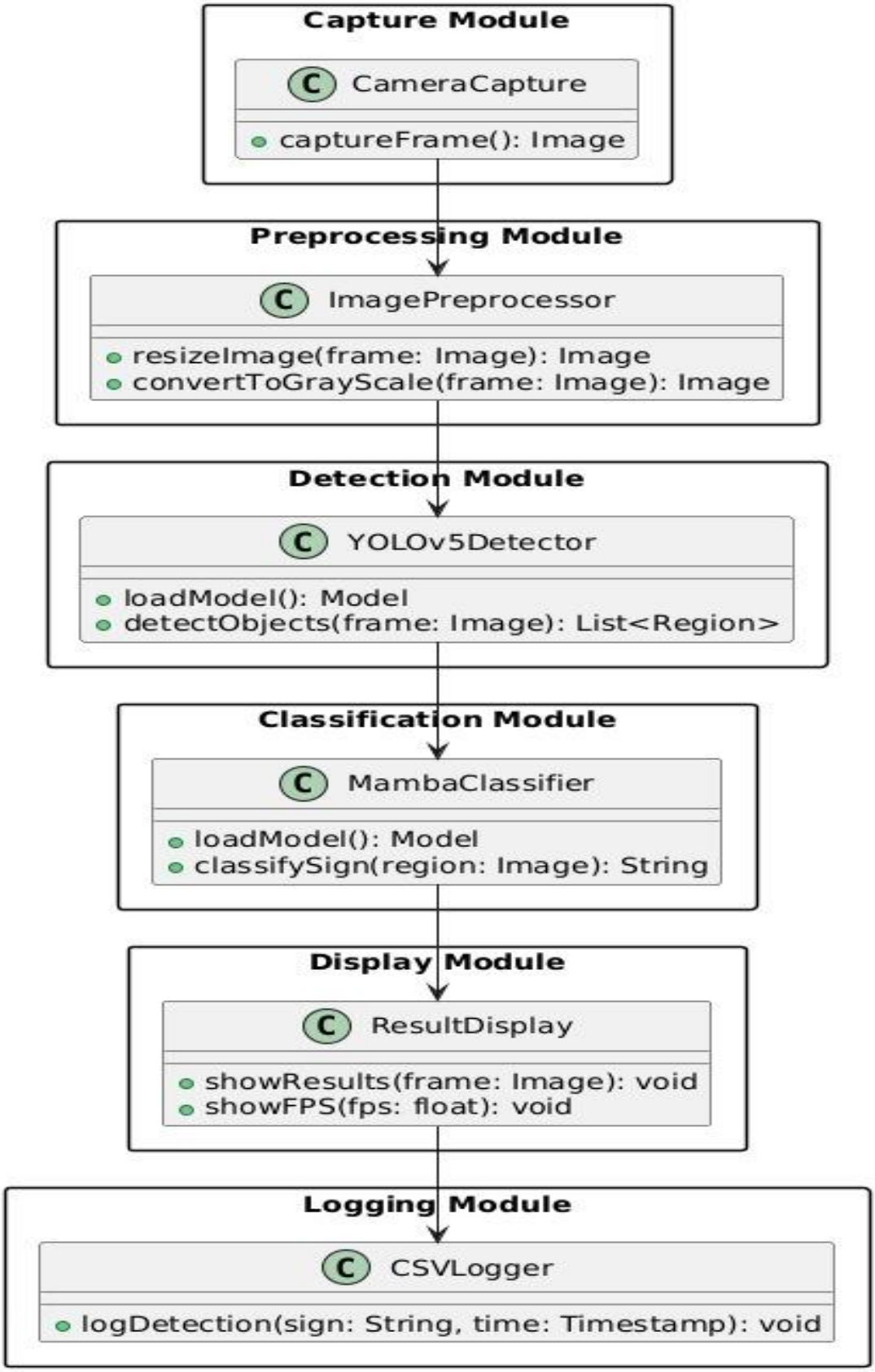


FIGURE 5: 5.6 PACKAGE DIAGRAM

CHAPTER 6

DEVELOPMENT TOOLS

6.1 INTRODUCTION

Development tools are crucial for efficient system building, testing, and deployment. This project used powerful frameworks, libraries, and hardware accelerators to create a robust traffic sign classification system. The chosen tools ensured high performance, smooth interaction, and scalability. This chapter overviews the software, hardware, and third-party tools used throughout the project, highlighting their role in achieving project goals effectively.

6.2 SOFTWARE DEVELOPMENT TOOLS

6.2.1 Python 3.8+

Python was chosen as the main language for its readability, rich library support, and compatibility with machine learning and GUI frameworks, making it ideal for this project.

6.2.2 PyTorch

PyTorch was used to develop the MambaTransformerClassifier due to its dynamic computation graph, GPU support, and ease of experimentation.

6.2.3 PyQt6

PyQt6 was used to build a modern, responsive GUI with features like drag-and-drop image upload, voice commands, and real-time result display.

6.2.4 OpenCV

OpenCV was used for image preprocessing, handling resizing, color conversion, and noise removal to prepare images for classification.

6.2.5 Matplotlib

Matplotlib was used to visualize training loss and accuracy, helping understand model performance and providing insights within the GUI.

6.2.6 Pandas & NumPy

Pandas managed CSV files and prediction logs, while NumPy handled tensor operations during preprocessing, ensuring efficient data management.

6.2.7 pyttsx3 (Text-To-Speech Engine)

pyttsx3 was used to provide voice feedback after classifications, enhancing interactivity and accessibility.

6.2.8 PyInstaller

PyInstaller packaged the project into a standalone Windows .exe, allowing users to run it without installing Python or dependencies.

6.2.9 YoloV5

YOLOv5 was used for fast, accurate real-time traffic sign detection, enabling efficient multi-sign detection with low latency.

6.2.10 DroidCam

DroidCam turns a smartphone into a webcam via Wi-Fi or USB, enabling low-latency video streaming. It was used here to connect a mobile camera to a laptop for flexible input.

6.3 HARDWARE DEVELOPMENT TOOLS

6.3.1 Mobile Camera

The mobile camera served as the primary video input source for the traffic sign detection system. It provided real-time footage, integrated with the laptop using the DroidCam application for seamless streaming.

6.4 TEXT EDITORS AND IDEs

6.4.1 Visual Studio Code (VS Code)

VS Code was the primary Integrated Development Environment used during development. Its wide plugin support, integrated terminal, and debugging tools helped improve productivity and code quality.

- Lightweight and highly customizable
- Auto-completion and linting for Python
- Integrated Git and Python terminal

6.5 OTHER UTILITIES

6.5.1 Jupyter Notebook

Used occasionally for testing model prototypes and visualizing intermediate results. It is particularly useful during the data exploration and model tuning phases.

CHAPTER 7

TESTING

Testing ensured the traffic sign recognition system met design goals through manual and automated checks. Each module, from the deep learning model to the PyQt6 GUI, was tested individually and together to catch edge cases and verify features like voice feedback, CSV export, and real-time inference. This chapter details the testing methods used to deliver a reliable, user-friendly system.

7.1 ACCEPTANCE TESTING

Acceptance testing confirmed that the system met the original requirements and user expectations, focusing on end-user usability. The tests validated accurate classification of various traffic signs, ensured the GUI supported image upload with instant predictions, verified audio feedback after each classification, and checked the CSV export and dark mode functionalities. All acceptance criteria were successfully met, and the application was deemed ready for deployment.

7.2 INTEGRATION TESTING

Integration testing ensured smooth interaction between system modules, including the PyTorch model, PyQt6 GUI, OpenCV preprocessing, and pyttsx3 voice feedback. Key points tested were seamless data flow from image upload to GUI output, coordination of GUI events with model inference, correct display of performance graphs in PyQt6, and timely voice output after classification. Minor issues were resolved, resulting in a well-coordinated application.

7.3 UNIT TESTING

Unit testing validated core components individually, including image preprocessing, prediction, CSV export, voice announcements, and dark mode toggle. Each function was tested with expected inputs and outputs, passing without critical issues, confirming their accuracy and reliability.

7.4 FUNCTIONAL TESTING

Functional testing validated the system’s behavior against requirements by using diverse traffic sign images, including varying lighting, occlusions, and backgrounds. Through real-time GUI interaction and test uploads, the system consistently delivered accurate classifications, reliable voice feedback, and stable performance, meeting all functional expectations.

CONCLUSION

The development of this Traffic Sign Classification System demonstrates the effectiveness of deep learning in enhancing road safety through accurate visual recognition. Using the MambaTransformerClassifier and PyTorch, the system achieves high accuracy in classifying traffic signs under varied conditions. The user-friendly desktop application built with PyQt6 offers real-time predictions along with features like voice feedback, dark mode, performance visualization with Matplotlib, and CSV export, enhancing usability and practicality. Thorough testing ensured the system's stability and efficiency, while its modular design supports future upgrades such as real-time video recognition and embedded system integration. This combination of advanced AI techniques and interactive design makes the system well-suited for both academic purposes and real-world deployment. Overall, this project provides a solid foundation for further advancements in intelligent transportation and computer vision technologies.

Future work could explore multilingual voice support and dynamic traffic rule enforcement based on location data, expanding its applicability in diverse driving environments. The adaptability of the system also opens opportunities for integration with smart city infrastructure, contributing to safer and more efficient transportation networks. Additionally, integrating machine learning-driven predictive analytics could further enhance proactive traffic management and accident prevention strategies. Continuous improvements in hardware acceleration and model optimization will also ensure the system remains effective on resource-constrained devices.

FUTURE ENHANCEMENTS

While the current version of the Traffic Sign Classification System meets its core objectives of accurate recognition and user-friendly interaction, there are several opportunities for further enhancement and real-world application. The following future enhancements are proposed to elevate the system's capabilities and adaptability:

1. Real-Time Video Stream Classification

Currently, the system is designed for static image classification. In the future, it can be extended to process real-time video feeds from dashcams or surveillance cameras. This would allow continuous monitoring and instant recognition of traffic signs on the move, making the system suitable for deployment in smart vehicles.

2. Integration with Embedded Systems

The system can be optimized and deployed on lightweight embedded platforms such as Raspberry Pi or NVIDIA Jetson Nano. This would enable on-board traffic sign recognition in autonomous vehicles or driver assistance systems without the need for high-end computers.

3. Support for Multilingual Audio Feedback

At present, the voice output is delivered in English. To improve accessibility and adoption across diverse regions, support for multilingual voice feedback can be added, allowing users to receive responses in their native languages.

4. Expansion to Include Traffic Violation Detection

The system can be enhanced to detect traffic violations such as wrong turns, signal jumping, or over speeding by analyzing vehicle behaviour in conjunction with recognized traffic signs, improving the overall intelligence of road monitoring.

5. Mobile Application Version

A lightweight mobile version of the application can be developed using frameworks like Flutter or React Native, enabling real-time traffic sign detection directly from smartphones, making it accessible to a broader audience.

6. Cloud-Based Logging and Analytics

Future versions can include integration with cloud services to store classification logs and generate analytical reports. This would be useful for fleet management systems, research institutions, or traffic authorities to study patterns and performance metrics.

APPENDICES – I

SOURCE CODE

MAMBA MODEL

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.amp import autocast, GradScaler  # Correct Import
torch.backends.cudnn.benchmark = True  # Optimized CUDA kernels
torch.backends.cudnn.deterministic = False
torch.set_float32_matmul_precision("high")
#  Optimized Depthwise Separable Convolution
class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super().__init__()
         self.depthwise = nn.Conv2d(in_channels, in_channels, kernel_size, stride, padding, groups=in_channels, bias=False, padding_mode='reflect')
        self.pointwise = nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False)
        self.norm = nn.GroupNorm(8, out_channels)
        self.act = nn.SiLU(inplace=True)  # In-place activation
        # He Initialization
        nn.init.kaiming_normal_(self.depthwise.weight, nonlinearity='relu')
        nn.init.kaiming_normal_(self.pointwise.weight, nonlinearity='relu')
    def forward(self, x):
        x = self.depthwise(x)
        x = self.pointwise(x)
        x = self.norm(x)
        return self.act(x)
```

```
#  Optimized Transformer Block with Flash Attention (if available)
class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.norm1 = nn.LayerNorm(embed_dim, elementwise_affine=False)
        self.norm2 = nn.LayerNorm(embed_dim, elementwise_affine=False)
        self.fc = nn.Sequential(
            nn.Linear(embed_dim, embed_dim * 4),
            nn.Linear(embed_dim * 4, embed_dim)
        )
        self.dropout = nn.Dropout(0.15)  # Prevent overfitting
    def forward(self, x):
        x = self.norm1(x)
        x = x + attn_out
        x = x + self.dropout(self.fc(self.norm2(x)))
        return x
#  Optimized Model
class MambaTransformerClassifier(nn.Module):
    def __init__(self, num_classes=43):
        super().__init__()
        self.layer1 = DepthwiseSeparableConv(3, 64)
        self.layer2 = DepthwiseSeparableConv(64, 128)
        self.pool = nn.AdaptiveAvgPool2d((8, 8))  #  Adaptive pooling
        # Flatten size calculation
        flatten_size = 128 * 8 * 8
        self.embedding = nn.Linear(flatten_size, 128)
        self.transformer = TransformerBlock(128, 4)
        self.fc = nn.Linear(128, num_classes)
        # Weight Initialization
```

```
        nn.init.trunc_normal_(self.embedding.weight, std=0.02)
        nn.init.trunc_normal_(self.fc.weight, std=0.02)
    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1).contiguous()
        x = x.squeeze(1)
        return self.fc(x)
#  Optimized Training Function with Mixed Precision & Error Handling
scaler = GradScaler(device="cuda")  # 🚀 Fixed FutureWarning
def train_step(model, dataloader, criterion, optimizer, device):
    model.train()
    for batch_idx, (images, labels) in enumerate(dataloader):
        try:
                images, labels = images.to(device, non_blocking=True), labels.to(device, non_blocking=True)
            optimizer.zero_grad()
                with autocast(device_type="cuda", dtype=torch.float16):  # 🚀 Fixed FutureWarning
                outputs = model(images)
                loss = criterion(outputs, labels)
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            if batch_idx % 10 == 0:  # Print every 10 batches to reduce overhead
                print(f"🎯 Batch {batch_idx+1}/{len(dataloader)} | Loss: {loss.item():.4f}")
        except torch.cuda.OutOfMemoryError as e:
            print(f"🔥 CUDA OOM at Batch {batch_idx+1}! Clearing Cache...")
```

```
            torch.cuda.empty_cache()
            continue
        except Exception as e:
            print(f"❌ Error in Batch {batch_idx+1}: {e}")
            import traceback
            print(traceback.format_exc())
            exit(1)
#  Optimized Model Initialization
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MambaTransformerClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-2)
if torch.cuda.is_available():
    model = torch.compile(model)  # 🚀 Torch.compile for Speed Optimization
```

DATASET LOADER

```
import os
import pandas as pd
from PIL import Image
from torch.utils.data import Dataset, DataLoader, random_split
import torchvision.transforms as transforms
import torch
class TrafficSignDataset(Dataset):
    def __init__(self, csv_file, img_dir, transform=None):
        """
        Traffic Sign Dataset Loader.
        :param csv_file: Path to the CSV file containing image paths and labels.
        :param img_dir: Root directory where images are stored.
```

```
        :param transform: Image transformations.
        """
        self.data = pd.read_csv(csv_file)
        self.img_dir = img_dir
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.data.iloc[idx]['Path'])
        # Check if image file exists
        if not os.path.exists(img_path):
        # Load Image
        image = Image.open(img_path).convert("RGB")
        label = int(self.data.iloc[idx]['ClassId'])
        # Apply transformations
        if self.transform:
        return image, label
#  Data transformations
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize to fixed size
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # Normalize
])
def get_dataloaders(batch_size=64, train_split=0.8):
    """
    Create train and validation DataLoaders.
    :return: train_loader, val_loader
    """
    dataset = TrafficSignDataset("dataset/Train.csv", "dataset", transform=transform)
    # Split dataset into train and validation
```

```
    train_size = int(train_split * len(dataset))
    val_size = len(dataset) - train_size
    train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
    # Create DataLoaders
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)
        val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)
    return train_loader, val_loader
# Test the dataset loading
if __name__ == "__main__":
    train_loader, val_loader = get_dataloaders()
    print(f"Train batches: {len(train_loader)}, Validation batches: {len(val_loader)}")
    # Display sample batch
    images, labels = next(iter(train_loader))
    print(f"Sample batch shape: {images.shape}, Labels: {labels[:5]}")
```

DATA TRAINER

```
import torch
import torch.nn as nn
import torch.optim as optim
import warnings
import traceback
from dataset_loader import get_dataloaders
from mamba_model import MambaTransformerClassifier
# Suppress only FutureWarnings & UserWarnings
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
class Trainer:
```

```
def __init__(self, config):
    try:
        print("📂 Loading dataset...")
      self.train_loader, self.val_loader = get_dataloaders(batch_size=config['batch_size'])
    except Exception as e:
        print("❌ Dataset loading failed:", e)
        print(traceback.format_exc())
        raise
    try:
        print("🛠️ Initializing model...")
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    except Exception as e:
        print("❌ Model initialization failed:", e)
        print(traceback.format_exc())
        raise
    try:
        print("⚙️ Setting up optimizer and loss function...")
        self.criterion = nn.CrossEntropyLoss()
        print("✅ Optimizer and loss function ready.")
    except Exception as e:
        print("❌ Optimizer setup failed:", e)
        print(traceback.format_exc())
        raise
    # Track best validation accuracy
    self.best_val_acc = 0.0
def train_epoch(self):
    self.model.train()
    running_loss = 0.0
```

```
    correct = 0
    total = 0
    try:
        for images, labels in self.train_loader:
            images, labels = images.to(self.device), labels.to(self.device)
            self.optimizer.zero_grad()
            with torch.amp.autocast(device_type="cuda"):
                outputs = self.model(images)
                loss = self.criterion(outputs, labels)
            self.scaler.scale(loss).backward()
            running_loss += loss.item()
            _, predicted = outputs.max(1)
            correct += predicted.eq(labels).sum().item()
            total += labels.size(0)
    except Exception as e:
        raise
    epoch_loss = running_loss / len(self.train_loader)
    epoch_acc = 100.0 * correct / total
    return epoch_loss, epoch_acc
def validate_epoch(self):
    self.model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    try:
        with torch.no_grad():
            for images, labels in self.val_loader:
                images, labels = images.to(self.device), labels.to(self.device)
                outputs = self.model(images)
```

```
                loss = self.criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = outputs.max(1)
                correct += predicted.eq(labels).sum().item()
                total += labels.size(0)
        except Exception as e:
            print("❌ Fatal Error in Validation Loop:", e)
            print(traceback.format_exc())
            raise
        return val_loss, val_acc
    def train(self, epochs):
        print(f"🚀 Training for {epochs} epochs on {self.device}...\n")
        for epoch in range(epochs):
            try:
                print(f"📌 Epoch {epoch + 1}/{epochs}")
                train_loss, train_acc = self.train_epoch()
                print(f"📉 Train Loss: {train_loss:.4f} | 🎯 Train Acc: {train_acc:.2f}%")
                val_loss, val_acc = self.validate_epoch()
                print(f"🔍 Validation Loss: {val_loss:.4f} | 🎯 Validation Acc: {val_acc:.2f}%")
                # Save model if validation accuracy improves
                if val_acc > self.best_val_acc:
                    self.best_val_acc = val_acc
                    torch.save(self.model.state_dict(), "mamba_transformer_best.pth")
            except Exception as e:
                print(f"❌ Fatal Error in Epoch {epoch+1}:", e)
                print(traceback.format_exc())
                raise
        print("\n🎯 **Training Summary** 🎯")
```

```
print(f"🔍 Best Validation Acc: {self.best_val_acc:.2f}%")
if __name__ == "__main__":
    config = {
        'batch_size': 64,
        'lr': 0.001,
        'epochs': 15
    }
    try:
        trainer = Trainer(config)
        trainer.train(config['epochs'])
    except Exception as e:
        print("❌ Fatal error in main execution:", e)
        print(traceback.format_exc())
```

TRAFFIC SIGN CLASSIFIER

```
import sys
import os
import torch
import numpy as np
import pyttsx3
from PyQt6.QtWidgets import (
    QApplication, QWidget, QLabel, QPushButton, QFileDialog, QVBoxLayout,
     QHBoxLayout, QScrollArea, QFrame, QGridLayout, QSizePolicy, QMessageBox, QPushButton, QDialog
)
from PyQt6.QtGui import QPixmap, QPainter, QPen
from PyQt6.QtCore import Qt, QRect, QThread, pyqtSignal
from PIL import Image
from torchvision import transforms
```

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import screeninfo
from mamba_model import MambaTransformerClassifier  # Ensure this file is present
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
import queue
import threading
import csv
# Load trained model securely
num_classes = 43
model = MambaTransformerClassifier(num_classes=num_classes)
weights_only=True), strict=False)
model.eval()
# Get screen DPI dynamically
screen = screeninfo.get_monitors()[0]  # Detect primary screen
dpi_scale = screen.width / 1920  # Scale based on 1080p resolution
# Dynamically adjust font sizes
mpl.rcParams.update({
    "axes.titlesize": 14 * dpi_scale,  # Scale Title Size
    "axes.labelsize": 12 * dpi_scale,  # Scale Labels
    "figure.titlesize": 16 * dpi_scale,  # Scale Figure Title
    "figure.dpi": 100 * dpi_scale,  # Adjust DPI dynamically
})
# Initialize text-to-speech engine
tts_engine = pyttsx3.init()
class TTSWorker(QThread):
    """Thread for text-to-speech to prevent UI blocking."""
```

```
    finished = pyqtSignal()
    def __init__(self, text):
        super().__init__()
        self.text = text
    def run(self):
        tts_engine = pyttsx3.init()
        self.finished.emit()  # Signal when done
# Define class labels
class_labels = {
    0: "Speed Limit 20 km/h", 1: "Speed Limit 30 km/h", 2: "Speed Limit 50 km/h",
    3: "Speed Limit 60 km/h", 4: "Speed Limit 70 km/h", 5: "Speed Limit 80 km/h",
    6: "End of Speed Limit 80", 7: "Speed Limit 100 km/h", 8: "Speed Limit 120 km/h",
    21: "Double Curve", 22: "Bumpy Road", 23: "Slippery Road",
    24: "Road Narrows From Right", 25: "Road Work Under Process", 26: "Traffic Signals Ahead",
     27: "Pedestrian Crossing", 28: "School Zone or Children Crossing", 29: "Cycles Crossing",
     30: "Slipperiness Due To Snow or Ice", 31: "Deer Crossing Area", 32: "End Of All Restrictions",
    33: "Turn Right Ahead", 34: "Turn Left Ahead", 35: "Go Straight",
    36: "Go Straight or Turn Right Ahead", 37: "Go Straight or Turn Left Ahead", 38: "Pass on Right",
}
  # Update with actual labels
# Image transformation pipeline
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
])
class AccuracyBar(QWidget):
```

```
    def __init__(self, parent=None):
        super().__init__(parent)
        self.accuracy = 0
        self.dark_mode = True  # Default is dark mode
    def setAccuracy(self, accuracy):
        self.accuracy = accuracy
        self.update()
    def setDarkMode(self, dark_mode):
        """ Updates color scheme based on dark mode setting """
        self.dark_mode = dark_mode
        self.update()
        text_color = Qt.GlobalColor.white if self.dark_mode else Qt.GlobalColor.black
        pen = QPen(line_color, 4)
        painter.setPen(pen)
        painter.drawLine(10, 20, self.width() - 10, 20)
        marker_x = int(10 + (self.accuracy / 100) * (self.width() - 20))
        pen.setColor(marker_color)
        painter.drawPoint(marker_x, 20)
        painter.setPen(text_color)
        painter.drawText(QRect(0, 30, 50, 20), Qt.AlignmentFlag.AlignLeft, "0%")
        painter.drawText(QRect(self.width() - 50, 30, 50, 20), Qt.AlignmentFlag.AlignRight,
"100%")
                    painter.drawText(QRect(self.width() // 2 - 25, 30, 50, 20),
Qt.AlignmentFlag.AlignCenter, f"{self.accuracy:.1f}%")
class TrafficSignClassifier(QWidget):
    def __init__(self):
        super().__init__()
        self.dark_mode = True
        self.image_paths = []
```

```
        self.initUI()
        self.tts_threads = []  # ✅ Store active TTS threads
        self.tts_queue = queue.Queue()
        self.tts_thread = threading.Thread(target=self.process_speech_queue, daemon=True)
        self.tts_thread.start()
        self.predictions = {}
    def save_results_to_csv(self):
        """Saves batch classification results to a CSV file."""
        if not self.predictions:
            QMessageBox.warning(self, "No Data", "No classification results available to save.")
            return
        file_path, _ = QFileDialog.getSaveFileName(self, "Save Report", "", "CSV Files (*.csv)")
            writer = csv.writer(file)
            writer.writerow(["Image File", "Predicted Class", "Accuracy (%)"])
            for image_path, (predicted_text, accuracy) in self.predictions.items():
                                writer.writerow([os.path.basename(image_path),  predicted_text, f"{accuracy:.2f}"])
        QMessageBox.information(self, "Report Saved", f"Results saved to {file_path}")
    def process_speech_queue(self):
        """Continuously processes speech requests from the queue."""
        tts_engine = pyttsx3.init()
        while True:
            text = self.tts_queue.get()
            if text is None:
                break  # Exit loop if None is received
            tts_engine.say(text)
            tts_engine.runAndWait()
```

```
def speak(self, text):
    self.tts_queue.put(text)
def initUI(self):
    self.setWindowTitle("Traffic Sign Classifier")
    self.setGeometry(100, 100, 900, 600)
    # Define buttons first
    self.upload_button = QPushButton("Upload Images", self)
    self.upload_button.clicked.connect(self.upload_images)
    self.dark_mode_button = QPushButton("Toggle Dark Mode", self)
    self.dark_mode_button.clicked.connect(self.toggle_dark_mode)
    self.set_dark_mode_styles()
    # Scroll area for displaying multiple images
    self.results_container = QWidget()
    self.results_layout = QGridLayout(self.results_container)  # Use GridLayout here
    self.results_container.setLayout(self.results_layout)  # Ensure layout is applied
    self.scroll_area.setWidget(self.results_container)
    layout = QVBoxLayout()
    layout.addWidget(self.upload_button)
    #analytics
    # Button to Show Class Distribution
    self.analytics_button = QPushButton("Show Analytics", self)
    self.analytics_button.clicked.connect(self.show_class_distribution)
    layout.addWidget(self.analytics_button)
    # Button to Show Accuracy Trend
    self.accuracy_trend_button = QPushButton("Show Accuracy Trend", self)
    layout.addWidget(self.accuracy_trend_button)
    self.setLayout(layout)
    #to save csv file
    self.save_csv_button = QPushButton("Save Results to CSV", self)
```

```
self.save_csv_button.clicked.connect(self.save_results_to_csv)
layout.addWidget(self.save_csv_button)
if self.dark_mode:
    self.setStyleSheet("""
        background-color: #121212;
        color: white;
        font-size: 16px;
        border-radius: 10px;
    """)
    self.upload_button.setStyleSheet("""
        QPushButton {
            background-color: #1E88E5;
            color: white;
            padding: 8px;
            border-radius: 5px;
        }
        QPushButton:hover { background-color: #1565C0; }
    """)
        QPushButton:hover { background-color: #666; }
    """)
else:
    self.setStyleSheet("""
        background-color: white;
        color: black;
        font-size: 16px;
        border-radius: 10px;
    """)
    self.upload_button.setStyleSheet("""
        QPushButton {
```

```
            background-color: #0D47A1;
            color: white;
            padding: 8px;
            border-radius: 5px;
        }
        QPushButton:hover { background-color: #1565C0; }
    """)
        QPushButton:hover { background-color: #BBB; }
    """)
    def show_matplotlib_figure(self, figure):
"""Displays a Matplotlib figure inside a PyQt6 dialog with a grey-themed toolbar."""
dialog = QDialog(self)
dialog.setWindowTitle("Analytics")
dialog.setGeometry(200, 200, 900, 650)
layout = QVBoxLayout()
#  Apply Grey Theme to Matplotlib Toolbar
toolbar.setStyleSheet("""
    QToolBar {
        background-color: #444;
        border: 1px solid #666;
    }
    QToolButton {
        color: white;
        background-color: #666;
        border: none;
        padding: 5px;
        margin: 2px;
    }
    QToolButton:hover {
```

```
            background-color: #888;
        }
        QLabel {
            color: white;
        }
    """)
    layout.addWidget(toolbar)
    layout.addWidget(canvas)
    dialog.setLayout(layout)
    dialog.exec()
def toggle_dark_mode(self):
    """Toggles between dark mode and light mode"""
    self.dark_mode = not self.dark_mode
    self.set_dark_mode_styles()
def upload_images(self):
    file_paths, _ = QFileDialog.getOpenFileNames(self, "Select Images", "", "Images (*.png *.jpg *.jpeg)")
    if file_paths:
    try:
        import screeninfo
        screen = screeninfo.get_monitors()[0]
        return screen.width / 1920  # Scale based on 1080p reference
    except:
        return 1  # Default scale for unknown screens
def show_class_distribution(self):
    """Displays a bar chart showing the distribution of detected traffic signs with UI scaling."""
    if not self.predictions:
        QMessageBox.warning(self, "No Data", "No classification results available.")
```

```
            return
        # Count occurrences of each traffic sign class
        class_counts = {}
        for _, (label, _) in self.predictions.items():
            class_counts[label] = class_counts.get(label, 0) + 1
        if not class_counts:  #  Prevents division by zero
            QMessageBox.warning(self, "No Data", "No valid traffic sign classifications found.")
            return
        #  Apply Matplotlib UI Scaling
        fig, ax = plt.subplots(figsize=(8, 5))
        ax.bar(class_counts.keys(), class_counts.values(), color="skyblue")
        #  Dynamically Adjust Text Sizes
        dpi_scale = self.get_dpi_scaling()
            ax.set_xticklabels(class_counts.keys(), rotation=45, ha="right", fontsize=11 * dpi_scale)
        ax.grid(axis="y", linestyle="--", alpha=0.7)
        self.show_matplotlib_figure(fig)  #  Fix plt.show()
    def show_accuracy_trend(self):
        """Displays a bar chart of prediction accuracy trends."""
        if not self.predictions:
            QMessageBox.warning(self, "No Data", "No classification results available.")
            return
        image_files = []
        accuracies = []
for img_path, (_, accuracy) in self.predictions.items():
            if accuracy is not None and 0 <= accuracy <= 100:  #  Prevent invalid values
                image_files.append(os.path.basename(img_path))
                accuracies.append(accuracy)
```

```
        if not accuracies:  #  Prevent empty plots
            QMessageBox.warning(self, "No Data", "No valid accuracy data available.")
            return
        fig, ax = plt.subplots(figsize=(8, 5))
        ax.bar(image_files, accuracies, color="green")
        ax.set_xticklabels(image_files, rotation=45, ha="right")
        ax.set_title("Prediction Accuracy Trend")
        ax.set_xlabel("Image File")
        ax.set_ylabel("Accuracy (%)")
        ax.set_ylim(0, 100)
        ax.grid(axis="y", linestyle="--", alpha=0.7)
        self.show_matplotlib_figure(fig)  #  Fix plt.show()
    from PyQt6.QtWidgets import QSizePolicy, QGridLayout
    def display_predictions(self):
        """Dynamically arranges images in a grid while ensuring previous results are properly cleared."""
        #  Properly Clear Previous Layout
        while self.results_layout.count():
            item = self.results_layout.takeAt(0)
            if item.widget():
                item.widget().deleteLater()  #  Proper widget deletion
        if not self.image_paths:
            return  # No images, so exit
        num_columns = max(1, self.width() // 300)
        row, col = 0, 0

        for image_path in self.image_paths:
            prediction_widget = self.create_prediction_widget(image_path)
            col += 1
```

```
            if col >= num_columns:
                col = 0
                row += 1
        self.results_container.adjustSize()  # Ensure container resizes
        self.scroll_area.setWidget(self.results_container)
    def create_prediction_widget(self, image_path):
        """Creates a clean, professional UI block for each image prediction."""
        frame = QFrame(self)
        frame.setFrameShape(QFrame.Shape.StyledPanel)
        frame.setStyleSheet("""
            QFrame {
                border: 2px solid #1E88E5;
                border-radius: 10px;
                padding: 10px;
                background-color: #1A1A1A;
            }
        """ if self.dark_mode else """
            QFrame {
                border: 2px solid #0D47A1;
                border-radius: 10px;
                padding: 10px;
                background-color: white;
            }
        """)
    layout = QHBoxLayout(frame)
        # Image Box (Centered)
        image_label = QLabel(frame)
                                pixmap = QPixmap(image_path).scaled(200, 200,
Qt.AspectRatioMode.KeepAspectRatio, Qt.TransformationMode.SmoothTransformation)
```

```
        image_label.setPixmap(pixmap)
        image_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        # Process Image
        image = Image.open(image_path).convert("RGB")
        image = transform(image).unsqueeze(0)
        output = model(image)
        # Prediction & Accuracy Labels (Updated color)
        text_color = "white" if self.dark_mode else "#0D47A1"  # Dark blue for light mode
        prediction_label = QLabel(f"Prediction: {predicted_text}", frame)
            prediction_label.setStyleSheet(f"font-size: 18px; font-weight: bold; color: {text_color};")
        accuracy_label = QLabel(f"Accuracy: {accuracy_percentage:.2f}%", frame)
        accuracy_label.setStyleSheet("font-size: 16px; color: #FBC02D;")
        accuracy_bar.setDarkMode(self.dark_mode)
        # Arrange Widgets
        text_layout = QVBoxLayout()
        text_layout.addWidget(prediction_label)
        layout.addWidget(image_label, alignment=Qt.AlignmentFlag.AlignCenter)
        layout.addLayout(text_layout)
            self.speak(f"Prediction: {predicted_text}. Accuracy: {accuracy_percentage:.2f} percent")
        self.predictions[image_path] = (predicted_text, accuracy_percentage)
        return frame
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = TrafficSignClassifier()
    window.show()
    sys.exit(app.exec())
```

CONVERT TO YOLO

```
import os
import pandas as pd
from PIL import Image
from shutil import copyfile
# Paths
base_dir = r"C:\Users\dhana\OneDrive\Desktop\Final year project\dataset"
output_dir = r"C:\Users\dhana\OneDrive\Desktop\Final year project\yolo_dataset"
for split in ["Train", "Test"]:  # You can add "Meta" if needed
    csv_path = os.path.join(base_dir, f"{split}.csv")
    df = pd.read_csv(csv_path)
    for i, row in df.iterrows():
        img_path = os.path.join(base_dir, row["Path"])
        label_path = os.path.splitext(os.path.basename(img_path))[0] + ".txt"
        # Create output folders
        img_out_dir = os.path.join(output_dir, "images", split.lower())
        label_out_dir = os.path.join(output_dir, "labels", split.lower())
        os.makedirs(img_out_dir, exist_ok=True)
        os.makedirs(label_out_dir, exist_ok=True)
        # Copy image
        img_out_path = os.path.join(img_out_dir, os.path.basename(img_path))
        copyfile(img_path, img_out_path)
        # Get image size
        width, height = row["Width"], row["Height"]
        # Get normalized bounding box
        x1, y1, x2, y2 = row["Roi.X1"], row["Roi.Y1"], row["Roi.X2"], row["Roi.Y2"]
        box_width = (x2 - x1) / width
        box_height = (y2 - y1) / height
        class_id = int(row["ClassId"])
```

```
        # Save label
        label_out_path = os.path.join(label_out_dir, label_path)
        with open(label_out_path, "w") as f:
                        f.write(f"{class_id}    {x_center:.6f}    {y_center:.6f}    {box_width:.6f}
{box_height:.6f}\n")
print("✅ Conversion complete!")
```

REAL TIME CLASSIFICATION (FOR LAPTOP CAM)

```
import os
import sys
import time
import cv2
import torch
import numpy as np
import csv
from datetime import datetime
from PIL import Image
from torchvision import transforms
from collections import deque, Counter
import pygetwindow as gw
# Set YOLOv5 path
YOLOV5_PATH = os.path.join(os.path.dirname(__file__), "yolov5")
if YOLOV5_PATH not in sys.path:
    sys.path.insert(0, YOLOV5_PATH)
from mamba_model import MambaTransformerClassifier
from yolov5.utils.general import non_max_suppression, scale_boxes
from yolov5.models.common import DetectMultiBackend
```

```
# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
half_precision = device.type == "cuda"
# Load YOLOv5 model
yolo = DetectMultiBackend(
                r"C:\Users\dhana\OneDrive\Desktop\Final year project\yolov5-master\runs\train\traffic_signs\weights\best.pt",
   device=device,
   dnn=False
)
stride, names = yolo.stride, yolo.names
img_size = 640
if half_precision:
   yolo.model.half()
# Load Mamba classifier
num_classes = 43
map_location=device), strict=False)
mamba.eval()
if half_precision:
   mamba.half()
# Transform for classifier (must match training preprocessing)
transform = transforms.Compose([
   transforms.Resize((32, 32)),
   transforms.ToTensor(),
   transforms.Normalize((0.5,), (0.5,))
])
# Class labels dictionary
class_labels = {
   0: "Speed Limit 20", 1: "Speed Limit 30", 2: "Speed Limit 50", 3: "Speed Limit 60",
```

```
    4: "Speed Limit 70", 5: "Speed Limit 80", 6: "End Speed Limit 80", 7: "Speed Limit 100",
    8: "Speed Limit 120", 9: "No Overtaking", 10: "No Overtaking by Trucks", 11: "Priority at Next Intersection",
     28: "Children", 29: "Bicycles", 30: "Ice/Snow", 31: "Wild Animals", 32: "End Restriction", 33: "Turn Right",
    34: "Turn Left", 35: "Go Straight", 36: "Go Straight or Right", 37: "Go Straight or Left", 38: "Keep Right",
    39: "Keep Left", 40: "Roundabout", 41: "End of No Overtaking", 42: "End No Overtake Trucks"
}
# Create a directory for saving ROI images that are passed to Mamba
output_dir = "yolo_crops"
os.makedirs(output_dir, exist_ok=True)
# CSV logging
csv_file = open("detections.csv", mode="a", newline="")
csv_writer = csv.writer(csv_file)
def log_detection(label):
    csv_writer.writerow([datetime.now().strftime("%Y-%m-%d %H:%M:%S"), label])
    csv_file.flush()
# Maintain a short-term history of detections
detections_history = deque()
    detections_history.append((current_time, label))
    while detections_history and current_time - detections_history[0][0] > 30:
        detections_history.popleft()
def detect_shape(c):
    # Approximate contour and check the number of vertices
    approx = cv2.approxPolyDP(c, 0.03 * cv2.arcLength(c, True), True)
        return "triangle"
    elif sides == 4:
```

```
        (x, y, w, h) = cv2.boundingRect(approx)
        aspect = float(w) / h
        return "rhombus" if 0.85 < aspect < 1.15 else "rectangle"
    elif sides == 8:
        return "octagon"
    else:
def draw_top_predictions(img, top_preds):
    """
    Draws only the predictions with high confidence (> 0.9).
    Also, highlights only the top-1 prediction.
    """    x, y = 10, 60
    for idx, (label, prob) in enumerate(top_preds):
        if prob < 0.9:
        cv2.putText(img, text, (x, y), cv2.FONT_HERSHEY_SIMPLEX,
                    0.8, (255, 255, 0), 2)
        y += 30
    # Optionally, highlight only the top-1 prediction
    if top_preds and top_preds[0][1] > 0.9:

        cv2.putText(img, f"{label} ({int(prob * 100)}%)", (10, 60),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
def process_frame(frame):
    # Brighten/normalize the frame for improved contrast/visibility.
    frame = cv2.convertScaleAbs(frame, alpha=1.2, beta=30)
    orig = frame.copy()
    # Prepare frame for YOLO
     img_resized = cv2.resize(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB), (img_size,
img_size))
```

```
                img_tensor = torch.from_numpy(img_resized).permute(2, 0, 1).float().div(255).unsqueeze(0).to(device)
    if half_precision:
        img_tensor = img_tensor.half()
 det = non_max_suppression(pred, 0.25, 0.45, agnostic=False)[0]
    top_preds = []  # List to hold classifier predictions
    if det is not None and len(det):
            if conf < 0.8:
                continue
            x1, y1, x2, y2 = map(int, xyxy)
            roi = orig[y1:y2, x1:x2]
            if roi.shape[0] < 20 or roi.shape[1] < 20:
                continue
            try:
                crop_resized = cv2.resize(roi, (32, 32))
                # Save the cropped ROI image to the output folder
                                                        save_filename = f"roi_{datetime.now().strftime('%Y%m%d_%H%M%S_%f')}.jpg"
                save_path = os.path.join(output_dir, save_filename)
                edges = cv2.Canny(cv2.GaussianBlur(gray, (3, 3), 0), 50, 150)
                                contours, _ = cv2.findContours(edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
                contours = sorted(contours, key=cv2.contourArea, reverse=True)[:3]
                # Only consider the ROI if it has an expected traffic sign shape
                  if any(detect_shape(c) in ["circle", "triangle", "octagon", "rhombus"] for c in contours):
                    # Preprocess ROI exactly as in training
                                        pil_img = Image.fromarray(cv2.cvtColor(crop_resized, cv2.COLOR_BGR2RGB))
                    tensor = transform(pil_img).unsqueeze(0).to(device)
```

```
                if half_precision:
                    tensor = tensor.half()
                with torch.no_grad():
                    out = mamba(tensor)
                    # Only use predictions with high confidence (> 0.8)
                    if conf_val.item() > 0.8:
                        # Optionally, draw the bounding box on the frame
                        cv2.rectangle(orig, (x1, y1), (x2, y2), (0, 255, 0), 3)
            except Exception as e:
                print(f"[!] Mamba Classification Error: {e}")
    # If predictions exist, sort them by confidence (highest first)
    if top_preds:
        # Highlight only the top-1 prediction.
        draw_top_predictions(orig, top_preds
    return orig
def draw_top_detections(frame):
    counter = Counter([label for _, label in detections_history])
    x, y = 10, 100
        cv2.putText(frame, text, (x, y + idx * 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
(255, 50, 50), 2)
def main():
        print("❌ Cannot access camera")
        return
 fps = 0
    prev_time = time.time()
    try:
        while True:
            ret, frame = cap.read()
            if not ret:
```

```
            break
        processed = process_frame(frame)
        draw_top_detections(processed)  # Optional: shows detections history
        curr_time = time.time()
        fps = 0.9 * fps + 0.1 * (1 / (curr_time - prev_time))
        prev_time = curr_time
                                cv2.putText(processed, f"FPS: {fps:.2f}", (10, 35),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)
        cv2.imshow("YOLO + Mamba Real-Time Traffic Sign Detection", processed)
        # Exit on ESC key or if the window is closed
         if cv2.waitKey(1) & 0xFF == 27 or cv2.getWindowProperty("YOLO + Mamba
Real-Time Traffic Sign Detection", cv2.WND_PROP_VISIBLE) < 1:
            break
    finally:
        cap.release()
        cv2.destroyAllWindows()
        csv_file.close()
        # Auto-restart app if window was closed
        window_titles = [w.title for w in gw.getWindowsWithTitle("YOLO + Mamba Real-
Time Traffic Sign Detection")]
        if not window_titles:
            print("🔄 Restarting application...")
            python = sys.executable
            os.execv(python, [python] + sys.argv)


if __name__ == "__main__":
    main()
```

REAL TIME CLASSIFICATION (FOR MOBILE CAM)

```
import os
import sys
# Set YOLOv5 path
YOLOV5_PATH = os.path.join(os.path.dirname(__file__), "yolov5")
if YOLOV5_PATH not in sys.path:
    sys.path.insert(0, YOLOV5_PATH)
import cv2
import torch
import numpy as np
import csv
import time
from datetime import datetime
from PIL import Image
from torchvision import transforms
from mamba_model import MambaTransformerClassifier
from yolov5.utils.general import non_max_suppression, scale_boxes
from yolov5.models.common import DetectMultiBackend

# ========================= Setup
=========================
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
half_precision = device.type == "cuda"
# Load YOLOv5 model
# YOLOv5 Model Load
# yolo = DetectMultiBackend("yolov5s.pt", device=device, dnn=False)
changed(5/7/2025)
stride, names = yolo.stride, yolo.names
img_size = 640
if half_precision:
    yolo.model.half()

# Mamba Classifier Load
num_classes = 43
mamba = MambaTransformerClassifier(num_classes).to(device)
mamba.eval()
if half_precision:
    mamba.half()

# Image transform (optimized to use torchvision transforms efficiently)
transform = transforms.Compose([
```

```
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Class labels

class_labels = {
    0: "Speed Limit 20", 1: "Speed Limit 30", 2: "Speed Limit 50", 3: "Speed Limit 60",
    4: "Speed Limit 70", 5: "Speed Limit 80", 6: "End Speed Limit 80", 7: "Speed Limit
100",
    8: "Speed Limit 120", 9: "No Overtaking", 10: "No Overtaking by Trucks", 11:
"Priority at Next Intersection",
    12: "Priority Road", 13: "Yield", 14: "Stop", 15: "No Vehicles", 16: "No Trucks", 17:
"No Entry",
    34: "Turn Left", 35: "Go Straight", 36: "Go Straight or Right", 37: "Go Straight or
Left", 38: "Keep Right",
    39: "Keep Left", 40: "Roundabout", 41: "End of No Overtaking", 42: "End No
Overtake Trucks"
}
# CSV writer setup
csv_file = open("detections.csv", mode="a", newline="")
csv_writer = csv.writer(csv_file)

def log_detection(label):
    csv_writer.writerow([datetime.now().strftime("%Y-%m-%d %H:%M:%S"), label])
    csv_file.flush()

from collections import deque, Counter

# Store detections with timestamps (deque for performance)
detections_history = deque()

    # Remove old entries (older than 30 seconds)
    while detections_history and current_time - detections_history[0][0] > 30:
        detections_history.popleft()

# ==================== Helper Functions ====================

def detect_shape(c):
    approx = cv2.approxPolyDP(c, 0.03 * cv2.arcLength(c, True), True)
    sides = len(approx)
```

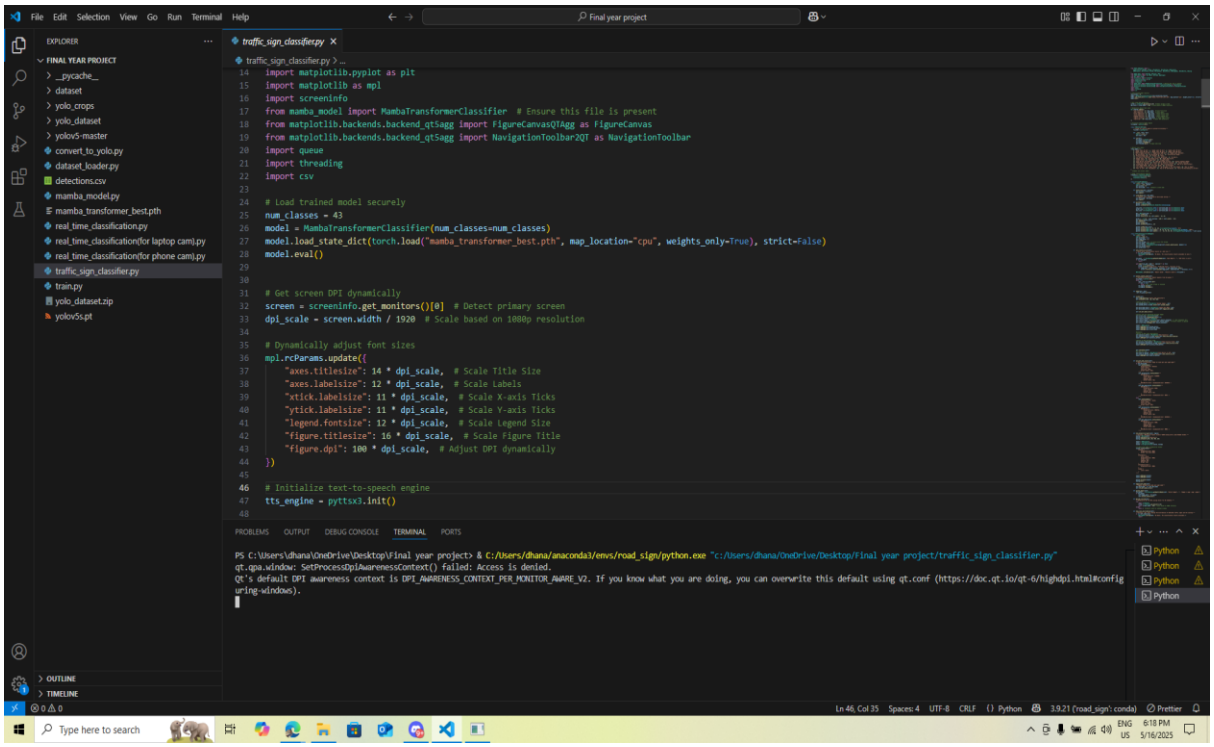
```
    if sides == 3:
        return "triangle"
        return "rhombus" if 0.85 < aspect < 1.15 else "rectangle"
    elif sides == 8:
        return "octagon"
    else:
        area = cv2.contourArea(c)
        peri = cv2.arcLength(c, True)
        if peri == 0:
            return None
        circularity = 4 * np.pi * area / (peri ** 2)
        return "circle" if circularity > 0.75 else None
def save_to_csv(label):
    csv_writer.writerow([datetime.now().strftime("%Y-%m-%d %H:%M:%S"), label])
    csv_file.flush()
    with torch.no_grad():
        pred = yolo(img_tensor)
        det = non_max_suppression(pred, 0.25, 0.45, agnostic=False)[0]
    if det is not None and len(det):
                continue
            x1, y1, x2, y2 = map(int, xyxy)
            roi = orig[y1:y2, x1:x2]
            if roi.shape[0] < 20 or roi.shape[1] < 20:
                continue
        try:
                crop_resized = cv2.resize(roi, (32, 32))
                gray = cv2.cvtColor(crop_resized, cv2.COLOR_BGR2GRAY)
                edges = cv2.Canny(cv2.GaussianBlur(gray, (3, 3), 0), 50, 150)
                if any(detect_shape(c) in ["circle", "triangle", "octagon", "rhombus"] for c in
contours):
                    pil_img = Image.fromarray(cv2.cvtColor(crop_resized,
cv2.COLOR_BGR2RGB))
                    tensor = transform(pil_img).unsqueeze(0).to(device)
                    if half_precision:
                        tensor = tensor.half()
                    with torch.no_grad():
                        out = mamba(tensor)
                        prob = torch.nn.functional.softmax(out, dim=1)
                            label = class_labels.get(label_idx.item(), "Unknown")
                            log_detection(label)
                            update_detections(label)
                            cv2.rectangle(orig, (x1, y1), (x2, y2), (0, 255, 0), 3)
```

```
            print(f"[!] Mamba Classification Error: {e}")
    return orig

# ==================== Main Loop ====================

def main():
    cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)
    if not cap.isOpened():
        print("❌ Cannot access camera")
        return
    fps = 0
    prev_time = time.time()
    try:
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            cv2.putText(processed, f"FPS: {fps:.2f}", (10, 35),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)
            cv2.imshow("YOLO + Mamba Real-Time Traffic Sign Detection", processed)

            if cv2.waitKey(1) & 0xFF == 27 or cv2.getWindowProperty("YOLO + Mamba
Real-Time Traffic Sign Detection", cv2.WND_PROP_VISIBLE) < 1:
                break
    finally:
        cap.release()
        cv2.destroyAllWindows()
        csv_file.close()
def draw_top_detections(frame):
    # Count frequency
    counter = Counter([label for _, label in detections_history])
    top = counter.most_common(5)  # Show top 5 signs

    x, y = 10, 60  # Starting position
    for idx, (label, count) in enumerate(top):
        text = f"{idx+1}. {label} ({count})"
        cv2.putText(frame, text, (x, y + idx * 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
(255, 50, 50), 2)

if __name__ == "__main__":
    main()
```

APPENDICES – II

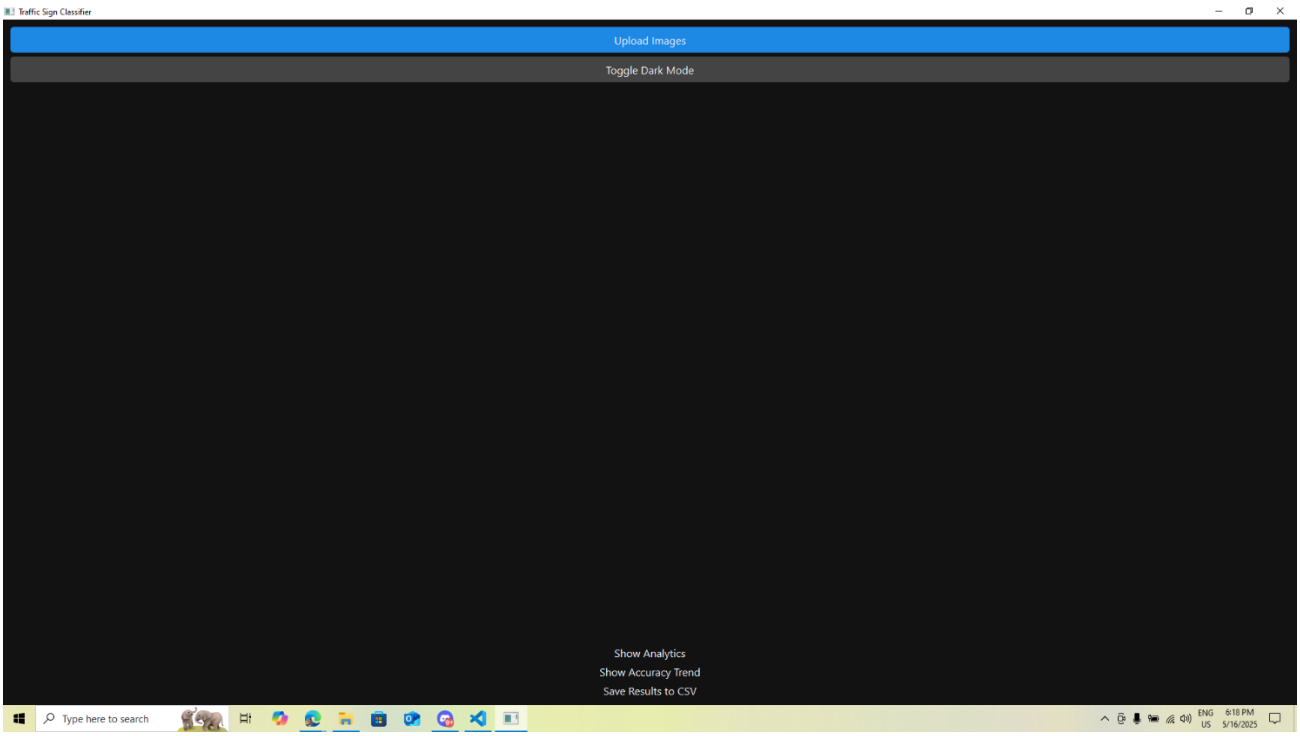
SNAPSHOTS

1 INITIALIZATION OF TRAFFIC SIGN CLASSIFIER



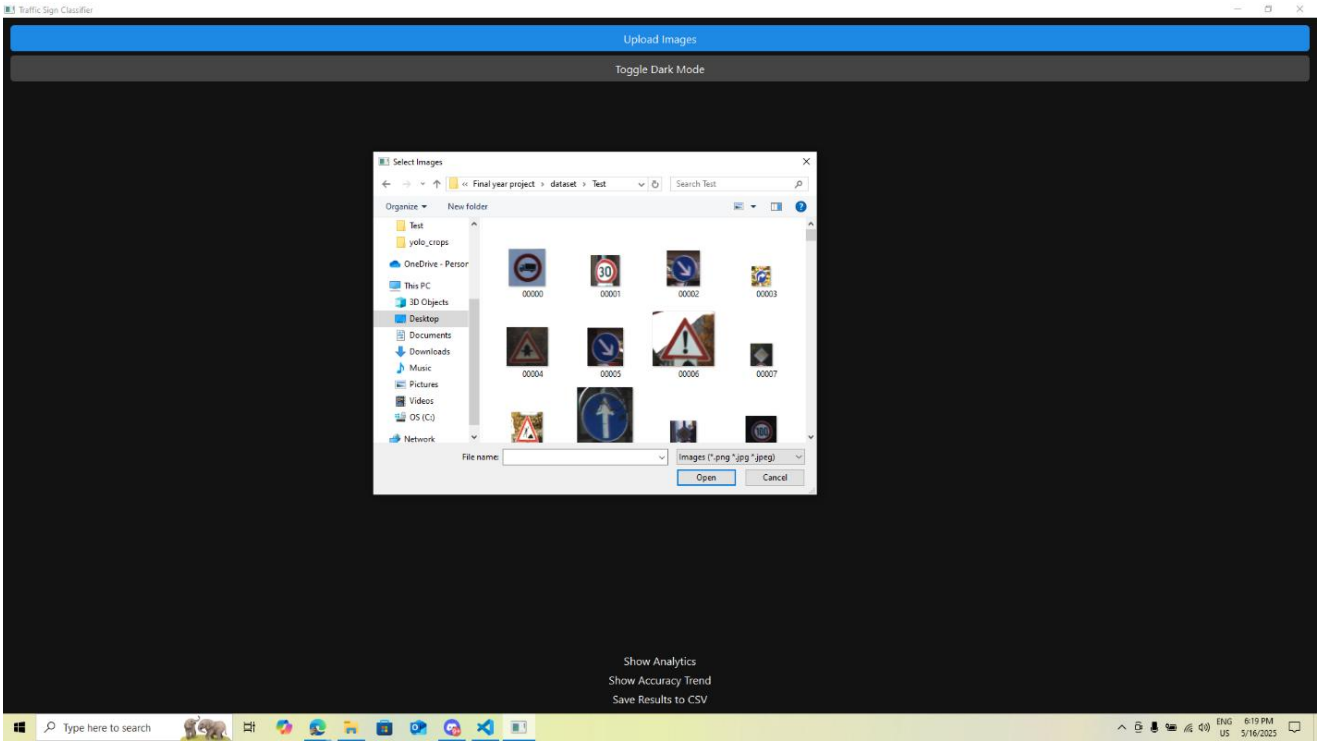
LABEL 1: INITIALIZATION OF TRAFFIC SIGN CLASSIFIER

1.1 INDEX OF TRAFFIC SIGN CLASSIFIER



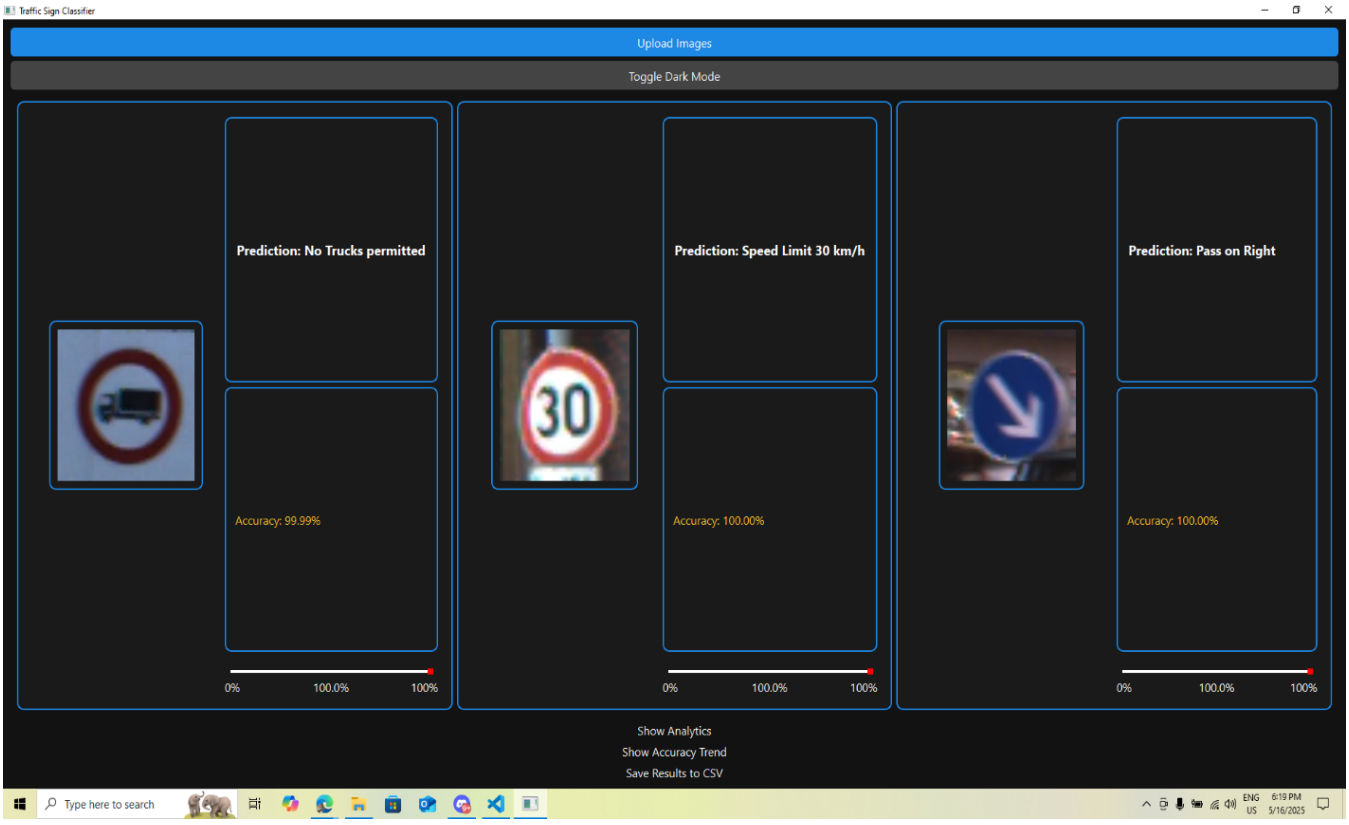
LABEL 2: INDEX OF TRAFFIC SIGN CLASSIFIER

1.2 IMAGE UPLOADATION



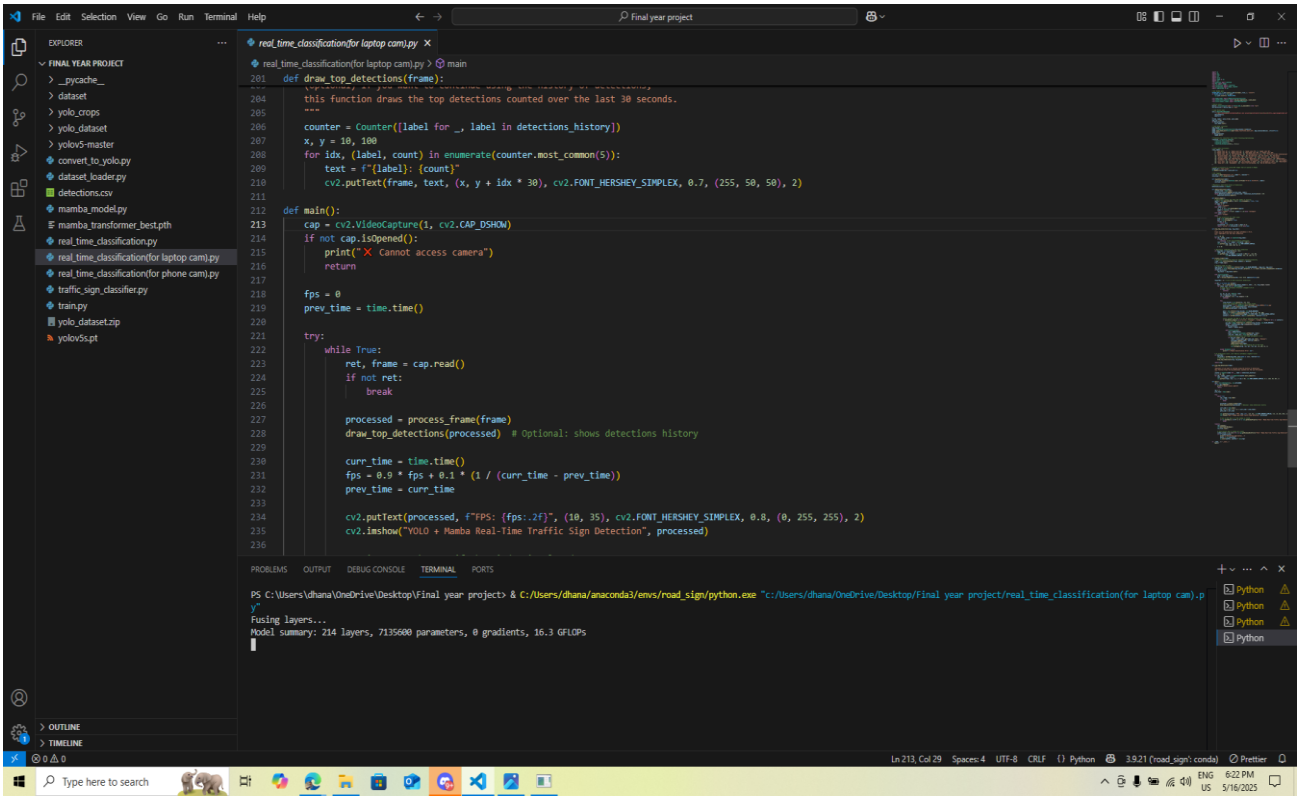
LABEL 3: IMAGE UPLOADATION

1.3 OUTPUT OF TRAFFIC SIGN CLASSIFIER



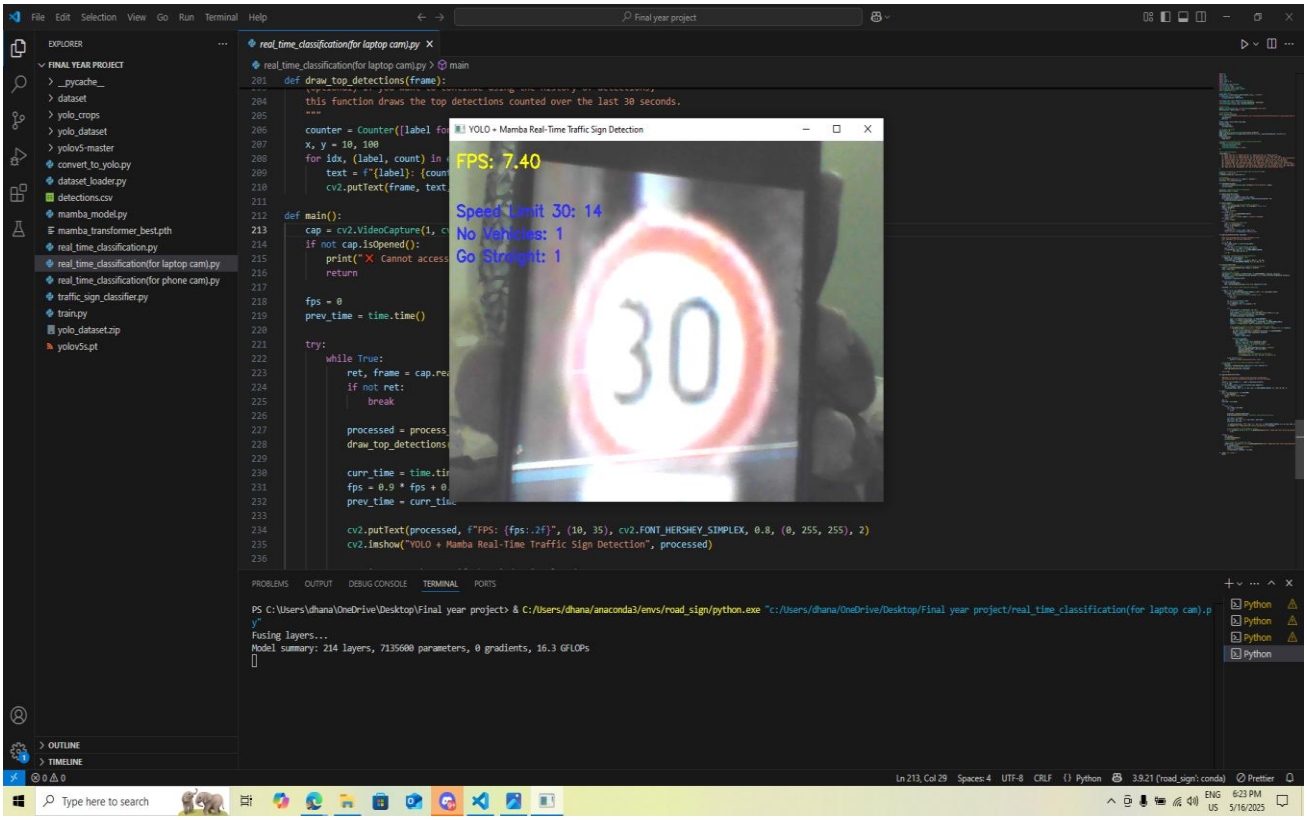
LABEL 4: OUTPUT OF TRAFFIC SIGN CLASSIFIER

2 INITIALIZATION OF REAL TIME CLASSIFICATION LAPTOP CAM



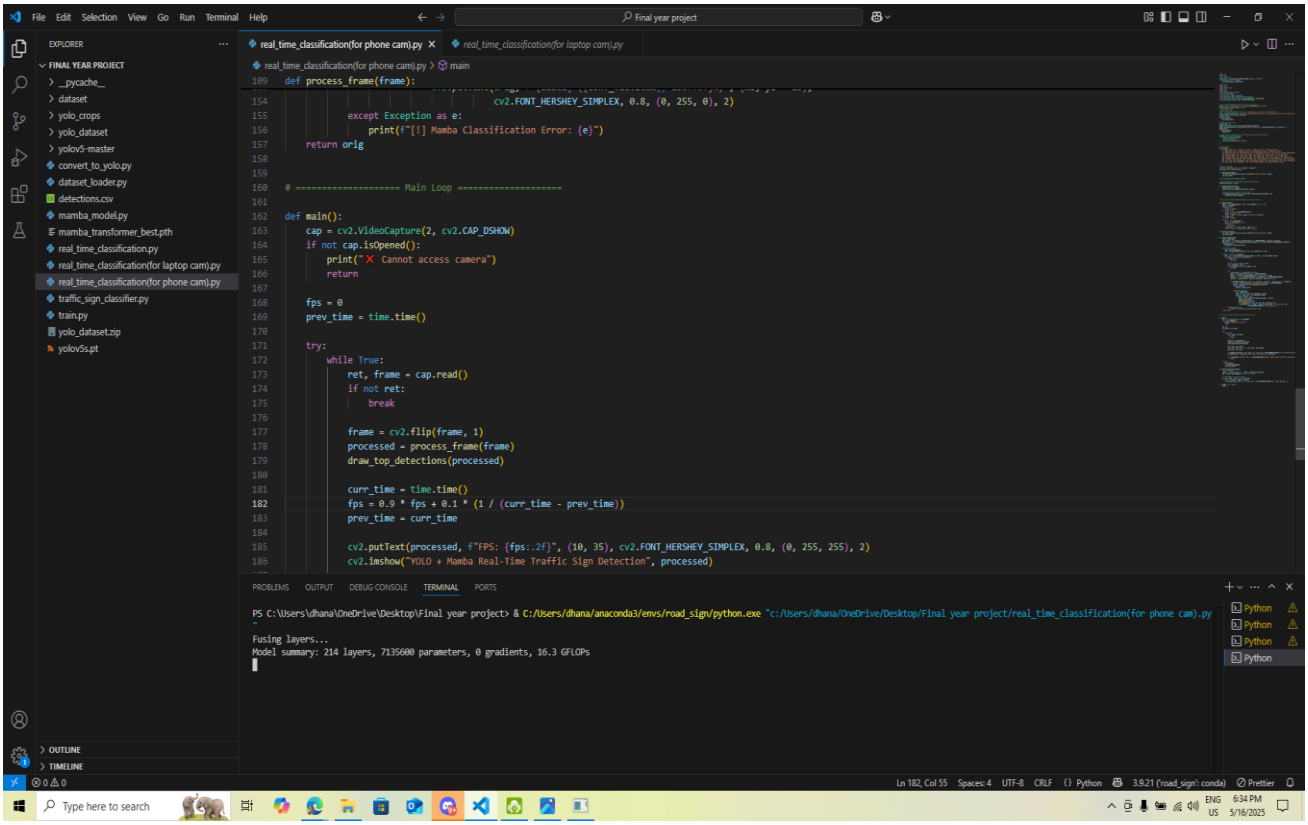
LABEL 5: INITIALIZATION OF REAL TIME CLASSIFICATION LAPTOP CAM

2.1 OUTPUT OF REALTIME CLASSIFICATION LAPTOP CAM



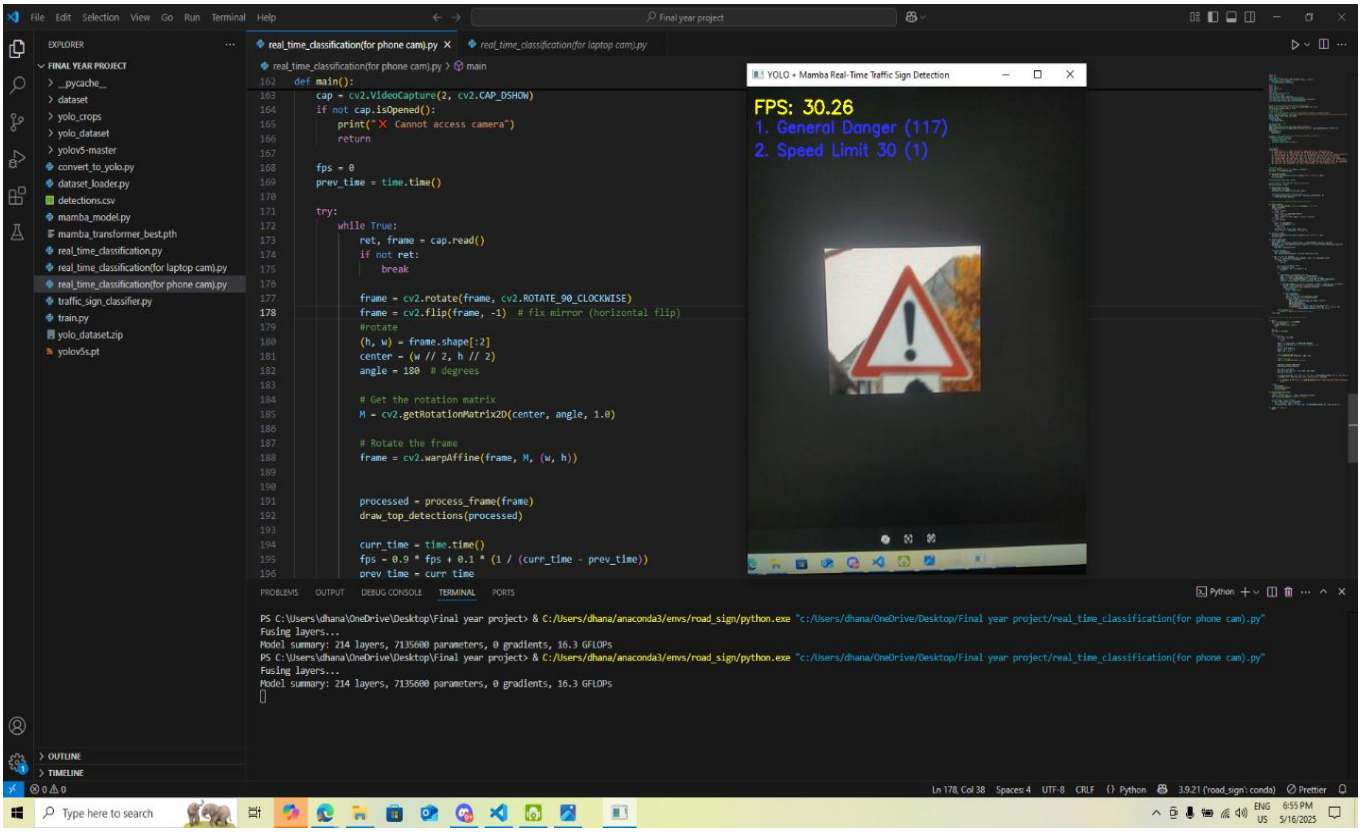
LABEL 6: OUTPUT OF REALTIME CLASSIFICATION LAPTOP CAM

3 INITIALIZATION OF REAL TIME CLASSIFICATION MOBILE CAM



LABEL 7: INITIALIZATION OF REAL TIME CLASSIFICATION MOBILE CAM

3.1 OUTPUT OF REAL TIME CLASSIFICATION MOBILE CAM



LABEL 8: OUTPUT OF REAL TIME CLASSIFICATION MOBILE CAM

REFERENCES

[1] G. Zeng, Z. Wu, L. Xu, and Y. Liang, "Efficient Vision Transformer YOLOv5 for Accurate and Fast Traffic Sign Detection," *Electronics*, vol. 13, no. 5, p. 880, Feb. 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/5/880MDPI+1ResearchGate+1

[2] T. Yu, "MDDFNet: Mamba-based Dynamic Dual Fusion Network for Traffic Sign Detection," *arXiv preprint arXiv:2505.05491*, May 2025. [Online]. Available: https://arxiv.org/abs/2505.05491arXiv

[3] S. Wu, X. Lu, and C. Guo, "YOLOv5_mamba: Unmanned Aerial Vehicle Object Detection Based on Bidirectional Dense Feedback Network and Adaptive Gate Feature Fusion," *Scientific Reports*, vol. 14, Article no. 22396, Sep. 2024. [Online]. Available: https://www.nature.com/articles/s41598-024-73241-xNature

[4] J. Wang, Y. Chen, M. Gao, and Z. Dong, "Improved YOLOv5 Network for Real-Time Multi-Scale Traffic Sign Detection," *arXiv preprint arXiv:2112.08782*, Dec. 2021. [Online]. Available: https://arxiv.org/abs/2112.08782arXiv

[5] J. Chen, H. Huang, R. Zhang, N. Lyu, Y. Guo, H.-N. Dai, and H. Yan, "YOLO-TS: Real-Time Traffic Sign Detection with Enhanced Accuracy Using Optimized Receptive Fields and Anchor-Free Fusion," *arXiv preprint arXiv:2410.17144*, Oct. 2024. [Online]. Available: https://arxiv.org/abs/2410.17144arXiv

[6] O. N. Manzari, A. Boudesh, and S. B. Shokouhi, "Pyramid Transformer for Traffic Sign Detection," *arXiv preprint arXiv:2207.06067*, Jul. 2022. [Online]. Available: https://arxiv.org/abs/2207.06067arXiv

[7] R.-C. Chen, S.-K. Tai, J.-C. Chen, and I. Y. Garta, "Improved Detection of Bad Traffic Signs Using Non-Maximum Suppression Ensemble," in *Proc. 2024 TAAI Conference*, 2024, pp. 1–6. [Online]. Available: https://www.taai.org.tw/files/domestic/55/PDF_final%20copy.pdftaai.org.tw

[8] Y. Zhang, P. Wang, X. Liu, Y. Yuan, and L. Wang, "MambaTSR: You Only Need 90k Parameters for Traffic Sign Recognition," *Neurocomputing*, vol. 545, p. 128104, 2024. [Online]. Available: https://doi.org/10.1016/j.neucom.2024.128104

[9] Y. Wang, H. Li, and X. Chen, "Traffic Sign Detection Based on the Improved YOLOv5," *Applied Sciences*, vol. 13, no. 17, p. 9748, 2023. [Online]. Available: https://doi.org/10.3390/app13179748

[10] J. Li and Q. Zhao, "YOLOv5-TS: Detecting Traffic Signs in Real-Time," *Frontiers in Physics*, vol. 11, p. 1297828, 2023. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fphy.2023.1297828/full

[11] YOLOv5 Masters Github: https://github.com/ultralytics/yolov5.git