TUNISIAN REPUBLIC
Ministry of Higher Education and Scientific Research

University of Carthage
Department of Engineering Physics and Instrumentation

Personal Professional Project

AI-Powered Road Sign Recognition for Driver Assistance and Autonomous Navigation

Realised by:

Abbes Youssef

Zidane Mohamed

Zghibi Ahmed

Znagui Wael

Supervisor:

Mme Mastouri Rekka

Specialization:

3rd year industrial computing and automation

Defended on 03/06/2025

Table of contents

1. Introduction

Road safety remains a pressing global challenge, even as technological advancements in the automotive sector continue to evolve. A significant portion of traffic accidents result from human error, particularly the failure to correctly interpret or react to road signs. According to Tunisia's National Road Safety Observatory, the first five months of 2025 alone saw 411 fatalities and 2,390 injuries caused by road accidents. These alarming figures highlight the urgent need for driver assistance technologies that support human decision-making and reduce avoidable risks.

This project presents the design and implementation of a low-cost, AI-powered driver assistance system capable of detecting road signs—especially speed limits—and comparing them with the actual speed of the vehicle. If the detected speed exceeds the speed limit, the system provides real-time alerts using a buzzer, LEDs, and a visual display, allowing the driver to take immediate corrective action.

To simulate realistic driving conditions, the system integrates several key components:

- An Arduino Uno simulates the car's Electronic Control Unit (ECU) by sending vehicle speed values via CAN bus, based on input from a potentiometer.
- A Raspberry Pi 4 processes speed data and uses a USB camera to detect traffic signs using a YOLOv11s object detection model.
- A buzzer, LEDs, and LCD screen provide feedback to the driver based on system decisions.

This project demonstrates how embedded systems, computer vision, and vehicle communication protocols can work together to enhance road safety. It aims to replicate features found in high-end driver assistance systems, but using affordable and open-source technologies. This makes the solution particularly valuable for developing regions like Tunisia, where access to such technologies is limited.

The structure of this report is as follows:

- Section 2 covers the background and key technologies that support our solution, including YOLOv11s, CAN bus, and Raspberry Pi GPIO functionality.

- Section 3 details the hardware architecture, including component roles, wiring, and inter-device communication.
- Section 4 focuses on the AI and software integration, explaining how the system performs detection, speed comparison, and user feedback.
- Section 5 presents the testing methodology and results.
- Sections 6–8 discuss the challenges encountered, potential improvements, and the final conclusions of the project.

By combining intelligent software with modular, cost-effective hardware, this system demonstrates the potential of AI in improving road safety through real-time driver support.

2. Background & Literature Review

The integration of intelligent systems into vehicles has become a crucial area of innovation. Modern automotive technologies are increasingly focused on delivering safety and decision-making support to drivers, especially in regions with high traffic accident rates like Tunisia [1]. In this context, our project combines embedded systems and artificial intelligence to detect road signs and assess driver compliance with speed limits.

2.1. Artificial Intelligence in Driver Assistance

Artificial Intelligence (AI) is at the core of Advanced Driver Assistance Systems (ADAS). ADAS helps drivers interpret traffic signs, anticipate hazards, and maintain safe driving practices. Speed limit recognition is among the most essential ADAS features, as it enables vehicles to alert or even intervene when speeding occurs. With increasing traffic complexity, AI-based systems are proving critical in reducing accident rates and enhancing road safety [1].

2.2. YOLO – Real-Time Object Detection

Our system utilizes **YOLOv11s** (You Only Look Once, version 11 – small), a modern deep learning model designed for real-time object detection [2]. YOLOv11s processes full image frames in a single pass, offering both speed and accuracy while being optimized for edge devices. This architecture is well-suited for limited-resource platforms such as the Raspberry

Pi due to its lightweight design and efficient inference pipeline. In our application, YOLOv11s is used to detect speed limit signs from live video input. Each detected sign is associated with a class ID, which is mapped to a corresponding numerical speed limit using a dictionary within the software. This extracted limit is then used in real-time to evaluate the vehicle's speed and trigger appropriate responses.

2.3. Raspberry Pi as an Embedded AI Hub

The Raspberry Pi 4 plays a central role in our system. It captures camera input, runs the YOLOv11s detection model, receives speed values from the Arduino, and manages the output devices. The Pi handles a buzzer for overspeed alerts, two LEDs for visual status, and an I2C 16x2 LCD display for feedback [5][6][7]. Its GPIO support and compatibility with Python libraries make it a strong platform for embedded AI applications.

2.4. Arduino for Speed Simulation

To simulate vehicle speed without accessing an actual car, we use an Arduino Uno connected to a potentiometer. The potentiometer's analog signal is converted into a digital speed value ranging from 0 to 120 km/h. The Arduino transmits this value to the Raspberry Pi via USB serial communication, providing a continuous and adjustable speed input. This approach is both cost-effective and practical for testing embedded decision-making logic in real-time conditions.

3. Embedded System and Hardware Architecture

The embedded system in our project integrates a Raspberry Pi and an Arduino Uno to simulate and monitor vehicle speed in real time. This setup ensures real-world testing without the need for actual vehicle data, allowing us to focus on speed detection, comparison, and alert mechanisms using accessible and modular components.

3.1. System Overview and Architecture

The system consists of two main parts: an Arduino that simulates the car's speed and a Raspberry Pi that processes this information and runs AI detection. The Arduino is connected to a potentiometer, which serves as a manual input to vary the speed dynamically. This value is sent to the Raspberry Pi via USB serial.

The Raspberry Pi, acting as the main controller, reads the incoming speed data and compares it with the speed limits detected from a live video stream processed by a YOLOv11s object detection model. Based on this comparison, it activates corresponding alert mechanisms like a buzzer, red/green LEDs, and displays the information on an LCD screen. Figure1 shows a simplified representation of the system architecture.

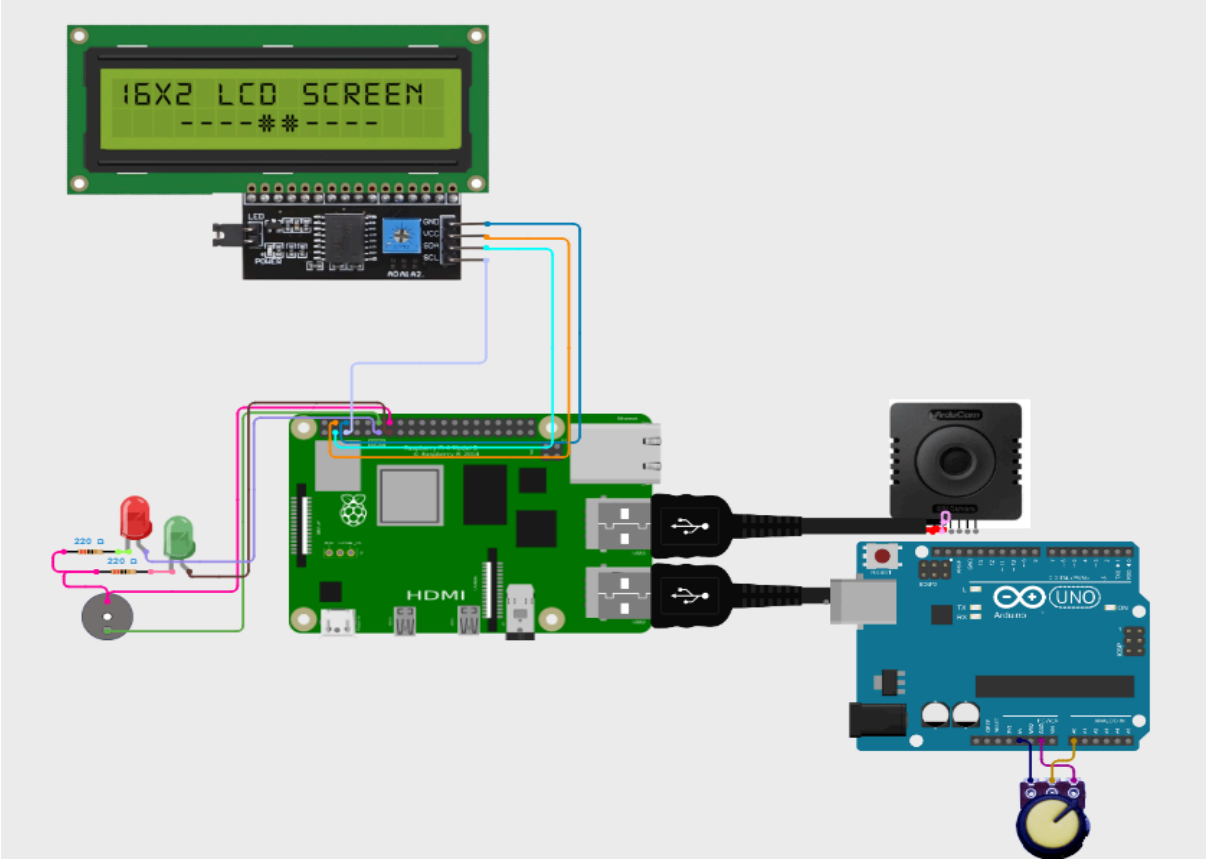

Figure1: *System architecture*

3.2. Arduino-based Speed Simulation

To simulate vehicle speed, a 10kΩ potentiometer is wired to an analog pin (A0) of the Arduino Uno. The analog value (ranging from 0 to 1023) is mapped to a realistic speed range of 0 to 230 km/h. The Arduino continuously sends this value over the serial port using Serial.println().

This approach allowed for precise control of the simulated speed during testing. However, we initially encountered erratic readings due to loose wiring in the breadboard setup.

3.3. Serial Communication and Integration

To transmit the simulated vehicle speed from the Arduino to the Raspberry Pi, we established a USB serial communication link, as it is shown in figure 2. The Arduino sends speed values as plain text over its USB connection using the Serial.println() function. On the Raspberry Pi side, we use the pyserial library to read the data through the corresponding serial port .[8]

This method provides a simple, direct, and reliable way to pass data between the two devices without the complexity of additional protocols or hardware modules. It also allows for easy debugging and testing, as the speed data can be monitored directly from a terminal or log.

Overall, the USB serial method has proven effective for integrating the Arduino and Raspberry Pi into a unified embedded AI system. It keeps the setup minimal and cost-efficient while allowing for real-time speed data processing.

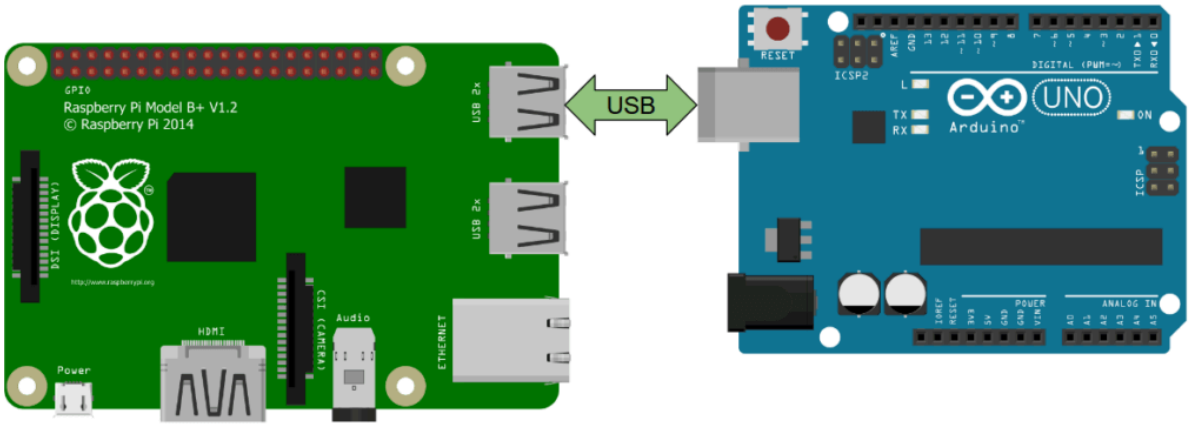

Figure2: *Raspberry Pi/Arduino Serial communication*

3.4. Output Management: Buzzer, LEDs, LCD

The Raspberry Pi controls several output devices via GPIO pins[8] (figure3 showcases The GPIO Pinout Diagram):

- A buzzer (connected to GPIO14) is activated when the current speed exceeds the detected speed limit.
- A red LED (GPIO17) indicates overspeed, while a green LED (GPIO27) indicates safe driving speed.

- An I2C 16x2 LCD screen displays both the current speed and the detected speed limit in real-time.[6]

The buzzer and LEDs are controlled using the `RPi.GPIO` library, while the LCD is managed through the `RPLCD` library using the I2C protocol. These components provide immediate visual and auditory feedback to the user based on system logic.

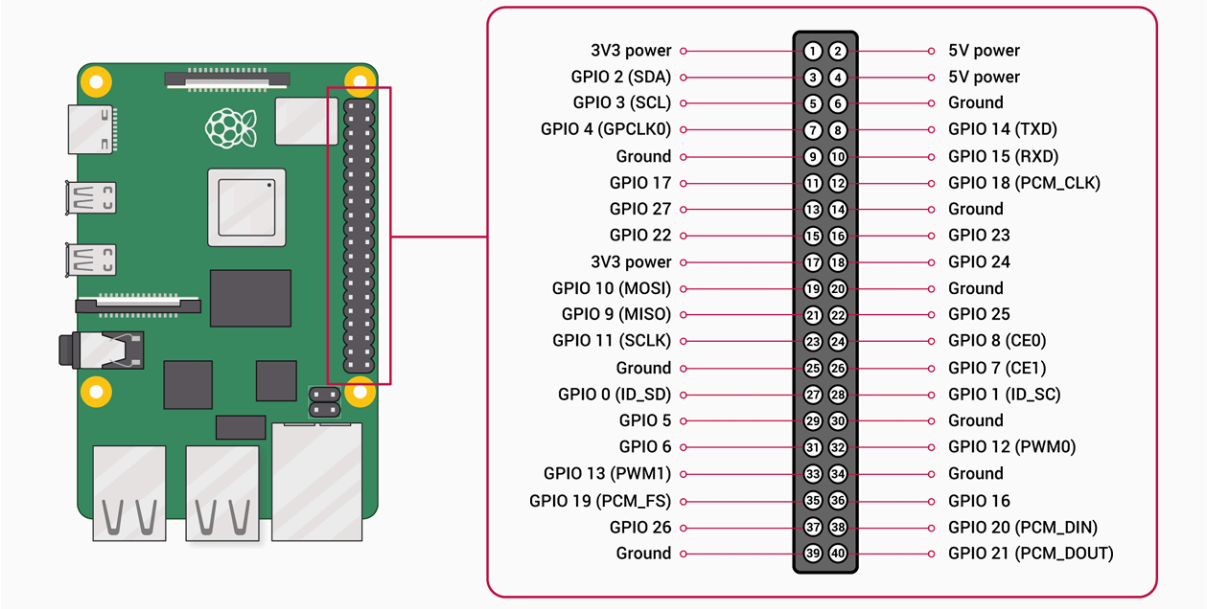

Figure3: *Raspberry PI GPIO Pinout Diagram[4]*

4. AI System & Software Implementation

The intelligence of our driver assistance system lies in its ability to detect speed limit signs using AI and compare them to the current vehicle speed. This section presents the software components that enable these tasks, including the object detection model, the logic for interpreting detections, and the integration of user alerts using GPIO outputs.

4.1. YOLO-Based Road Sign Detection

At the core of our visual recognition system is YOLOv11s (You Only Look Once version 11 – small), a real-time object detection model developed for high-speed, high-accuracy tasks on resource-constrained devices. YOLOv11s processes entire image frames in a single forward pass and returns bounding boxes along with class predictions for detected objects. In our

implementation, the model is trained to specifically identify speed limit signs from a live video stream captured via a USB camera connected to the Raspberry Pi.

We selected the YOLOv11s variant for its efficient performance on edge devices such as the Raspberry Pi 4. It is capable of detecting multiple traffic sign classes, each represented by a numerical class ID. In our software, these IDs are mapped to their corresponding speed limits (e.g., class 0 → 20 km/h, class 1 → 30 km/h, etc.) through a dictionary-based lookup. Once a valid speed sign is detected, the class is extracted and stored as the current speed limit [2].

Detection is performed in real time on image frames captured with the OpenCV library. The output is visualized by drawing bounding boxes and class labels on the video feed, which aids in both debugging and performance evaluation [3].

4.2. Speed Limit Comparison

Once the speed limit is extracted from the YOLO model, it is compared to the simulated vehicle speed received from the Arduino via serial communication. The Raspberry Pi reads the speed as an integer and stores it as the current vehicle speed.

The comparison logic is straightforward:

- If the current speed exceeds the detected limit, the system activates the buzzer and red LED as a warning to the driver.
- If the current speed is within the allowed limit, the green LED is activated, and the buzzer remains off.

In both cases, the system updates an LCD display via the I2C interface to show the current speed and the detected limit in real time. This ensures the driver always has clear, visual feedback about the vehicle's status.

4.3. Full System Operation Flow

The software system runs as a single Python script. The figure 4 illustrates how each component interacts within the system pipeline.:

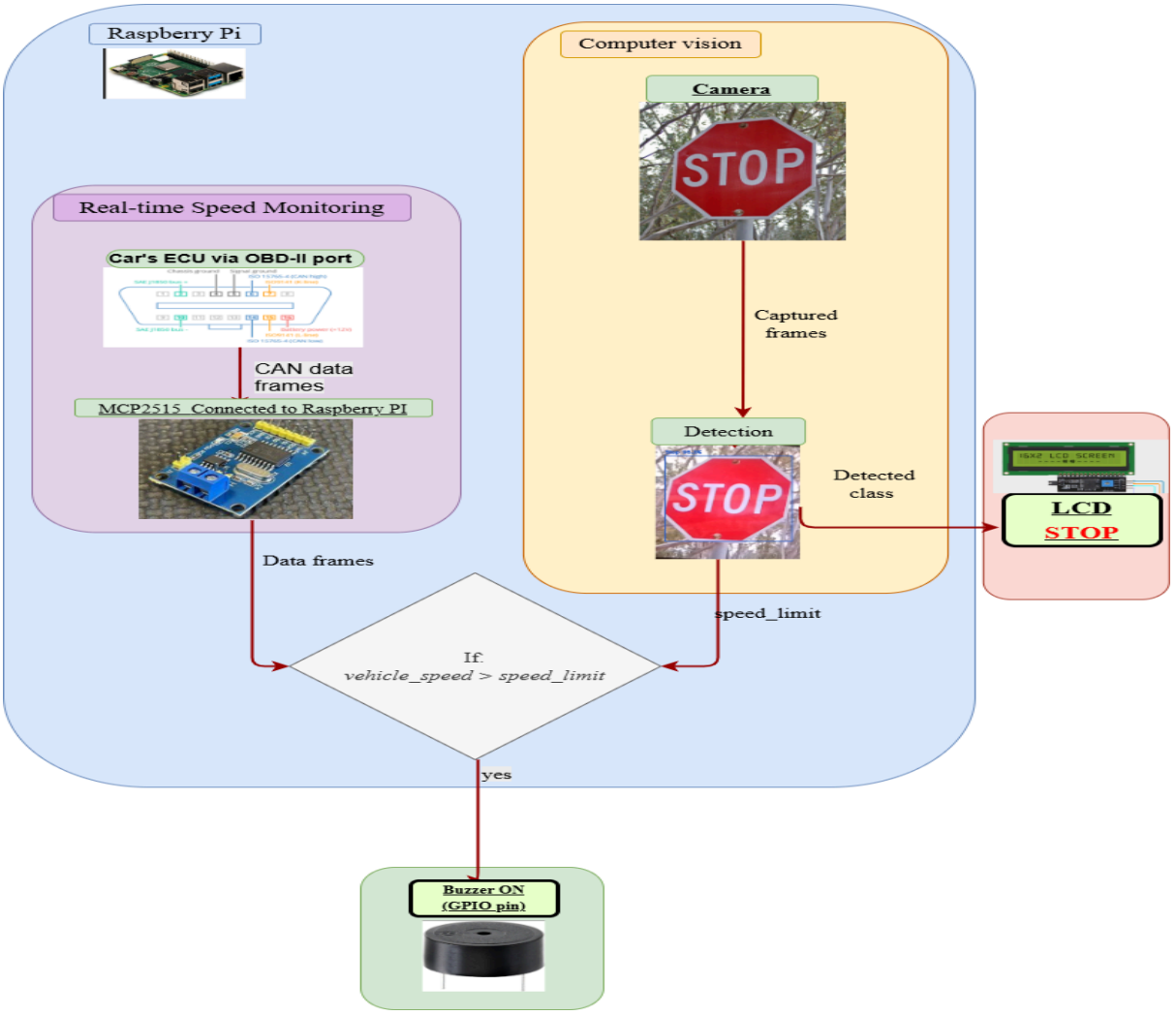

***Figure4**: full system operation flow*

5. AI Model Development and Implementation

The artificial intelligence component of our road sign detection system represents the core computational engine that enables real-time recognition and classification of traffic signs for driver assistance applications. This section presents a comprehensive analysis of our model development process, from dataset preparation through deployment optimization for Raspberry Pi 4 hardware. The implementation leverages the YOLOv11 architecture, chosen for its superior balance between detection accuracy and computational efficiency, making it ideally suited for edge computing applications in automotive environments.

5.1. Dataset Selection and Preprocessing

The foundation of any robust computer vision system lies in the quality and comprehensiveness of its training data. For our road sign detection system, we utilized the German Traffic Sign Recognition Benchmark (GTSRB) dataset, which provides a standardized and internationally recognized collection of traffic sign images[10] captured under diverse real-world conditions.

The GTSRB dataset initially contained 43 distinct traffic sign classes (figure 5), encompassing speed limits, warning signs, mandatory signs, and prohibitory signs.

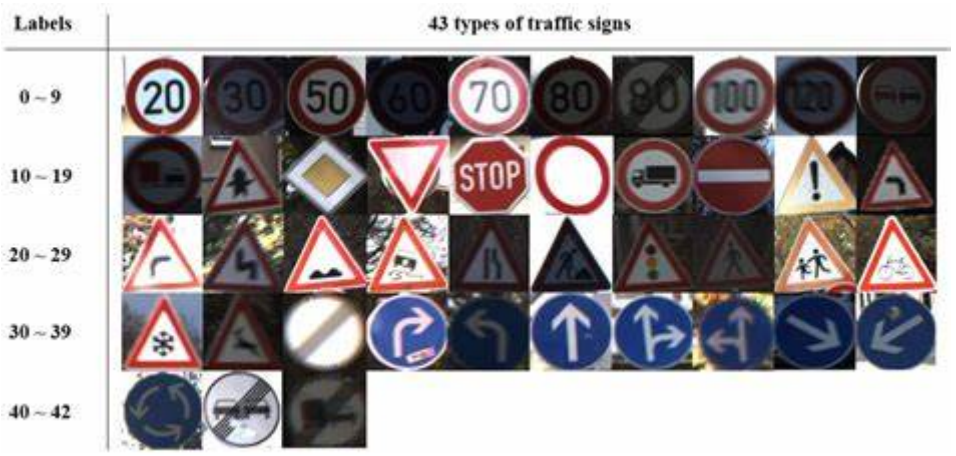

Figure 5: *Forty-three kinds of German traffic signs from GTSDB dataset*

However, for our specific application targeting driver assistance systems, we performed a strategic class reduction to focus on the most critical signs for road safety. Through careful analysis of traffic safety statistics and consultation with automotive safety standards, we eliminated 17 classes that were either region-specific or had minimal impact on driver assistance functionality.

The final dataset comprised 26 essential traffic sign classes, including:

- Speed limit signs (20, 30, 50, 60, 70, 80, 100, 120 km/h)
- Critical warning signs (dangerous curves, road work, pedestrian crossings)
- Mandatory traffic control signs (stop, yield, priority road)

- Directional and navigational signs (turn indicators, roundabout)

5.1.1. Class Distribution Analysis and Balancing

Prior to model training, we conducted a comprehensive analysis of class distribution within our selected dataset. The distribution revealed significant imbalances (figure 6), with some classes containing substantially fewer samples than others, which could lead to biased model performance favoring over-represented classes.

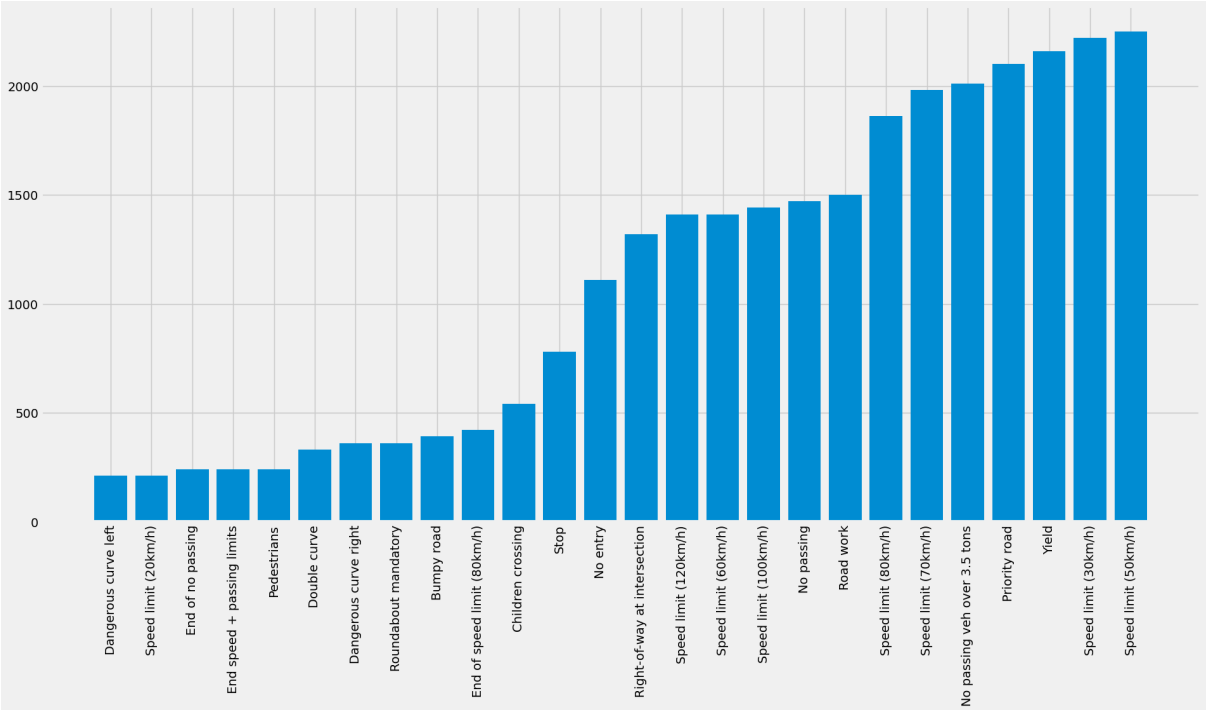

Figure 6: *The distribution of training samples across different traffic sign classes before augmentation (highlighting the class imbalance issue)*

To address this imbalance, we implemented a targeted data augmentation strategy focusing specifically on underrepresented classes. Nine classes were identified as requiring augmentation: classes 0, 19, 41, 32, 27, 21, 20, 22, and 28. These classes represented critical traffic signs such as speed limits, dangerous curves, and pedestrian crossings, making their accurate detection paramount for safety applications. In addition to augmentation, we also deliberately removed samples from overrepresented classes to reduce dataset bias and further balance the distribution of training data across all classes.

Here's the distribution of samples per class after preprocessing

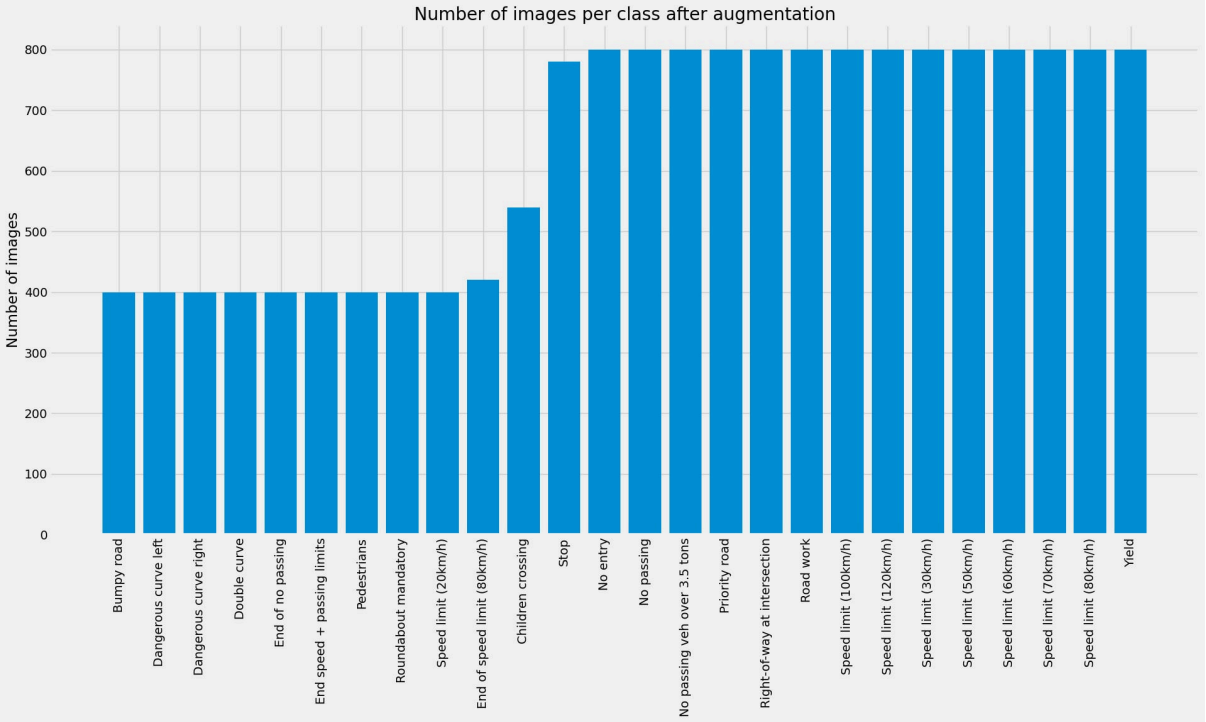

Figure 7: *The distribution of training samples across different traffic sign classes*

5.1.2. Advanced Data Augmentation Pipeline

Our augmentation pipeline utilized the Albumentations library, implementing a sophisticated combination of geometric and photometric transformations[11] designed to simulate real-world variations in traffic sign appearance. The augmentation strategy was carefully calibrated to preserve the semantic integrity of traffic signs while introducing sufficient variability to improve model generalization.

The transformation pipeline included:

Geometric Transformations:

- Rotation: ±20° to simulate camera angle variations
- Horizontal flipping: 50% probability for symmetric signs
- Vertical flipping: 10% probability for specific sign orientations

Photometric Transformations:

- Random brightness and contrast adjustments: ±50% variation
- Hue, saturation, and value modifications: accounting for lighting conditions
- RGB channel shifting: simulating different camera sensors
- Gaussian blur: 30% probability to simulate motion blur and focus variations

The augmentation process was executed in two phases, generating approximately 2,000 additional training samples for underrepresented classes. This strategic approach ensured balanced representation while maintaining the authenticity of traffic sign characteristics essential for reliable detection.

5.2. Model Architecture Selection and Configuration

The selection of an appropriate model architecture represents a critical decision point that directly impacts both detection performance and deployment feasibility on resource-constrained hardware. After comprehensive evaluation of various state-of-the-art object detection frameworks, we selected YOLOv11-small (YOLOv11s) as our base architecture[12].

the figure below shows the YOLOv11s architecture

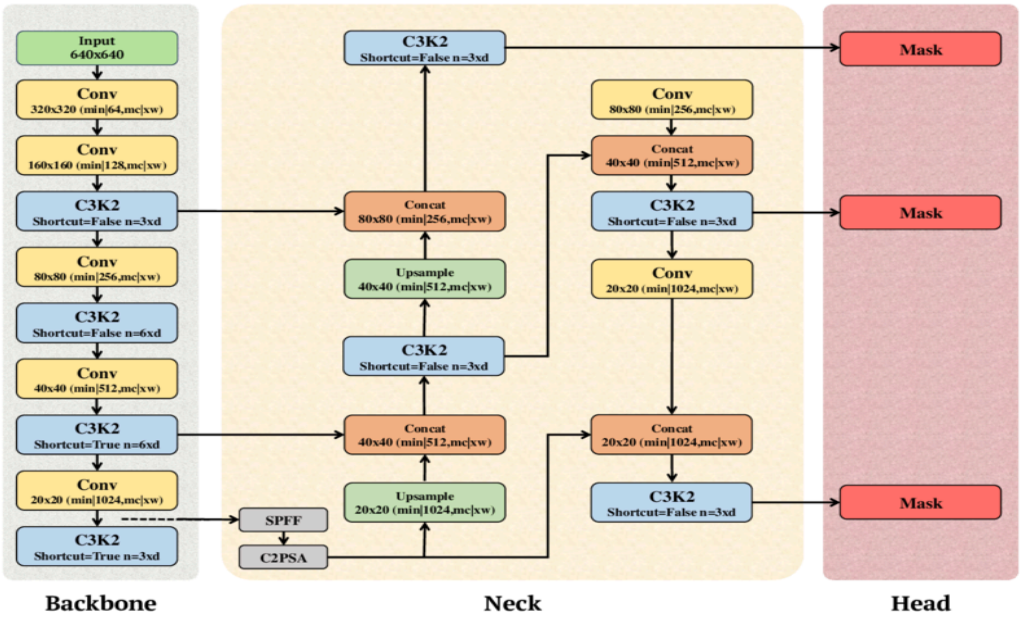

Figure 8: *YOLOv11s architecture[16]*

YOLOv11s offers several advantages for our specific application requirements:

- Computational Efficiency: Optimized for real-time inference on edge devices
- Detection Accuracy: Superior performance in small object detection scenarios
- Memory Footprint: Reduced model size suitable for Raspberry Pi 4 constraints
- Framework Maturity: Robust implementation with extensive community support

5.2.1. YOLO Format Conversion and Dataset Structuring

The transition from GTSRB's native annotation format to YOLO-compatible structure required careful consideration of bounding box coordinate systems and class indexing. We developed a comprehensive conversion pipeline that accurately transformed Pascal VOC format annotations to YOLO's normalized coordinate system.

The conversion process involved:

1. Coordinate Normalization: Converting absolute pixel coordinates to relative coordinates normalized by image dimensions
2. Bounding Box Format: Transforming (x_min, y_min, x_max, y_max) to (x_center, y_center, width, height)
3. Class Remapping: Ensuring consistent class indexing across training and validation sets
4. Dataset Partitioning: Implementing 90-10 train-validation split for robust model evaluation

5.2.2. Training Configuration and Hyperparameter Optimization

Our training configuration was meticulously designed to achieve optimal performance while preventing overfitting and ensuring stable convergence. The model was trained for 50 epochs with a batch size optimized for the available computational resources.

Key Training Parameters:

- Image Resolution: 640×640 pixels (standard YOLO input size)
- Learning Rate Schedule: Cosine annealing with warm-up period

- Data Loading: Multi-threaded preprocessing for efficient GPU utilization
- Loss Function: Combined classification, localization, and confidence losses
- Regularization: Dropout and weight decay for generalization improvement

5.3. Model Performance Analysis and Validation

The trained model demonstrated exceptional performance across multiple evaluation metrics, validating the effectiveness of our development approach. Comprehensive testing revealed consistent improvement throughout the training process, with final metrics indicating production-ready performance levels.

5.3.1. Training Convergence and Loss Analysis

The training process exhibited stable convergence across all loss components, indicating well-balanced learning dynamics. The multi-component loss function effectively optimized object localization, classification accuracy, and confidence calibration simultaneously, as it is shown in figure 9.

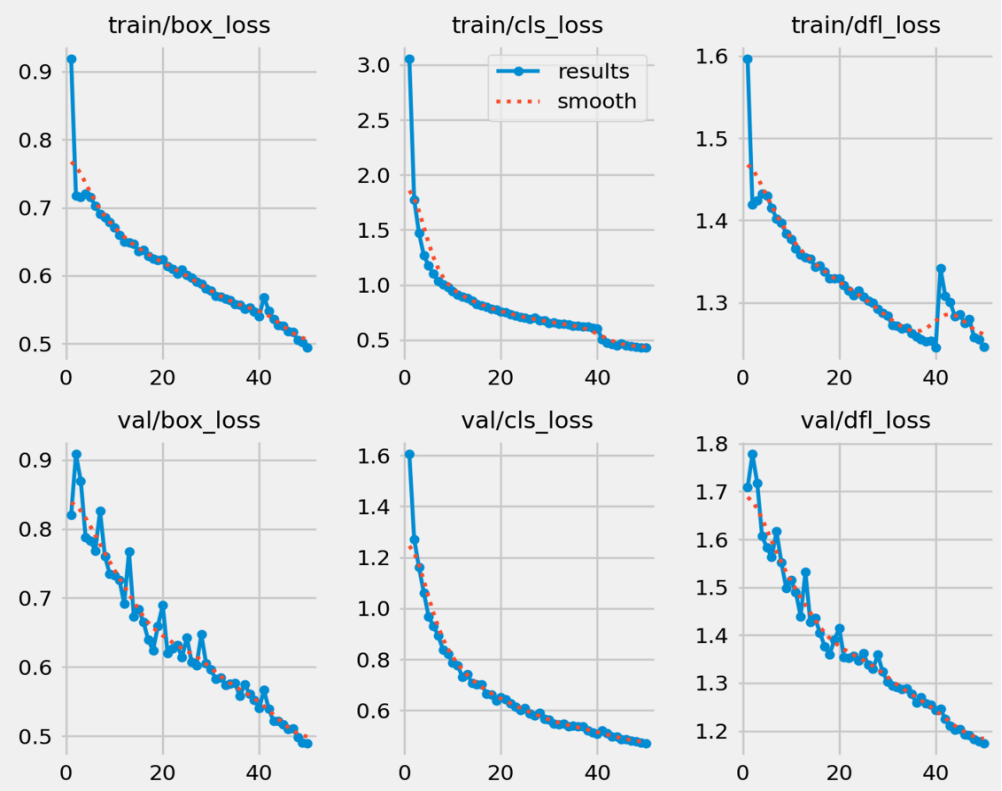

Figure 9: *Training and validation loss curves across 50 epochs*

Key performance indicators at training completion:

- Box Loss: Decreased from 0.919 to 0.489 (46% reduction)
- Classification Loss: Reduced from 3.055 to 0.429 (86% reduction)
- Distribution Focal Loss: Improved from 1.597 to 1.174 (26% reduction)

5.3.2. Performance Metrics Evolution

The model's performance metrics, in figure 10, demonstrate consistent improvement throughout the training process, with all key indicators reaching exceptional levels by training completion.

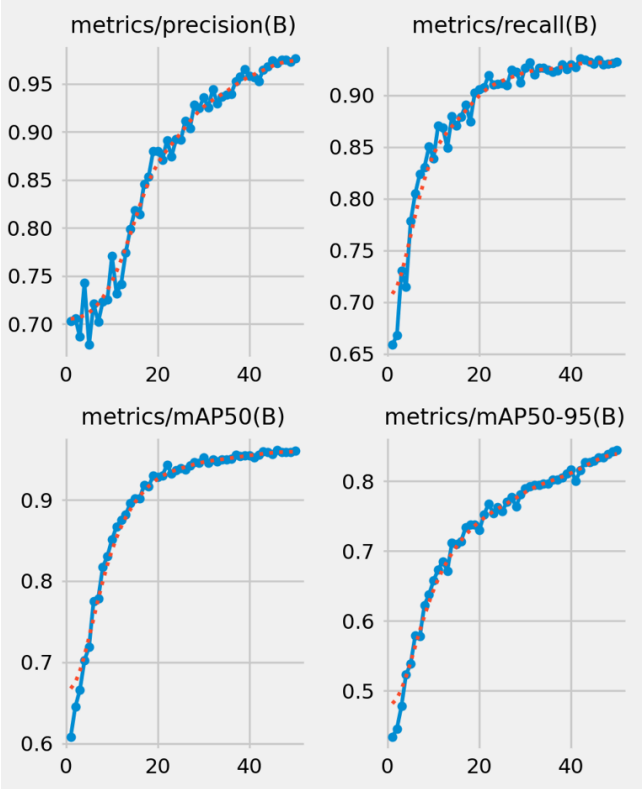

***Figure 10**: Evolution of performance metrics across training epochs*

Performance Metrics:

- Precision: Achieved 97.65% (final epoch)
- Recall: Reached 93.23% (final epoch)
- mAP@0.5: 96.05% (IoU threshold of 0.5)
- mAP@0.5:0.95: 84.41% (averaged across IoU thresholds 0.5-0.95)

Precision: measures the accuracy of positive predictions, calculated as the ratio of true positive detections to the total number of positive predictions (true positives + false positives). A precision of 97.65% indicates that out of all objects the model predicted as positive detections, 97.65% were correctly identified.

Recall: Recall measures the model's ability to identify all relevant instances, calculated as the ratio of true positive detections to the total number of actual positive instances (true positives + false negatives). A recall of 93.23% means the model successfully detected 93.23% of all actual objects present in the dataset.

mAP@0.5: Mean Average Precision at IoU threshold 0.5 evaluates detection accuracy by considering a detection correct when the Intersection over Union (IoU) between the predicted and ground truth bounding boxes exceeds 0.5. This metric averages the precision across all object classes at this specific IoU threshold.

mAP@0.5:0.95: This comprehensive metric calculates the mean Average Precision across multiple IoU thresholds from 0.5 to 0.95 in increments of 0.05. It provides a more stringent evaluation of model performance by requiring higher localization accuracy, making it particularly valuable for applications demanding precise object localization.

5.3.3. Precision-Recall Analysis and Model Calibration

The model's precision-recall characteristics demonstrate exceptional discriminative capability across the confidence threshold spectrum. The F1-confidence curve analysis reveals optimal operating points for different application scenarios, enabling fine-tuned deployment based on specific safety requirements, as shown in figure 11.

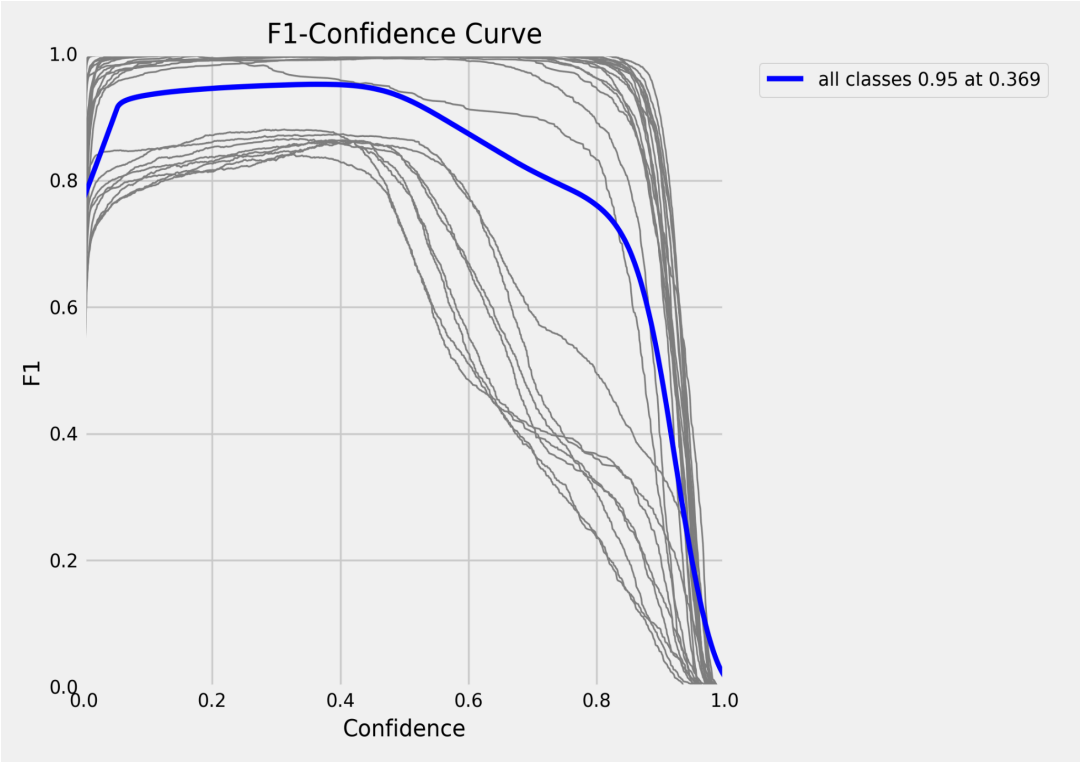

***Figure 11**: F1-Confidence curve showing across all classes*

The confidence curve analysis indicates that the model maintains high precision (>95%) while achieving excellent recall (>93%) at the optimal threshold of 0.369. This calibration ensures minimal false positive rates while maintaining high detection sensitivity, crucial for safety-critical driver assistance applications.

5.3.4. Per-Class Performance Analysis

Detailed analysis of per-class performance reveals consistent accuracy across different sign categories, as shown in figure 12 :

High-Performance Classes (>65% mAP):

- Speed limit signs (particularly 30, 50, 60 km/h)
- Stop and yield signs
- No passing indicators

Moderate Performance Classes:

- Warning signs with complex geometries
- Directional indicators
- Road work and construction signs

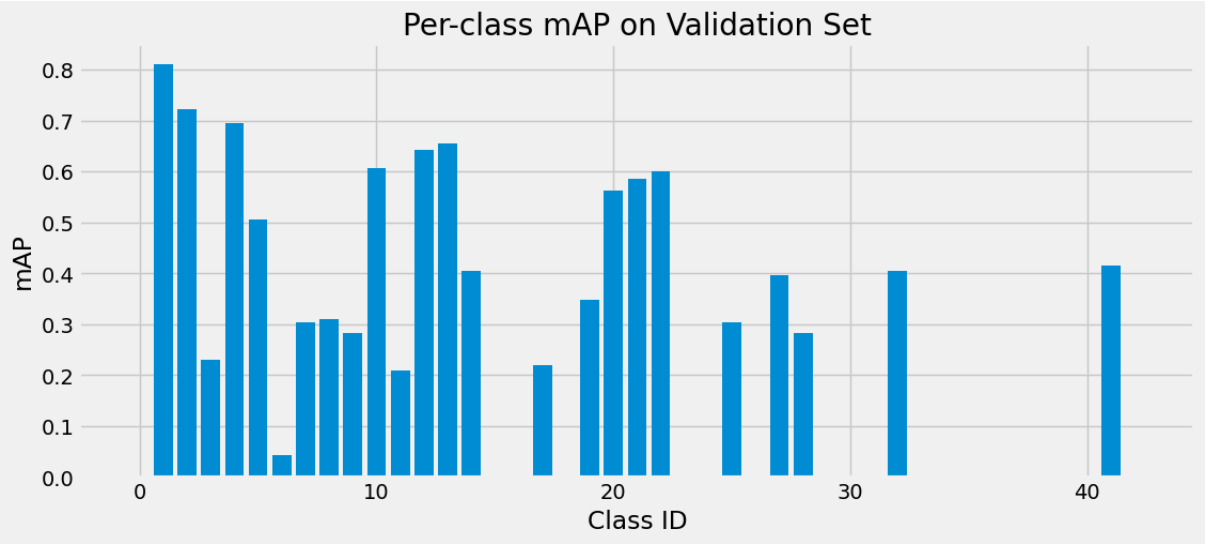

Figure 12: *Per-class mAP scores showing model performance across different traffic sign categories*

6. Challenges & Solutions

During the development of our AI-powered driver assistance system, we encountered several technical and integration-related challenges. This section outlines the most significant difficulties and the solutions we applied to ensure the system functioned reliably under test conditions.

6.1. Challenge: Realistic Vehicle Data via CAN Bus

Problem:

In real automotive applications, vehicle speed and other sensor data are typically transmitted over a CAN (Controller Area Network) bus. The most accessible interface to this bus is the OBD2 (On-Board Diagnostics 2) port, which has been legally required in all consumer

vehicles in Europe and the US since 2006. This port provides standardized access to CAN frames and is usually located within reach of the driver's seat, allowing diagnostic tools or embedded systems to receive real-time vehicle data[9]. The figure below showcases the common pinout of an OBD-II port.

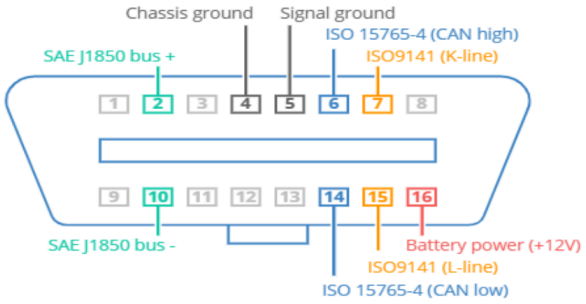

Figure 13: *OBD2 Connector [14]*

Our original intention was to connect to this system using two MCP2515 CAN modules, one on the Raspberry Pi and the other on the Arduino as shown in the figure 14. However, due to the unavailability of a second MCP2515 module during testing, we were unable to implement full CAN-based communication.

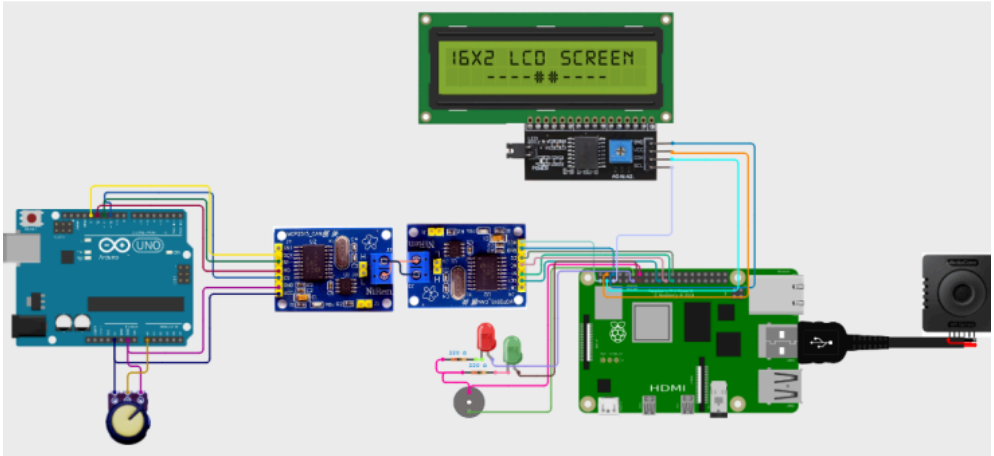

Figure 14: *System architecture using MCP2515 CAN modules*

Solution:

To ensure progress on system integration and testing, we substituted the CAN setup with UART communication via USB serial. In this approach, the Arduino Uno is connected to the Raspberry Pi through USB. A 10kΩ potentiometer is connected to the Arduino to simulate

analog speed input. The Arduino reads this value, maps it to a range of 0–120 km/h, and transmits it as a speed value over the serial connection.

Although this method does not replicate the full complexity of in-vehicle CAN messaging, it allowed us to simulate realistic and adjustable speed data without requiring a vehicle or full CAN hardware setup. This workaround enabled us to validate our system's detection, comparison, and alert logic under controlled and repeatable test conditions.

6.2. Challenge: Running Real-Time AI on a Low-Power Device

Running a YOLOv11s model on a Raspberry Pi 4 presented performance challenges due to the device's limited processing power and memory. Processing full-resolution video frames in real time resulted in occasional latency and dropped detections, especially under sustained use.

Solution:
To address these limitations, we used the smallest available version of YOLOv11 (YOLOv11s), optimized for edge deployment. Additionally, we reduced the input frame resolution and implemented optional frame skipping, which lowered the computational load. This trade-off preserved a balance between real-time performance and detection accuracy, allowing the system to remain responsive without significantly compromising reliability [2][3].

6.3. Challenge: Serial Communication Errors

At times, the Raspberry Pi received corrupted or incomplete data from the Arduino via USB serial. This caused errors in speed interpretation and occasionally disrupted system logic.

Solution:
We implemented robust error handling in Python using try-except blocks and checks for numeric-only input. We also added serial flushing and minimal delay in the Arduino loop to ensure consistent transmission. This significantly reduced communication issues during long runtime tests.

6.4. Challenge: Synchronization Between Speed Input and YOLO Detection

There was no built-in synchronization between the video frames processed by YOLO and the speed values read from the Arduino. This caused occasional mismatches between speed comparisons and detection timing.

Solution:
We ensured that the speed value used for comparison was updated continuously and stored globally. The logic was restructured so that every decision (buzzer, LED, LCD update) was based on the latest available detection and speed reading. While not perfect, this improved consistency during practical testing.

6.5. Challenge: False Detections in Speed Limit Recognition

YOLOv11s occasionally misclassified signs or detected nonexistent ones, especially in cases of blurry video frames or excessive ambient noise.

Solution:
To address this, we applied filtering logic: speed limit signs were only considered valid if they were detected across multiple consecutive frames. We also adjusted the model's confidence threshold to reduce sensitivity to noise. These steps improved overall detection stability [2].

7. Future Improvements

While the current implementation of the AI-powered driver assistance system demonstrates functional performance in detecting speed limits and responding to overspeeding, several opportunities exist to enhance and expand the system in future iterations.

7.1. Integration with GPS and Map Data

Incorporating a GPS module would allow the system to access real-time location data and cross-reference it with map-based speed limits. This would improve reliability in cases where signs are missing or occluded, providing a second layer of verification.

7.2. Voice Feedback for Driver Alerts

To improve user experience and accessibility, the system could be extended to provide spoken alerts using a text-to-speech module. For example, the system could say “You are exceeding the speed limit” instead of relying only on a buzzer and LEDs.

7.3. Enhanced Object Detection (Multi-class YOLO Model)

Future versions could detect more than just speed limit signs—such as stop signs, pedestrian crossings, traffic lights, or obstacles. This would require training or fine-tuning the YOLO model on a broader dataset but would significantly increase system capability.

7.4. Vehicle Control Integration

In an advanced setup, the system could be connected to vehicle actuators (e.g., throttle or brakes) for autonomous intervention in case of speeding. While this goes beyond the current scope, it aligns with the trend toward self-driving assistance systems.

7.5. Improved Camera Module and Night Vision Support

Using the official Raspberry Pi Camera Module or adding infrared support would enhance image clarity and make the system more reliable under low-light or night-time driving conditions.

8. Conclusion

This project successfully demonstrates the feasibility of building a low-cost, real-time driver assistance system using embedded hardware and artificial intelligence. By integrating object detection through YOLOv11s with real-time speed monitoring via Arduino and Raspberry Pi, we were able to develop a prototype that detects speed limit signs and compares them with the current vehicle speed, triggering appropriate visual and auditory alerts.

The combination of a simulated speed input system using a potentiometer, USB serial communication, and GPIO-controlled outputs (buzzer, LEDs, LCD) allowed for realistic testing without needing access to a physical vehicle. The system operated with satisfactory accuracy and responsiveness, proving the value of lightweight deep learning models such as yolov8n.pt in constrained environments like the Raspberry Pi [2][3].

Despite some challenges—including processing speed limitations, synchronization issues, and occasional detection inaccuracies—the system met its objectives. We applied practical solutions such as model optimization, error filtering, and power management to ensure consistent functionality.

Beyond its technical merit, this project illustrates the potential for democratizing driver safety technologies. By relying on open-source tools and widely available hardware, we developed an intelligent assistant system that could be adapted for educational purposes, fleet monitoring, or future integration into real vehicles.

The work done here lays a solid foundation for future enhancements, including the use of GPS, additional sign classes, voice alerts, and cloud-based data reporting. Ultimately, this system is a small but meaningful step toward safer roads through smart, accessible technology.

References

[1] The accident situation in Tunisia

https://news-tunisia.tunisienumerique.com/road-accidents-in-tunisia-tunis-leads-sfax-is-the-deadliest/

(accessed on 02/06/2025)

[2] Ultralytics YOLOv11s Documentation

https://docs.ultralytics.com/models/yolo11/

(accessed on 06/02/2025)

[3] OpenCV-Python Documentation

https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

(accessed on 06/02/2025)

[4] Raspberry Pi GPIO with Python

https://www.halvorsen.blog/documents/programming/python/resources/powerpoints/Raspberry%20Pi%20GPIO%20with%20Python.pdf

(accessed on 06/02/2025)

[5] RPi.GPIO Documentation

https://www.raspberrypi.com/documentation/computers/os.html

(accessed on 06/02/2025)

[6] RPLCD Library for I2C LCDs

https://rplcd.readthedocs.io/en/stable/getting_started.html

(accessed on 06/02/2025)

[7] Raspberry Pi Documentation

https://www.raspberrypi.com/documentation/

(accessed on 06/02/2025)

[8] Raspberry Pi as an Embedded AI Hub

https://docs.arduino.cc/language-reference/en/functions/communication/serial/

(accessed on 06/02/2025)

[9]OBD2 connector

https://en.wikipedia.org/wiki/OBD-II_PIDs

(accessed on 06/02/2025)

[10] Stallkamp, J., Schlipsing, M., Salmen, J., & Igel, C. (2012). Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, 32, 323-332.

[11] Buslaev, A., Iglovikov, V. I., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). Albumentations: fast and flexible image augmentations. *Information*, 11(2), 125.

[12] Jocher, G., Chaurasia, A., & Qiu, J. (2023). YOLO by Ultralytics. *GitHub repository*. https://github.com/ultralytics/ultralytics

(accessed on 06/02/2025)

[14]OBD2 Explained - A Simple Intro

https://www.csselectronics.com/pages/obd2-explained-simple-intro

(accessed on 06/02/2025)

[15]Tunisia's National Road Safety Observatory

https://onsr.nat.tn/onsr/index.php?page=0ar

(accessed on 06/02/2025)

[16] LEAF-Net: A Unified Framework for Leaf Extraction and Analysis in Multi-Crop Phenotyping Using YOLOv11 - Scientific Figure on ResearchGate. https://www.researchgate.net/figure/The-schematic-diagram-of-YOLOv11-illustrating-its-three-core-components-Backbone-Neck_fig1_386467184 (accessed on 01/06/2025)