Chapter 1
INTRODUCTION

Chapter 1
INTRODUCTION

1.1 Background

Nowadays, the autonomous or self-driving vehicles have a predominant impact in transportation industry. The vehicle-to-vehicle communications, artificial intelligence (AI) and cloud computing technology are playing a major role in evolution of autonomous vehicles and interest among the researchers from academia and industry people. These cutting-edge technologies help the autonomous vehicles to percept the environment or detect the objects using end sensors, radar, LiDAR, cameras, etc., make proactive decisions, and control driving functions, accordingly. The autonomous vehicles have a long past and till now these vehicles are categorized from Level-0 to Level-5. Most of the vehicle accidents are caused by human fault. To overcome this tragedy, self-driving vehicle concept came into the market. These self-driving vehicles or autonomous vehicles have ADAS (Advanced Driver Assistance System) feature which helps in reducing the no. of accidents by analyzing its surroundings and taking necessary actions. The ADAS feature uses advance technology, sensors, cameras, RADAR etc. These vehicles are operated by partial/less human effort or fully automatic manner. They use sensors, cameras, RADAR, LiDAR to perceive the surroundings like; traffic signal, objects etc. and take decisions on real time.

While driving, we humans cannot see all the directions at a time but the autonomous vehicles can see the 360° view with the help of different sensors and cameras efficiently and also can measure the longitude and latitude of the vehicle. Till now the automated driving levels are categorized into 6 levels i.e. from Level-0 to Level-5. In the first 3 levels (Level-0, Level-1 and Level-2) the human monitors the driving environment whereas in the other 3 levels (Level-3, Level-4 and Level-5) the automated system monitors the driving environment. These self-driving vehicles use Artificial Intelligence (AI) to simulate like human, percept the surroundings, perform actions such as; steering, accelerating and apply brakes. The software of the vehicles is associated with Google Maps for advance recognition of destination, traffic signals, road conditions etc. These vehicles are trained from different data sources, real life scenarios like; traffic signals, objects, pedestrians, and other conditions using neural networks. Here all the data during driving are stored in cloud for future analysis, training the model and better improvement of the vehicle. Different automobile companies like; BMW, Audi, Tesla, Volvo are still working on development of self-driving vehicles. However, in India, companies like; TATA and Mahindra have introduced some of the ADAS features in their vehicles which comes under Level-2 category.

The autonomous vehicles have potential to bring measure change in future transportation industry which will definitely lead to reduce the no. of accidents, traffic jam, efficient driving environment. Despite all of the major advantages, there is some problem concerned with self-driving car like; cyber security, software issue etc. which can have bad impact on the vehicle. So, we should stay alert while driving these vehicles instead of giving all the control to the vehicle itself.

1.2 Motivation

According to official statistics published by the Ministry of Road Transport and Highways (MoRTH), 4,61,312 persons were killed in road crashes in the year 2022. This corresponds to 11.3 deaths per 100,000 population. The table topper becomes the state of Tamil Nadu with 64,105 total reported deaths in the year 2022. This makes the need for an autonomous driving mechanism as the fatalities in the human driving is increasing day-by-day. The leading factors which lead to such accidents are driving under the influence of alcohol, underage drivers, poor infrastructure. Autonomous vehicles sense the environment from any interference in the route, speed control, correct navigation etc. The self-driving technology can help in eliminating the driver fatigue, distraction and improve decision making. Autonomous vehicles can optimise the traffic regulation by communicating with nearby vehicles and therefore reducing traffic congestion. Autonomous vehicles can improve the mobility for the people with disability, kids as it can reduce the cost of transportation and increases accessibility. Autonomous vehicles can also reduce the fuel consumption and emissions and thus benefitting the environment. These promising benefits provide a significant motivation for this study.

1.3 Contribution of the Proposed Work

This proposed work presents a significant contribution to the development and practical application of autonomous vehicle technologies through the design and implementation of a low-cost, modular self-driving car prototype. The following are the key contributions of the study:

- **Design and Implementation of a Low-Cost Autonomous Vehicle Prototype**

  The dissertation introduces a cost-effective and scalable prototype of a self-driving car that integrates real-time computer vision, embedded systems, and IoT functionalities. By utilizing affordable components such as the Raspberry Pi, ESP32 microcontroller, and L298N motor driver, the project demonstrates how advanced autonomous features can be implemented in resource-constrained environments, making it suitable for educational and experimental use.

- **Custom Traffic Sign Detection Using YOLOv5**

  A major technical contribution is the training and deployment of a custom YOLOv5 (You Only Look Once) object detection model for real-time traffic sign recognition. The model, trained on both public and custom datasets, achieved high accuracy while being optimized for edge devices like the Raspberry Pi. This significantly improved real-time perception capabilities compared to traditional CNN-based models.

- **Real-Time Lane Detection with OpenCV**

  The prototype implements a real-time lane detection module using classical computer vision techniques (Canny edge detection, Hough transform) in OpenCV, allowing the vehicle to navigate road lanes effectively and maintain driving discipline.

- **Integration of Safety Features: Drowsiness and Alcohol Detection**

  The system includes drowsiness detection using dlib-based facial landmark detection and alcohol detection using the MQ3 sensor, both of which contribute to the safety and

reliability of semi-autonomous vehicles. These features enable the prototype to monitor driver conditions and respond accordingly.

- **Remote Control and Manual Override via Mobile Application**

  A Bluetooth-enabled Android mobile application, developed using React Native and Kotlin, allows for manual control of the vehicle. This application ensures real-time user interaction and testing, supports system debugging, and provides a fail-safe manual override in case of autonomous system malfunction.

- **Cloud-Based Monitoring with ThingSpeak**

  The integration of ThingSpeak cloud services via UDP enables real-time telemetry data logging and remote system monitoring. This adds a robust IoT component to the system, supporting data visualization, diagnostics, and future improvements through cloud analytics.

- **System Modularity and Educational Value**

  The modular design of the system allows for easy expansion, testing, and integration of additional components such as V2X communication or LiDAR. This makes the project a valuable educational tool for students and researchers exploring embedded systems, AI, robotics, and smart transportation.

Through the above contributions, this dissertation provides a comprehensive reference for developing autonomous driving prototypes and serves as a foundational step towards more advanced and real-world deployable self-driving car systems.

1.4 Organisation of Dissertation

This dissertation is structured into multiple chapters, each focusing on a specific component of the research and development process involved in designing and implementing a low-cost autonomous vehicle prototype. The chapters are organized as follows:

- **Chapter 2: Literature Review**

This chapter presents a detailed survey of existing work related to autonomous driving technologies. It covers key topics such as computer vision for traffic sign and lane detection, embedded systems for vehicle control, communication protocols, safety systems, and cloud-based data analysis. The review also compares traditional approaches with the methods adopted in this project, emphasizing the shift to YOLOv5 and real-time embedded solutions.

- **Chapter 3: System Model & Problem Statement**

This chapter describes the overall architecture of the self-driving car system, including hardware components, software modules, sensor integration, and control logic. It explains the modular design, detailing the functions and interactions of each subsystem: perception, decision-making, control, communication, and cloud integration. The chapter also addresses challenges faced during system implementation and their solutions.

- **Chapter 4: Proposed Methodology**

This chapter outlines the step-by-step development methodology used to build the autonomous vehicle. It includes dataset collection and model training for traffic sign recognition, computer vision techniques for lane detection, logic for obstacle avoidance and cruise control, drowsiness and alcohol detection mechanisms, and real-time mobile control via a custom-built application. Each subsystem is described in terms of its design, tools, implementation, and performance of our designed system.

- **Chapter 5: Simulation Results and Discussion**

This chapter discusses the experimental setup, testing scenarios, and performance metrics used to evaluate the effectiveness of the self-driving car prototype. It presents results from real-time testing of traffic sign recognition, lane following, safety feature activation, and manual control. The chapter also highlights system strengths and areas for improvement.

- **Chapter 6: Conclusions and Future Works**

The final chapter summarizes the key outcomes and contributions of the dissertation. It reflects on the system's limitations and proposes future enhancements, such as integration with LiDAR, V2X communication, and advanced navigation strategies. The chapter concludes by emphasizing the educational value and real-world applicability of the prototype system.

Chapter 2
LITERATURE REVIEW

Chapter 2

LITERATURE REVIEW

The development of self-driving cars represents one of the most transformative innovations in modern transportation. Over the past decade, autonomous vehicle technology has evolved rapidly, driven by advancements in computer vision, machine learning, sensor fusion, and real-time embedded systems. This growing interest stems from the promise of reducing traffic accidents, improving mobility, and enhancing transportation efficiency.

To understand the current state of the field and position our project within this landscape, it is essential to explore existing research, methodologies, and technologies that have contributed to the progress of self-driving systems. This literature review provides a comprehensive overview of key areas such as lane detection, traffic sign recognition, obstacle avoidance, decision-making algorithms, and vehicle control mechanisms. Additionally, it highlights the integration of cloud-based monitoring, microcontroller communication protocols, and mobile interfaces, which collectively enable the functionality of modern autonomous vehicles. The state-of-the-arts related to the proposed work of the dissertation are briefly illustrated as follows.

2.1 Traffic Sign Detection and Recognition

Gupta *et al.* [1] presented traffic sign recognition in self-driving car using a traditional Convolutional Neural Network (CNN). CNNs are widely used in image classification tasks due to their ability to automatically extract spatial hierarchies of features from input images. Our model was trained on a dataset containing various traffic signs (e.g., stop, left, right) and deployed on the Raspberry Pi. While the CNN achieved acceptable accuracy during offline training, its real-time performance on embedded hardware was suboptimal. The model exhibited slower inference speed and struggled with consistent detection in dynamic conditions, especially under varying lighting, motion blur, and small object sizes. These limitations highlighted the need for a more optimized solution tailored to real-time object detection in constrained environments.

Khanam *et al*. [2] demonstrated that the critical aspect of autonomous vehicles is their ability to perceive and interpret the environment accurately. Traditional methods for traffic sign detection relied on Convolutional Neural Networks (CNNs), which, while effective for image classification, often struggled with real-time performance on embedded hardware due to slower inference speeds and inconsistent detection under dynamic conditions. Diwan, *et al.* [6] To address these limitations, recent studies have shifted toward YOLOv5 (You Only Look Once), a state-of-the-art object detection model known for its balance of speed and accuracy. YOLOv5 treats detection as a single regression problem, enabling real-time identification of multiple objects with minimal latency, making it particularly suitable for deployment on resource-constrained platforms like the Raspberry Pi.

2.2 Lane Detection Techniques

Xu & Zhang [4] Lane detection, another cornerstone of autonomous driving, has traditionally employed computer vision techniques such as Canny edge detection and Hough transforms. These methods are effective in controlled environments but face challenges in complex scenarios like poor lighting or occluded lane markings. Megalingam *et al.* [31] Recent research explores the integration of deep learning-based approaches, such as LaneNet and SCNN, to improve robustness. Additionally, sensor fusion techniques combining camera data with GPS and IMU inputs have shown promise in enhancing lane detection accuracy, particularly in dynamic or unstructured environments.

2.3 Obstacle Avoidance

Mahmud *et al.* [7] Obstacle avoidance systems have evolved from basic ultrasonic sensors like the HC-SR04 to more sophisticated solutions involving LiDAR and radar (2023). Ziębiński *et al.* [8] While ultrasonic sensors are cost-effective for short-range detection, their limitations in range and object classification have led to the adoption of multi-sensor systems that combine ultrasonic, vision, and LiDAR data for comprehensive environmental perception.

2.4 Drowsiness and Alcohol Detection

Mehta *et al.* [32] Safety features such as drowsiness and alcohol detection have also gained prominence in autonomous and semi-autonomous vehicles. Drowsiness detection systems leverage facial landmark tracking and Eye Aspect Ratio (EAR) calculations to monitor driver alertness, triggering alerts when signs of fatigue are detected. Hariharan *et al.* [27] Similarly, alcohol detection systems using MQ-3 sensors provide a preventive measure against impaired driving by disabling vehicle operation when alcohol vapours are detected.

Numerous research studies have explored self-driving car systems using deep learning approaches, focusing on tasks such as object detection, lane following, traffic sign recognition, and obstacle avoidance. While many of these implementations employed earlier versions of object detection models like YOLOv3, SSD, or Faster R-CNN, our work distinguishes itself by leveraging YOLOv5, a more recent and advanced model known for its superior speed and accuracy in real-time detection. In this project, we successfully implemented YOLOv5 for traffic sign recognition with high precision and recall, demonstrating enhanced detection performance even in challenging environments. To the best of our knowledge, the specific combination of YOLOv5 with real-time actuation on embedded hardware like Raspberry Pi 4 and ESP32, along with live data monitoring via ThingSpeak Cloud, has not been previously achieved at this level of accuracy and integration. This positions our work as a novel and practical contribution to the ongoing development of autonomous vehicle technologies using deep learning.

2.5 Chapter Summary

These studies laid the groundwork by demonstrating the feasibility of real-time detection and control in constrained environments. However, limitations in accuracy, processing speed, and hardware compatibility often hindered full deployment on low-cost embedded systems. Recent

works have attempted to bridge this gap, but few have explored the integration of YOLOv5, an optimized and lightweight object detection model, for real-time traffic sign recognition on edge devices like the Raspberry Pi 4. Additionally, the combination of traffic analysis, driver alertness detection using Dlib, and IoT-based data transmission to platforms like ThingSpeak remains underexplored. This review underscores the need for a unified and efficient system that can perform detection, actuation, and cloud integration with high accuracy precisely what our project aims to achieve.

Chapter 3

SYSTEM MODEL & PROBLEM STATEMENT

Chapter 3
SYSTEM MODEL & PROBLEM STATEMENT

3.1 System Model

The System Model chapter serves as a foundational framework for understanding how various components within a system interact, operate, and contribute to the overall functionality. A system model is a simplified representation of a real-world system, designed to capture its essential features and behavior while omitting unnecessary complexity. It provides a clear, structured view of the system's architecture, processes, data flow, and interactions between different modules or subsystems. In the sections that follow, we will detail the system's key elements, explore their interactions, and present diagrams where necessary to illustrate the conceptual and practical structure of the system.

The system model section focuses the models/ techniques that are used in our proposed methodology (discuss in Chapter 4) are briefly elucidated as follows:

3.1.1 Traditional Approach for Traffic Sign Recognition

Convolutional Neural Networks (CNNs) are a specialized kind of artificial neural network used predominantly in the field of computer vision. CNNs are designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers. Unlike traditional neural networks, which treat inputs as one-dimensional vectors, CNNs process data in multiple dimensions, making them ideal for images that have width, height, and color channels. A CNN begins with an input image that is passed through a series of convolutional layers, where filters (also called kernels) convolve around the input to produce feature maps. These feature maps capture various aspects of the image like edges, textures, and patterns. The learned filters in the initial layers typically detect simple patterns, such as edges and colors, while deeper layers progressively recognize complex patterns like shapes or even entire objects.

Internally, the working of a CNN can be broken down into several key stages. The first is the convolution operation, which involves sliding a filter over the input image and computing dot products between the filter and the receptive field. This operation helps in feature extraction. Following convolution, a non-linear activation function such as ReLU (Rectified Linear Unit) is applied to introduce non-linearity into the model, allowing it to learn more complex functions. After activation, pooling layers (commonly max pooling) are used to reduce the spatial dimensions of the feature maps, which helps in reducing computation and controlling overfitting. These layers downsample the feature maps while retaining the most significant information. As the network deepens, the spatial size reduces, but the depth (number of feature maps) increases, allowing the network to learn a rich hierarchy of features. Eventually, the

output of the convolution and pooling layers is flattened and fed into fully connected layers, which perform the high-level reasoning in the network. The final layer is typically a softmax or sigmoid layer for classification tasks. CNNs are trained using a process called backpropagation, where the model adjusts the filter values based on the error between the predicted and actual outputs. This entire architecture enables CNNs to achieve remarkable performance in image recognition, classification, and various other visual tasks. Here the figure 3.1 describes the internal working of CNN.

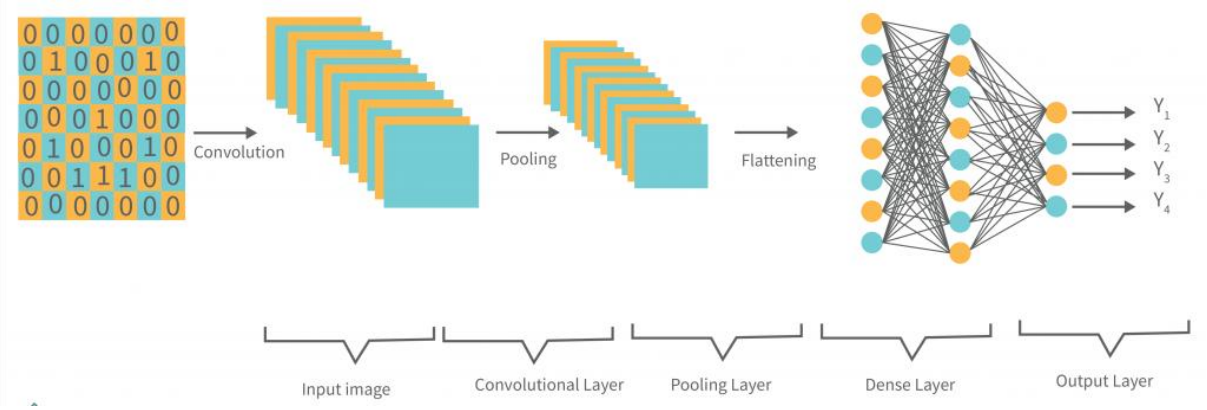

Fig 3.1: Internal working of CNN.

Convolutional Neural Networks have revolutionized the field of computer vision by enabling machines to recognize and understand images with near-human accuracy. By stacking layers of convolutions, activations, and pooling, CNNs learn to extract hierarchical features that can be used for complex visual tasks. With the advent of powerful GPUs and large datasets, CNNs have become a cornerstone of modern AI applications. A deep understanding of CNNs from basic layer structure to advanced architectures provides a strong foundation for building and optimizing intelligent visual systems.

3.1.2 Ultralytics YOLO

Ultralytics is a platform where a developer can create, train and deploy machine learning models easily. Ultralytics is a platform which gives supports of various machine learning and deep learning frameworks. Ultralytics is mostly used in vision programming tasks like object recognition, image classification and image segmentation etc. YOLO (You only look once) is a state-of-the-art (SOTA) object detection algorithm that has become main method of detecting objects in the field of computer vision. Previously people used techniques such as sliding window object detection, R CNN, Fast R CNN and Faster R CNN. But after its invention in 2015, YOLO has become an industry standard for object detection due to its speed and accuracy. Ultralytics YOLO models are widely used in traffic sign detection for self-driving cars. These models can accurately detect and recognize traffic signs in real-time, which is crucial for safe and efficient navigation. The YOLO models are known for their speed and efficiency, making them suitable for deployment in edge devices such as those used in autonomous vehicles. This capability ensures that traffic sign detection can be performed quickly and accurately, even in complex and challenging environments. Here the figure 3.1 shows how detection is performed over a captured image frame.

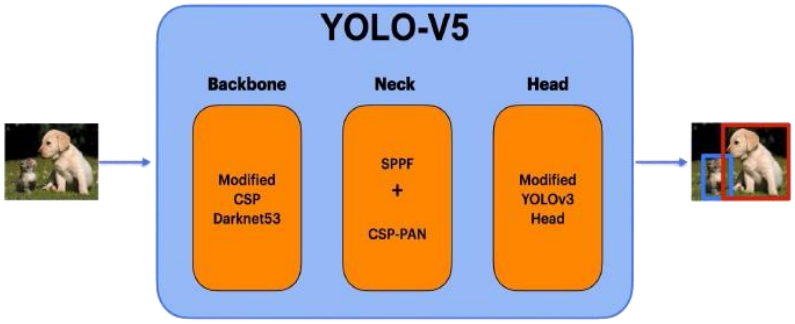

Fig 3.2: Detection of objects using YOLO.

Benefits of using YOLO for Traffic Sign Detection:

- **Real-time performance**: Critical for self-driving cars to make split-second decisions.
- **Lightweight models** (like YOLOv5s or v5n): Can run on embedded systems like Jetson Nano or Raspberry Pi.
- **Transfer learning**: Easily fine-tune on custom traffic sign datasets.
- **Broad deployment support**: Can export to .pt, ONNX, TensorRT, CoreML, etc.

While several advanced versions of the YOLO (You Only Look Once) family are available such as YOLOv6, YOLOv7, and YOLOv8 we specifically chose YOLOv5 for our project based on a combination of performance, hardware compatibility, community support, and ease of deployment. Although newer versions like YOLOv8 offer improved accuracy and more modern architectures, they often require higher computational resources and more complex setups, which can be a challenge when working on resource-constrained platforms like the Raspberry Pi 4. Our project aimed to build a lightweight, efficient, and real-time traffic sign recognition system for a prototype self-driving vehicle. YOLOv5, particularly the YOLOv5s variant, offered a perfect balance between speed, accuracy, and model size.

YOLOv5 is a version of the YOLO (You Only Look Once) family of computer vision models used for object detection. It was released by Ultralytics on June 25, 2020, and comes in four main versions: small (s), medium (m), large (l), and extra-large (x), each offering progressively higher accuracy rates and taking different amounts of time to train.

YOLOv5s (small – fastest, lowest accuracy)

YOLOv5m (medium)

YOLOv5l (large)

YOLOv5x (extra-large – slowest, highest accuracy)

YOLOv5s, specifically, is the smallest version of YOLOv5, designed for faster processing speeds compared to larger versions like YOLOv5x. It uses a variant of the

EfficientNet architecture called EfficientNet-L2, which is more efficient than the EfficientDet architecture used in YOLOv5, with fewer parameters and higher computational efficiency. The architecture of YOLOv5s includes the use of Cross Stage Partial Networks (CSPNet) in its backbone to extract rich, informative features from input images. This helps in improving processing time with deeper networks. More over YOLOv5s is a smaller, faster version of YOLOv5 designed for efficient object detection tasks. Here the figure 5.2 shows the internal architecture of YOLO v5.

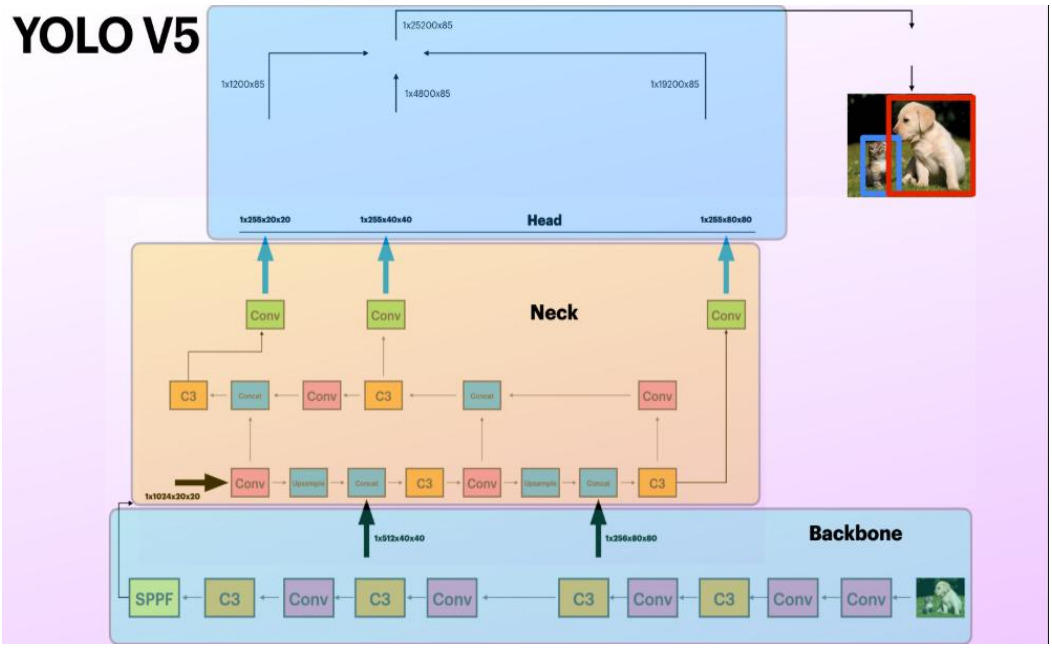

Fig 3.3: Internal architecture of YOLOv5.

YOLOv5 is a fast, accurate object detection model developed by **Ultralytics**. It's written in **PyTorch**, which makes it easy to use and modify. Despite the name, YOLOv5 is not from the original YOLO authors (Redmon et al.), but it became very popular due to its ease of use, speed, and performance.

YOLOv5 Variants

Table 3.1: YOLOv5 comes in different sizes each is a trade-off between speed and accuracy:

Model	Size	Speed (FPS)	Accuracy (mAP)	Use Case
v5n	Nano	Fastest	Lowest	Edge devices, real-time detection
v5s	Small	Very Fast	Decent	Mobile, low-latency tasks
v5m	Medium	Balanced	Better	General-purpose tasks
v5l	Large	Slower	High	High accuracy needs
v5x	Extra- large	Slowest	Highest	Heavy-duty applications

YOLOv5s

Specifically, YOLOv5s (the "small" model):

- Fast and lightweight
- Great for real-time applications
- Slightly lower accuracy than larger models
- Ideal for mobile and embedded devices

When to use YOLOv5s

- We need speed over perfect accuracy
- We're deploying to resource-constrained environments
- We're doing real-time detection (e.g., CCTV, drones)

The real-time traffic sign detection system effectively integrates deep learning (YOLOv5) with OpenCV to enable fast and accurate recognition of traffic signs from live video input. This setup plays a crucial role in perception modules of autonomous vehicles, allowing for real-time environmental understanding and decision-making.

3.1.3 Dlib

Dlib is a powerful modern C++ toolkit that provides machine learning algorithms and tools for creating complex software in C++ and Python. One of its most popular features is face landmark detection, which is extensively used in applications such as face recognition, emotion detection, and face alignment. Dlib's facial landmark detector is based on an ensemble of regression trees, which allows it to detect 68 specific key points on a human face with high accuracy and efficiency. The detection process typically involves two steps: first, using a Histogram of Oriented Gradients (HOG) + Linear SVM or a Convolutional Neural Network (CNN) to detect the face region; second, applying a pre-trained shape predictor model (like shape_predictor_68_face_landmarks.dat) to map the 68 facial landmarks. These landmarks represent positions around the eyebrows, eyes, nose, mouth, and jawline. Dlib's detector works well even under partial occlusion and different facial orientations, making it highly reliable for real-time applications. The library is well-optimized and can be integrated into various pipelines for tasks like face alignment (to correct pose variations), blink detection, emotion classification, and even lip reading. While Dlib's shape predictor is not a deep learning-based approach, it still offers remarkably good results, especially for small to medium-scale applications where computational resources are limited. For more robust or real-time systems, the face detection stage can be substituted with faster detectors like OpenCV's DNN or YOLO, followed by Dlib's landmark predictor.

The indexes of the 68 co-ordinates can be visualized on the figure 3.4:

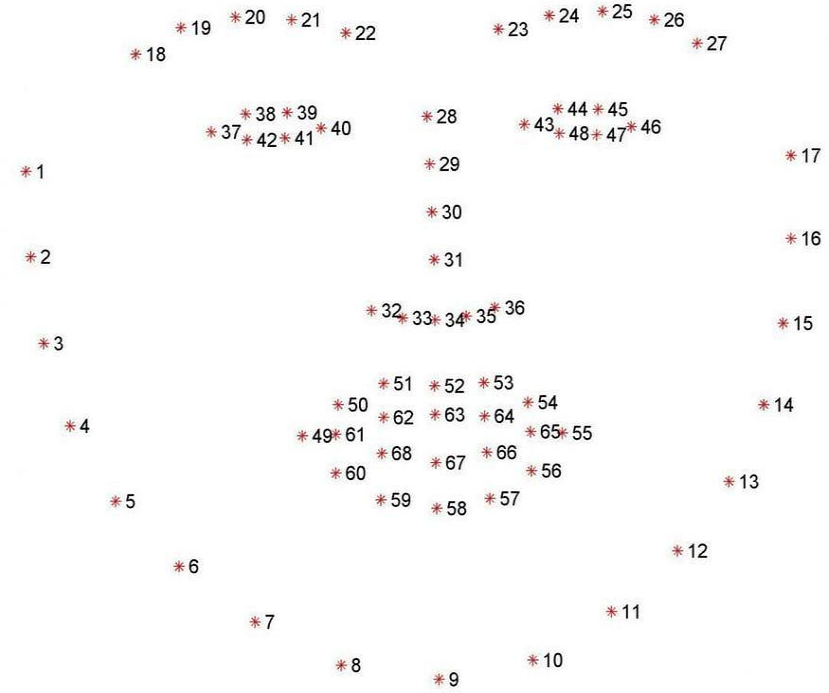

Fig 3.4 Facial co-ordinate representation.

- **Applications of Dlib**:
  - Face alignment (pose correction).
  - Emotion detection and facial expression recognition.
  - Head pose estimation.
  - Eye blink or mouth movement detection.
- **Pros**:
  - Lightweight and easy to use.
  - High accuracy for static and real-time images.
  - No need for GPU for decent performance.
- **Limitations**:
  - Not deep learning-based — may lack robustness under extreme variations.
  - Slower than lightweight CNN models when using HOG-based detection.
- **Integration Tips**:
  - Can combine with OpenCV or YOLO for faster face detection.
  - Best used for controlled environments or small-scale applications.

3.2 Problem Statement

The primary problem this project aims to address is the development of an intelligent self-driving car system capable of operating autonomously and safely in real-world environments. The system must accurately recognize traffic signs in real time using deep learning techniques, reliably detect and follow lanes under varying road and lighting conditions, and identify obstacles to avoid collisions. Additionally, it must monitor the driver's state to detect drowsiness or alcohol influence using facial analysis and sensors, ensuring human oversight safety. The car should also implement adaptive cruise control that adjusts speed based on detected traffic elements and road conditions, integrating all subsystems into a cohesive, responsive, and efficient autonomous driving framework.

3.3 Chapter Summary

Convolutional Neural Networks (CNNs), YOLOv5, and Dlib represent powerful tools in the field of computer vision, each serving distinct but often complementary purposes. CNNs form the foundation of modern visual recognition systems, excelling in feature extraction and classification tasks due to their hierarchical learning of spatial features. YOLOv5 builds on this by offering real-time object detection with remarkable speed and accuracy, making it ideal for applications such as traffic sign recognition, license plate detection, and autonomous driving. On the other hand, Dlib is a versatile machine learning library, widely known for its robust facial recognition and landmark detection capabilities, often used in biometric and surveillance systems. Together, these technologies provide a comprehensive toolkit for developing intelligent vision-based systems capable of understanding and interacting with the real world.

Chapter 4

PROPOSED METHODOLOGY

Chapter 4

PROPOSED METHODOLOGY

This chapter outlines the systematic approach adopted for the development and implementation of our self-driving car project. Our project integrates multiple advanced technologies, including computer vision, machine learning, sensor data processing, and real-time control systems, to achieve autonomous navigation and decision-making. This section describes the step-by-step processes followed, starting from data collection and model training to hardware integration and real-world testing. By breaking down the complex goal of self-driving into manageable modules such as traffic sign recognition, lane detection, obstacle avoidance, and cruise control, we have developed a modular yet cohesive system architecture. Each subsystem was carefully designed, trained, validated, and integrated using both software (Raspberry Pi, ESP32, custom AI models) and hardware components (motors, sensors, actuators). The methodology provides detailed insights into the algorithms, tools, hardware setup, communication protocols, and cloud platforms employed to realize a fully functional prototype capable of autonomous operation.

4.1 Traffic Sign Recognition

In this module, the self-driving car identifies and classifies various traffic signs such as stop, left turn, and right turn signs. We trained a custom object detection model using **YOLOv5** on a curated traffic sign dataset to achieve real-time recognition. The camera mounted on the vehicle continuously captures frames, and the trained model processes these images to detect signs, sending appropriate signals for vehicle control and decision-making.

4.1.1 Dataset Collection

The foundation of any machine learning model is high-quality annotated data. For traffic sign detection, the dataset must encompass a diverse range of traffic sign types, lighting conditions, occlusions, weather variations, and viewing angles to simulate real-world driving scenarios. In this study, traffic sign images were sourced from publicly available datasets and, where necessary, supplemented by manually collected and annotated images. The primary datasets used include:

- GTSRB (German Traffic Sign Recognition Benchmark) from Kagel:
  - Contains over 60,000 images across 43 traffic sign categories.
  - Provides varied backgrounds, lighting, and weather conditions.
- Custom Dataset:
  - Captured using smartphone and dashcam footage in local environments.
  - Labelled manually using Roboflow and converted into YOLO format.

4.1.2 Data Annotation

All images were annotated using bounding boxes specifying the location and class of each traffic sign. Annotations followed the YOLO format(data.yaml):

<class_id> <x_center> <y_center> <width> <height>

All coordinates were normalized relative to the image dimensions. The dataset was then split into training (80%), validation (10%), and testing (10%) subsets.

Data Augmentation:

To increase model robustness and generalization, data augmentation techniques were applied:

- Random rotation
- Horizontal flipping
- Gaussian noise

These augmentations mimic real-world environmental variations that a self-driving vehicle may encounter.

4.1.3 Model Architecture and Training

The YOLOv5s (You Only Look Once v5 Small) architecture was selected for its balance of accuracy and real-time inference speed essential for deployment in self-driving systems. YOLOv5 is a single-stage object detector, optimized for performance, and implemented in PyTorch by Ultralytics.

Key characteristics include:

- Backbone: CSPDarknet53
- Neck: PANet for feature aggregation
- Head: YOLO detection layers with anchor boxes

Training Configuration:

- Input Image Size: 640x640 pixels
- Batch Size: 16
- Epochs: 800
- Optimizer: Stochastic Gradient Descent (SGD)
- Learning Rate: 0.01
- Loss Function: Combined objectness, classification, and localization losses

Training was conducted using the Ultralytics training script:

python train.py --img 640 --batch 16 --epochs 800 --data data.yaml --weights yolov5s.pt

Model performance was assessed using:

- mAP (mean Average Precision) @ IoU=0.5
- Precision and Recall
- F1-Score
- Inference Time per Frame

4.1.4 Deployment

The trained model achieved a mean Average Precision (mAP) of 91.75% on the validation set, with real-time inference capability (~45 fps) on an NVIDIA Jetson device. The model was further exported to ONNX and TensorRT formats for deployment on embedded systems.

The YOLOv5-based traffic sign detection model demonstrated high accuracy and low latency, making it suitable for real-time self-driving applications. The use of both public and custom datasets, combined with data augmentation, significantly enhanced the model's ability to generalize to diverse driving conditions. The fig 4.1 shows how YOLO works internally.

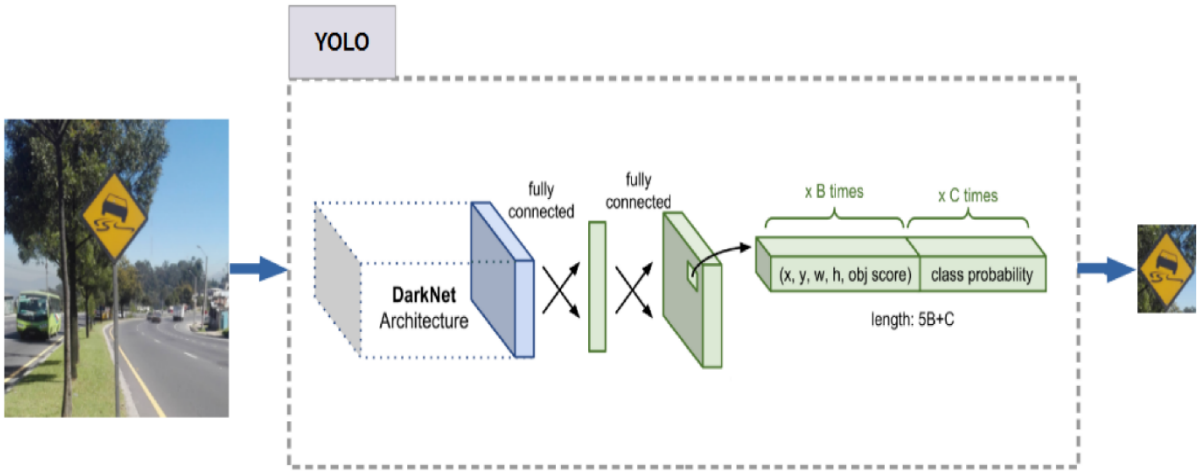

Figure 4.1: Working principle of YOLO v5 for traffic sign detection.

4.1.5 Realtime Detection using OpenCV and PyTorch

Real-time object detection is an essential feature for self-driving vehicles, enabling them to make immediate decisions based on their environment. Integrating a trained deep learning model with a real-time video stream is critical for recognizing and responding to traffic signs on the road. This section discusses the implementation of a real-time traffic sign detection system using OpenCV and PyTorch, leveraging a pre-trained YOLOv5 model.

The real-time detection pipeline consists of the following components:

- Camera Input: Live feed from a webcam or dashcam using OpenCV.
- Pre-processing: Resizing and normalizing frames to the model's expected input size.
- Model Inference: Running each frame through the YOLOv5 PyTorch model.

- Post-processing: Extracting bounding boxes, class labels, and confidence scores.
- Visualization: Overlaying detection results on the original video stream.

4.1.6 Implementation Details

(i) Dependencies

- Python 3.x
- PyTorch
- OpenCV
- Ultralytics YOLOv5 repository

(ii) Loading the Model

Model loading is a crucial step in any deep learning-based system, as it initializes the trained weights and configurations required for performing inference. In our self-driving car project, we used a custom-trained YOLOv5s model for traffic sign detection, saved in the PyTorch .pt format. Loading this model involves restoring the neural network architecture and its learned parameters so that it can accurately identify traffic signs in real-time video streams. We use torch.hub.load() or torch.load() depending on our setup. For example, using torch.hub.load('ultralytics/yolov5', 'custom', path='best.pt'), we load the model directly from the YOLOv5 GitHub repository, specifying our custom weights file (best.pt). This command initializes the network architecture, loads the trained weights, and prepares the model for inference. Alternatively, for more control or offline environments, we can load the model using model = torch.load('best.pt', map_location=device) and then set it to evaluation mode using model.eval(). Once loaded, the model can accept input images or video frames, process them, and return detection results such as bounding boxes, class labels, and confidence scores. Efficient model loading ensures quick startup, reliable inference, and seamless integration into the real-time perception pipeline of the vehicle. It forms the backbone of our object detection module, enabling the system to recognize and respond to traffic signs during navigation.

(iii) Capturing Live Video

Capturing real-time video is a fundamental component of our traffic sign detection system, enabling the vehicle to perceive and respond to dynamic road environments. In our implementation, we utilize a Raspberry Pi Camera (or a USB webcam) connected to a Raspberry Pi 4 to continuously capture video frames. This live video feed acts as the primary input for the YOLOv5-based traffic sign detection model. Using OpenCV, the camera is initialized with cv2.VideoCapture(0) or cv2.VideoCapture('/dev/video0'), depending on the hardware. The video capture loop reads each frame in real time, processes it through the trained YOLOv5 model, and displays the detection results instantly. Each frame is resized and normalized to match the model's input requirements (e.g., 640×640 resolution), and then passed to the model for inference. The model returns bounding boxes, class labels, and confidence

scores, which are drawn on the frame using OpenCV's drawing functions. This annotated frame is either displayed on a local monitor or stored for analysis. Real-time processing ensures the system can detect signs like "Stop," "Left Turn," or "Right Turn" as the vehicle moves, allowing for quick and intelligent decision-making. Frame rates are optimized using YOLOv5s for lightweight inference, which is ideal for embedded platforms like the Raspberry Pi. By capturing and analyzing live video streams, the vehicle gains the ability to monitor its surroundings continuously, ensuring safety and autonomy in real-world driving scenarios. This real-time feedback loop is essential for responsive and adaptive self-driving behaviour.

(iv) Performance and Optimization

1. Inference Speed

The system achieved an inference speed of approximately 45 FPS on a GPU (e.g., NVIDIA RTX 3060), and around 15–20 FPS on a CPU, depending on model size (yolov5s recommended for real-time use).

2. Latency Minimization Techniques

- Using smaller YOLO variants (v5s or v5n)
- Batch size = 1 for real-time single-frame processing
- TorchScript or ONNX conversion for deployment on edge devices
- Leveraging TensorRT(GPU) on NVIDIA Jetson devices

Our approach demonstrates the effectiveness of YOLOv5s for real-time traffic sign detection on low-power edge devices. By combining high-accuracy modelling with efficient deployment on Raspberry Pi and Jetson Nano, the system enables intelligent control of autonomous vehicles, proving to be a scalable solution for future smart mobility systems.

4.2 Lane Detection

Road Lane Detection requires to detection of the path of self-driving cars and avoiding the risk of entering other lanes. Lane recognition algorithms reliably identify the location and borders of the lanes by analysing the visual input. Advanced driver assistance systems (ADAS) and autonomous vehicle systems both heavily rely on them.

The lane detection system enables the vehicle to stay within the correct driving path. Using computer vision techniques such as *Canny edge detection, Hough transform*, and color thresholding, the vehicle detects lane markings from the road in real-time. The lane information is used to calculate the vehicle's steering angle, ensuring smooth navigation and maintaining lane discipline during autonomous driving. Here the figure 4.2 shows the systematic approach for lane detection.

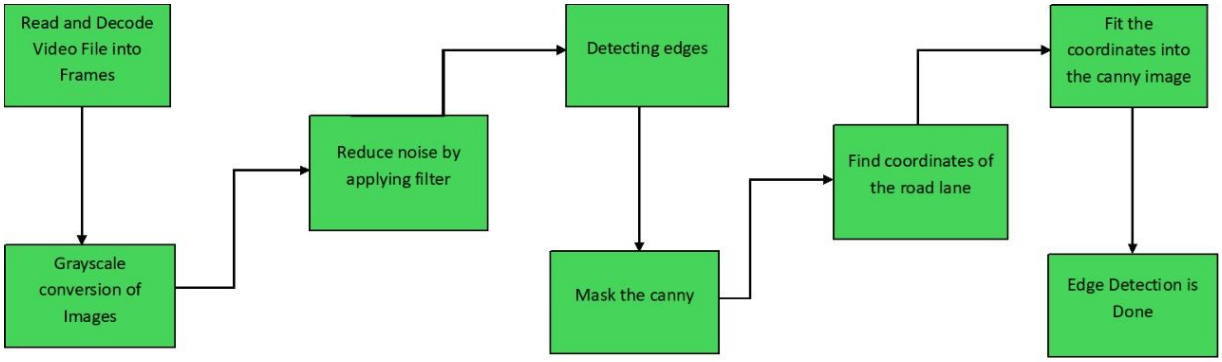

Fig 4.2: Step-wise workflow diagram for lane detection.

- **Capturing and decoding video file:** We will capture the video using VideoFileClip object and after the capturing has been initialized every video frame is decoded (i.e. converting into a sequence of images).
- **Grayscale conversion of image:** The video frames are in RGB format, RGB is converted to grayscale because processing a single channel image is faster than processing a three-channel coloured image.
- **Reduce noise:** Noise can create false edges, therefore before going further, it's imperative to perform image smoothening. Gaussian blur is used to perform this process. Gaussian blur is a typical image filtering technique for lowering noise and enhancing image characteristics. The weights are selected using a Gaussian distribution, and each pixel is subjected to a weighted average that considers the pixels surrounding it. By reducing high-frequency elements and improving overall image quality, this blurring technique creates softer, more visually pleasant images.

  Here is the formula for Gaussian blur:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- **Canny Edge Detector:** It computes gradient in all directions of our blurred image and traces the edges with large changes in intensity. To detect changes in pixel in real-time frames we use Canny Edge detection algorithm which is applies over a noise reduced image and gives the co-ordinates of edges in a matrix format.

  Equation behind Canny edge detection:

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2} Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

- **Region of Interest:** This step is to take into account only the region covered by the road lane. A mask is created here, which is of the same dimension as our road image. Furthermore, bitwise AND operation is performed between each pixel of our canny image and this mask. It ultimately masks the canny image and shows the region of interest traced by the polygonal contour of the mask.
- **Draw lines on the Image or Video:** After identifying lane lines in our field of interest using Hough Line Transform, we overlay them on our visual input (video stream/image).

4.2.1 Proposed Road Lane Detection System

Step 1: Capture real-time data

The system uses a front-facing camera mounted on the vehicle (or a simulated dashcam in testing) to capture continuous video frames of the road ahead.

Step 2: Frame Pre-processing

- Convert image to grayscale
- Apply Gaussian Blur to reduce noise
- Canny Edge Detection to extract edges

Step 3: Line Averaging and Lane Overlay

- Separate left and right lines based on slope
- Average them to get one line for each side
- Extrapolate to draw full lanes on the road

Step 4: Overlay and Visualization

Draw the detected lanes back onto the original image using OpenCV and calculate FPS.

Original Image:

Here we have an original image in figure 4.3 and we apply step by step techniques for lane detection.

Fig 4.3: Original image before lane detection.

And here are the step wise results after applying algorithms for lane detection:

Grayscale Blurred Image Canny edge Lane marking

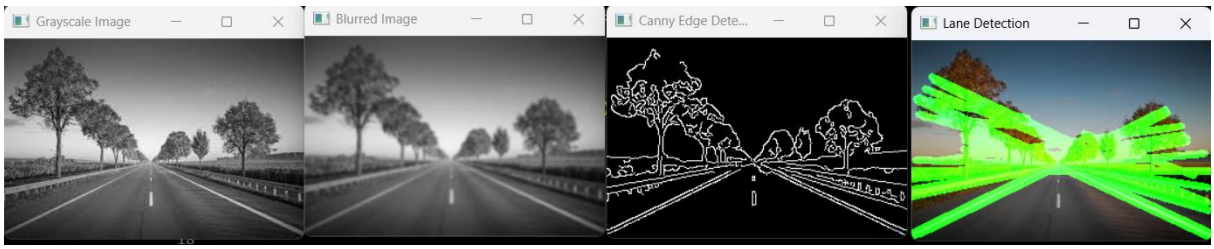

Fig 4.4: Procedural steps for lane detection.

4.2.2 System Implementation

a) Hardware Setup

- Front-facing USB/web camera or dashcam module
- Computer or embedded platform (e.g., Raspberry Pi, NVIDIA Jetson)

b) Software Stack

- Python 3.x
- OpenCV
- NumPy

Optional:

- ROS (Robot Operating System) for integration with autonomous stack
- TensorFlow/PyTorch if using deep learning-based lane detection

4.2.3 Limitations

- Sensitive to road noise such as cracks, shadows, or vehicles
- Traditional techniques may fail in complex intersections or curved roads
- Performance drops in night-time driving or wet roads unless enhanced

4.2.4 Improvements Required

To improve robustness and accuracy:

- Integrate deep learning-based lane detection models (e.g., SCNN, LaneNet)
- Use semantic segmentation to identify lanes more contextually
- Fuse with GPS and IMU data for localization and road curvature estimation
- Implement temporal smoothing using Kalman Filters or LSTMs for stability

Real-time lane detection using image processing provides an efficient and low-cost solution for road following in autonomous vehicles. While traditional computer vision techniques work well in controlled environments, further improvements through deep learning and sensor fusion are necessary for full-scale deployment in dynamic real-world conditions.

4.3 Obstacle Avoidance and Cruise Control

Obstacle avoidance ensures the safety of the vehicle by detecting and responding to objects in its path. Using ultrasonic sensors and vision-based object detection, the system detects nearby obstacles and adjusts the vehicle's speed or steering accordingly. The cruise control system maintains a steady speed when the path is clear, adjusting dynamically based on sensor feedback to prevent collisions. Obstacle avoidance is a fundamental aspect of smart car systems, enabling autonomous vehicles to navigate safely in dynamic environments. It involves the detection of obstacles using sensors and making real-time decisions to avoid collisions. In this project, the obstacle avoidance mechanism is implemented using an HC-SR04 ultrasonic sensor, which measures the distance to nearby objects by emitting ultrasonic waves and capturing their echoes. The sensor is interfaced with an ESP32 microcontroller, which processes the distance data and controls the motion of the vehicle accordingly.

The HC-SR04 sensor emits high-frequency sound waves that bounce off objects and return to the sensor, allowing it to calculate the distance to the object based on the time taken for the sound waves to return. Object detection is crucial for safe navigation in autonomous vehicles. While advanced systems utilize vision, LiDAR, and radar for perception, ultrasonic sensors offer a reliable and cost-effective solution for short-range detection, especially in low-speed environments such as parking or obstacle avoidance at low velocity. In self-driving cars, ultrasonic sensors are typically used for short-range detection, such as parking assistance and collision avoidance when the vehicle is moving slowly. They are often placed around the car's perimeter, usually inside the front and rear bumpers, and work in conjunction with other ADAS sensors like cameras, radar, and LiDAR.

Below the figure 4.5 shows existence of obstacle in a certain radius.

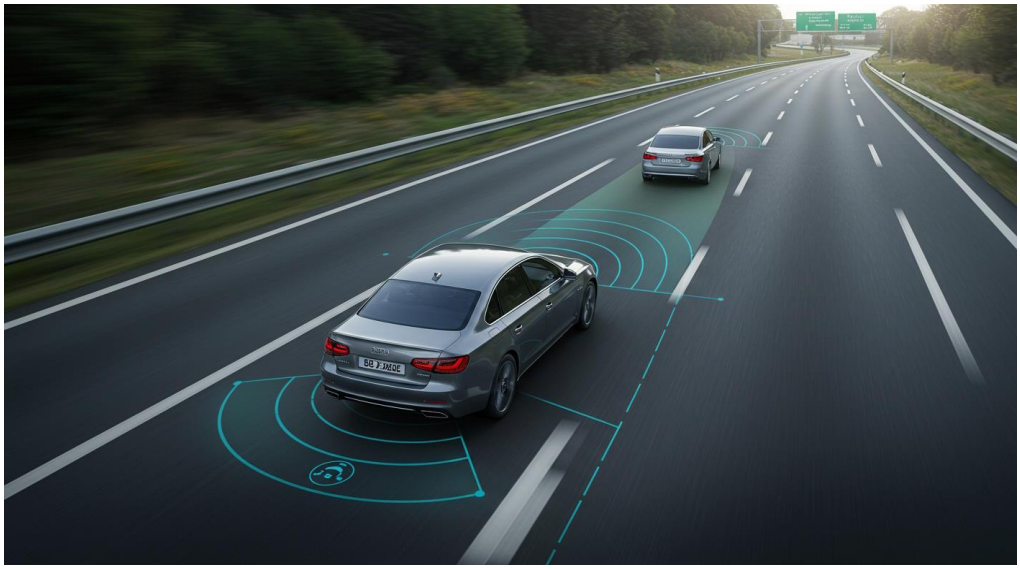

Fig 4.5: Obstacle detection**.**

The HC-SR04 ultrasonic sensor is one of the most commonly used modules for measuring distances with reasonable accuracy and minimal hardware complexity.

- **HC-SR04 Ultrasonic Sensor**: Used for short-range object detection in self-driving cars, emitting high-frequency sound waves to calculate distances to objects.234
- **Advanced Driver Assistance Systems (ADAS)**: Incorporate ultrasonic sensors for parking assistance and collision avoidance, enhancing driver safety.3
- **Self-Driving Cars**: Use ultrasonic sensors in conjunction with other sensors like cameras, radar, and LiDAR for comprehensive object detection.3
- **Limitations**: Ultrasonic sensors have limited range and may not detect small or fast-moving objects, requiring integration with other sensors for comprehensive coverage.

(b) Sensor Overview: HC-SR04

The HC-SR04 ultrasonic sensor measures distance by transmitting ultrasonic waves and measuring the time taken for the echo to return. It is ideal for detecting objects within a range of 2 cm to 400 cm.

(i) Specifications

Table 4.1: Specifications of Ultrasonic HC-04 Sensor:

Parameter	Value
Operating Voltage	5V DC
Detection Range	2 cm – 400 cm
Accuracy	±3 mm
Measuring Angle	~15°
Interface	4 pins (VCC, GND, TRIG, ECHO)
Response Time	~10 ms

(ii) Working Principle

- **Distance Measurement (Obstacle Detection)**

  The HC-SR04 ultrasonic sensor measures the distance between the vehicle and any object in front of it.

  Trigger Pin sends an ultrasonic pulse (8µs pulse).

  Echo Pin receives the reflected signal.

  Distance is calculated using the time delay between sending and receiving:

  Distance is calculated using the formula:

$$\text{Distance (cm)} = \frac{\text{Time (µs)} \times 0.0343}{2}$$

- **Obstacle Detection Logic**

  If the measured distance is below a safety threshold (e.g., 30 cm), the system detects an obstacle. Trigger a brake or stop command to the motor controller (L298N or ESC). Optionally trigger a buzzer or LED alert to indicate danger.

- **Cruise Control Mechanism**

  Cruise control maintains a constant speed unless an obstacle is detected.
  - If no obstacle is within the threshold:
    - Motor runs at preset speed (PWM control).
  - If an obstacle is detected:
    - Speed is reduced or motor is stopped based on proximity:
      - 10–30 cm: Slow down
      - <10 cm: Stop completely
  - Once the obstacle is cleared, system resumes normal speed.

- **Integration with Microcontroller**

  Sensor readings are processed using Arduino/ESP32. Output signals control motor speed via PWM or motor driver (L298N). System can be enhanced with OLED display or Bluetooth module for monitoring.

4.3.1 System Implementation

i) Hardware Integration

- **Microcontroller**: ESP32 (or any compatible board)
- **Sensor Placement**: Front bumper or side of the vehicle for close-range detection
- **Optional Components**: Buzzer or LED for alerts, relay for safety system

ii) Software Logic

We implementing software logic to the secondary controller ESP32 board using Arduino software. We set a distance threshold for object detection (that is 30cm in our case) and if any obstacle is present with in this radius, then the controller perform actuation.

4.3.2 Applications in Self-Driving Cars

- **Parking Assistance**: Detects nearby walls or other vehicles during parking.
- **Low-Speed Collision Avoidance**: Warns of obstacles when moving slowly in traffic.
- **Blind Spot Monitoring**: Helps detect objects not visible to cameras or LiDAR.
- **Backup and Reversing Safety**: Acts as a rear obstacle warning sensor.

4.3.3 Performance Evaluation

Testing Setup:

- Environment: Indoor and outdoor (daylight)
- Test surface: Solid wall and vehicle body
- Range: 5 cm to 250 cm
- Power Supply: 5V from Arduino

The HC-SR04 ultrasonic sensor provides a simple, cost-effective method for short-range object detection in self-driving vehicles. While it cannot replace more advanced sensors, it plays a vital supporting role in low-speed scenarios such as parking, reversing, or close-quarters navigation. Properly calibrated and placed, it enhances safety and reliability in autonomous systems. However, ultrasonic sensors have limitations. They are not suitable for detecting objects at high speeds or over long distances, as their range is limited to about 8 to 15 feet.3 Additionally, they may not be able to detect small or multiple objects moving at fast speeds.

4.4 Drowsiness Detection

Driver drowsiness detection adds a critical safety feature to the system. A real-time monitoring system based on **facial landmark detection** using machine learning tracks eye closure, blink rates, and head movements. If signs of drowsiness are detected, an alert mechanism is triggered to warn the driver, helping to prevent accidents caused by fatigue.

4.4.1 Environment Setup

Hardware Setup

- Front-facing USB/web camera or dashcam module
- Computer or embedded platform (e.g., Raspberry Pi, NVIDIA Jetson)

Software requirements

- Python 3.x
- OpenCV
- NumPy

4.4.2 Methodology:

Step 1: **Face and Landmark Detection**

The detection process begins with capturing live video feed from the webcam. Each frame from the stream is resized and converted to grayscale to reduce computational overhead and improve detection performance. The grayscale frame is passed to dlib's frontal face detector, which applies a Histogram of Oriented Gradients (HOG) + Linear SVM based model to detect the presence and bounding box of human faces. Upon successful detection of a face, the next crucial step is to detect facial landmarks using dlib's 68-point pre-trained facial landmark predictor. This model identifies key facial features such as the eyes, nose, mouth, jawline, and eyebrows. For drowsiness detection, only the eye region is of importance, which includes 6 landmark points for each eye (total 12 for both).

Step 2: **Eye Aspect Ratio (EAR) Computation**

This Driver Drowsiness Detection algorithm works by continuously analysing eye movements using facial landmarks. The Eye Aspect Ratio (EAR) is a lightweight yet powerful feature for detecting eye closure without requiring deep learning. By monitoring the EAR across multiple frames and comparing it to a threshold, the system distinguishes between normal eye activity and potential drowsiness. It is an effective, real-time solution suitable for embedded systems and plays a vital role in *driver safety*, especially in fatigue-prone scenarios such as long-distance or night driving. Its implementation is computationally efficient, requires no training, and provides an elegant example of geometry-based computer vision for real-world safety applications. The figure 4.6 shows how eye aspect ratio is calculated using the eye coordinates.

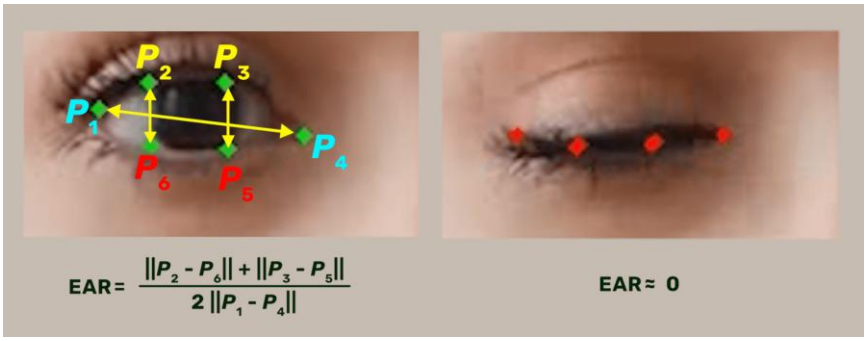

Fig 4.6: Eye aspect ratio calculation.

Step 3: **Temporal Monitoring and Drowsiness Detection**

Since occasional blinks are normal, a single low EAR reading is not sufficient to declare drowsiness. Instead, the system monitors the EAR over a sliding window of frames. If the EAR falls below a predefined threshold (typically around 0.25) for a continuous number of frames (e.g., 25 frames), the driver is considered drowsy. This approach filters out rapid eye blinks and short-term occlusions, ensuring that the system responds only to prolonged eye closures, a symptom strongly correlated with microsleep or fatigue.

Step 4: **Alert Mechanism**

Once the system detects that the EAR has remained below the threshold for more than the specified number of frames, it concludes that the driver is drowsy. As a result, it triggers an alert mechanism. The alert is both visual and auditory: a warning message is overlaid on the video feed, and a sound is played using the pygame.mixer module. This multimodal alert is crucial to ensure that even if the driver's eyes are closed or they're not paying attention to the display, the auditory cue can still catch their attention and potentially prevent an accident.

Step 5: **Threshold Tuning and Real-World Application**

The choice of threshold values for EAR and frame count is critical and usually determined empirically through experimentation and literature review. The threshold must be low enough to detect eye closure but high enough to avoid false positives from frequent blinking or squinting. The frame count should correspond to approximately 1–2 seconds of continuous eye closure to balance responsiveness with robustness. In real-world automotive environments, this system can be mounted on dashboards using Raspberry Pi or edge AI cameras, and integrated with vehicle control systems to issue alerts or trigger automatic braking if the driver remains unresponsive.

Here the figure 4.7 shows the overall architecture of driver drowsiness detection and alert mechanism.

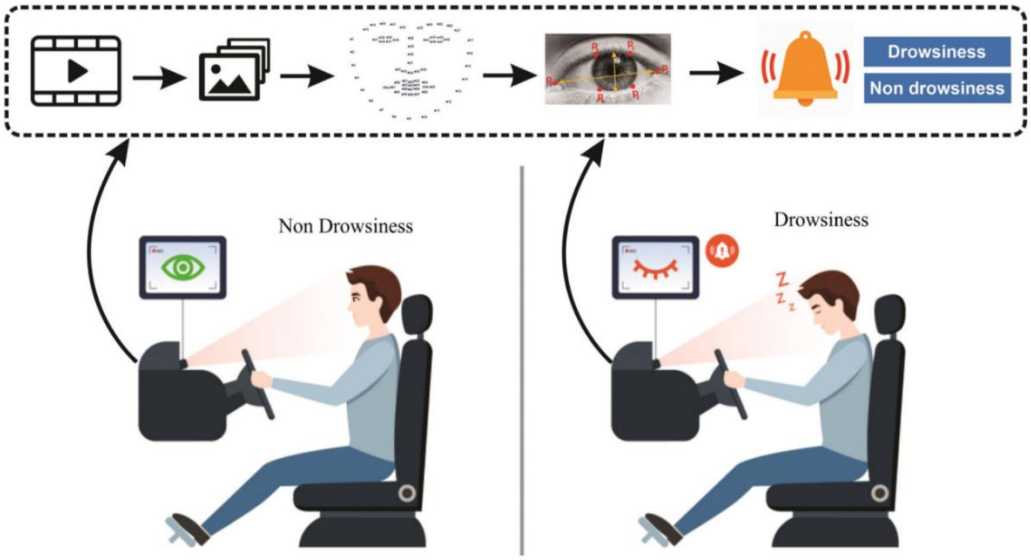

Fig4.7: Real-time drowsiness detection and alert mechanism.

4.4.3 Software Stack for Driver Drowsiness Detection System:

Here's a breakdown of the full software stack used:

Programming Language

- Python: Core language for implementation due to its simplicity and vast library support.

Computer Vision

- OpenCV: For video frame capture, image processing, and visualization.
- Imutils: Utility functions for easy image manipulation and resizing.

Facial Landmark Detection

- **Dlib**: For real-time facial landmark detection (68-point model).
- **face_utils (from imutils)**: To extract and process eye region landmarks.

Audio Feedback

- **pygame.mixer**: For playing an alarm sound when drowsiness is detected.

Mathematical Computation

- **scipy.spatial.distance**: To compute Euclidean distances between eye landmarks for EAR calculation.

Hardware Interface (Optional)

- **cv2.VideoCapture**: Captures real-time webcam feed or USB camera input.

Platform

- Runs on Windows/Linux/macOS; optimized version can be deployed on Raspberry Pi with a compatible camera module.

Driver drowsiness detection is a safety system designed to monitor and analyze a driver's eye activity in real-time to prevent accidents caused by fatigue. Using a camera and facial landmark detection, the system calculates the Eye Aspect Ratio (EAR) to determine whether the driver's eyes are closing or blinking abnormally. If the EAR drops below a specific threshold for a set duration, it indicates possible drowsiness, triggering visual and audio alerts. This technology significantly enhances road safety by providing timely warnings, reducing the chances of accidents due to reduced alertness, especially during long or night-time driving.

4.5 Remote Control Vehicle

For situations requiring manual intervention, the vehicle can be remotely controlled through a dedicated wireless interface. Using an **ESP32** and Bluetooth/Wi-Fi communication, commands from a mobile application or controller are sent to the vehicle, allowing remote navigation and control. This feature is especially useful for system testing, calibration, and emergency override scenarios.

Methodology:

4.5.1 Mobile Application for Manual Control via Bluetooth

In parallel with the development of the autonomous system for the self-driving car, an essential addition was made in the form of a custom Android mobile application. This application was developed using React Native for its flexibility in creating cross-platform apps and Kotlin to integrate native Android functionalities—specifically, the Bluetooth communication features required to connect with and send instructions to the ESP32 microcontroller. This application empowers users to operate the car in manual mode, offering critical support during testing, providing manual override capabilities, and acting as a key interaction interface in demonstrations and exhibitions.

4.5.2 App Design and Architecture

The design of the mobile app was kept minimal and user-friendly to accommodate a wide range of users, from engineers to students and casual users in demo sessions. The architecture of the app is split into two main modules:

(i) **Bluetooth Connection Screen**

This is the first screen displayed upon opening the app. It handles device discovery and pairing. Using the device's Bluetooth hardware, the app searches for available devices within range. A successful connection is confirmed visually with on-screen feedback,

and the app then maintains an active Serial Port Profile (SPP) link with the ESP32 until the user disconnects or the session times out.

(ii) **Manual Control Screen**

Upon successful connection, the user is taken to a control interface resembling a virtual joystick or remote controller. The snapshot of the control screen is given in figure 4.8. This screen includes button-based controls such as:

- Forward (1)
- Backward (2)
- Left (3)
- Right (4)
- Stop (0)

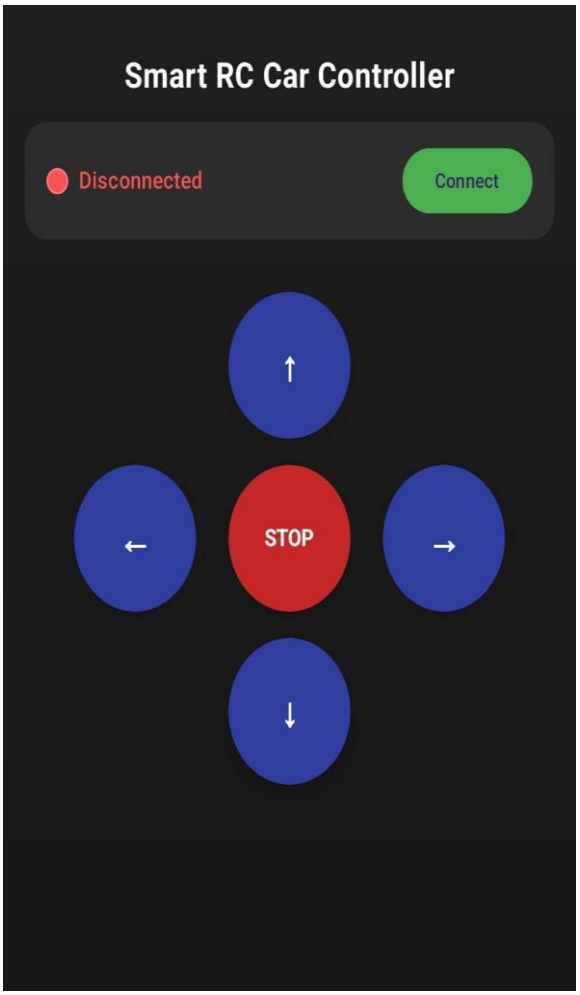

Figure 4.8: User interface of the Android application used to control the self-driving car.

Here the fig 4.9 shows the image of final setup of the vehicle.

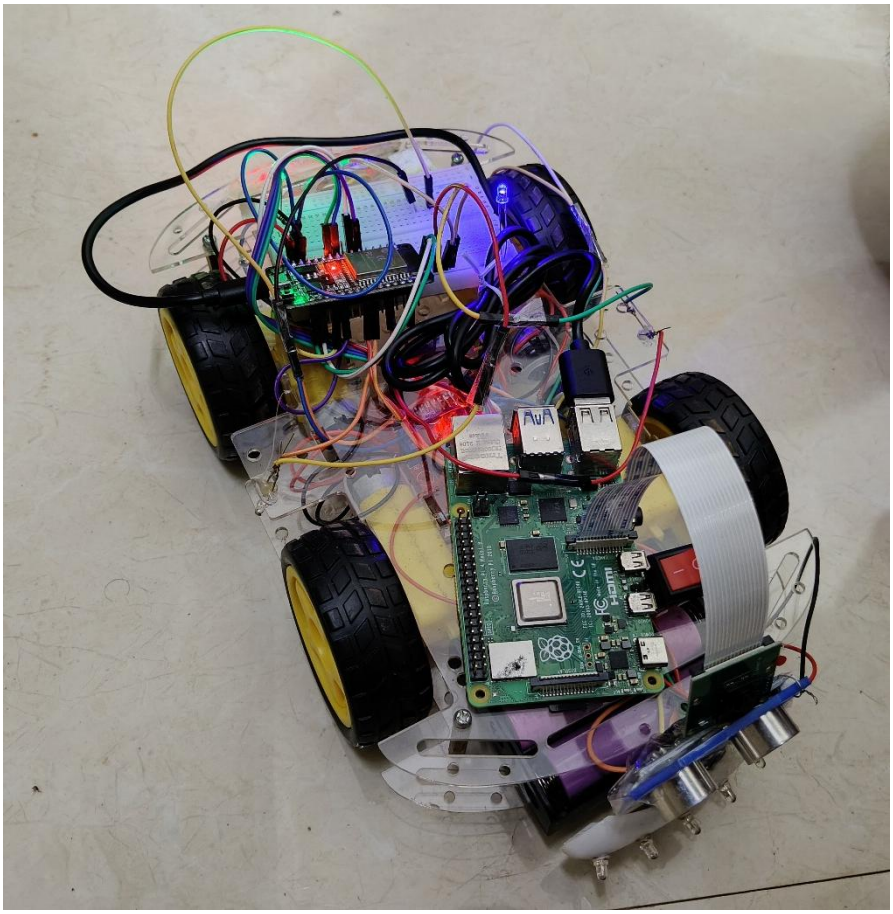

Figure 4.9: Prototype after complete simulation.

These controls are mapped to simple character codes, which are transmitted over the serial Bluetooth channel. When a user presses a button, the corresponding character is sent to the ESP32, which then translates the command to control signals for the L298N motor driver module, driving the car's motion.

4.5.3 Implementation and Communication Protocol

The Bluetooth communication follows the Serial Port Profile, which emulates RS-232 serial communication over Bluetooth. The ESP32 receives character commands via its UART port, where it processes them in real-time. The motor control logic written in the ESP32's firmware ensures that the commands correspond precisely to directional movements. Debouncing and command queuing techniques were implemented to prevent input flooding and ensure smooth operation.

4.5.4 Performance Evaluation

The application underwent rigorous testing under both indoor and outdoor environments. Key performance metrics observed were:

- **Bluetooth connection time:** Averaged 2–3 seconds after device discovery.

- **Command transmission delay:** Less than 150 milliseconds on average, ensuring near real-time responsiveness.
- **Operational range:** Effective within a 10–15 meters radius, consistent with Bluetooth's specifications.
- **Power efficiency:** Minimal battery drain observed on smartphones during extended usage due to optimized Bluetooth handling in Kotlin.

During testing, the app reliably controlled the car without data loss or noticeable lag. The commands executed promptly and precisely, giving the user a tangible sense of control. Multiple Android devices were tested to ensure cross-device compatibility, which further validates the app's robustness.

4.6 Alcohol Detection

In a self-driving car project (especially college-level or semi-autonomous prototypes), alcohol detection methodology typically refers to how the system detects if the driver is intoxicated and prevents driving if alcohol is detected.

In addition, alcohol consumption can lead to driver impairment, which is a major cause of car accidents around the world. Indeed, drinking alcohol before (or even while) driving decreases several of the driver's functional abilities, including tracking power, vision, concentration, reaction time, and proper speed control, all of which increase the risk of a crash.

Objective:

To design and implement a low-cost alcohol detection system that can:

- Detect the presence of alcohol in the driver's breath or surroundings.
- Prevent vehicle operation if alcohol is detected beyond a predefined threshold.
- Integrate with the existing hardware/software architecture of autonomous or semi-autonomous vehicles.

4.6.1 System Design and Implementation

(i) Sensor-Based Detection Approach

The primary method used in this system is gas-sensor-based alcohol detection, particularly using the MQ-2 or MQ-3 gas sensor.

Key Components:

- MQ-3 sensor: More specific to alcohol vapours than MQ-2.
- Microcontroller (e.g., Arduino or Raspberry Pi): Reads sensor data and handles logic.
- Relay module: Controls access to vehicle ignition or drive mode.
- Buzzer/display unit: Provides real-time feedback.

Working Principle:

The MQ sensor detects alcohol vapor in the driver's breath. If the vapor concentration exceeds a set threshold (e.g., 0.5V output from sensor), the system:

- Triggers an alert (visual/audio).
- Disables vehicle ignition or drive engagement.

(ii) Integration with Vehicle System

- For manual or semi-autonomous vehicles, the sensor can be placed on the dashboard or near the steering wheel to detect driver breath.
- For autonomous vehicles, the sensor system can log events, alert remote operators, or interface with the vehicle's main control unit (via CAN bus or GPIO).

Integrating alcohol detection in self-driving or semi-autonomous vehicles enhances overall road safety by ensuring the human operator is sober when needed. While current sensor-based approaches offer a simple and cost-effective solution, future systems can evolve toward smarter, multi-modal safety systems combining sensor data, behavioural analysis, and AI for more reliable detection.

4.7 Chapter Summary

In our self-driving car project, we integrated multiple computer vision and IoT functionalities to simulate autonomous driving behaviour. YOLOv5 was employed for real-time traffic sign detection, trained on a custom dataset including stop, left, and right signs, enabling the system to recognize and respond to road instructions. CNN models were used for additional classification tasks such as backup recognition or for fallback visual processing. For lane detection, OpenCV techniques processed live camera feeds from a Raspberry Pi 4 with an RPiCam Rev 3, helping the vehicle maintain proper alignment on the road. Drowsiness and alcohol detection modules were implemented using Dlib's facial landmark detection and custom image processing to monitor driver alertness. The ESP32 microcontroller controlled the car's motion and actuators (e.g., LEDs or motors) via an L298 motor driver, based on the interpreted vision data. Serial communication (PySerial) enabled real-time command transfer between the Raspberry Pi and ESP32. Additionally, system data, including detection results and sensor feedback, was transmitted to ThingSpeak Cloud for remote monitoring and analytics, providing a complete end-to-end smart mobility solution.

Chapter 5

SIMULATION RESULTS & DISCUSSIONS

Chapter 5
SIMULATION RESULTS & DISCUSSIONS

With the rapid advancement in Artificial Intelligence, Embedded systems, and Robotics, the transportation sector is undergoing a revolutionary transformation. The rise of autonomous and semi-autonomous vehicles is driving innovation in the field of Intelligent Transportation Systems (ITS). Among the many critical aspects of self-driving technology, the ability to recognize traffic signs, detect obstacles, and control vehicle speed dynamically lies at the core of safe and efficient navigation. The integration of these capabilities into a single system prototype forms the basis of this research project.

This thesis presents the design and implementation of a miniature self-driving four-Wheel-Drive (4WD) car simulation that replicates essential features of an autonomous vehicle. The system is built on an embedded hardware platform consisting of a Raspberry Pi 3, ESP32 microcontroller, Raspberry Pi Camera Module Rev 3, and an L298N motor driver, interfaced with DC motors for vehicle motion. The system integrates three key subsystems: Traffic Sign Recognition using YOLOv5s, Obstacle Detection using HC-SR04 ultrasonic sensor, and Cruise Control, working together to simulate autonomous decision-making in real-time.

One of the most vital abilities of a self-driving vehicle is its capacity to understand and respond to road signage. In this project, a traffic sign recognition system has been implemented using YOLOv5s, a state-of-the-art object detection algorithm known for its real-time performance and high accuracy. The model was trained on a custom dataset comprising essential traffic signs such as Stop, Left Turn, Right Turn, and Speed Limit. The YOLOv5s model processes live video frames from the Raspberry Pi Camera, detects signs in real-time, and classifies them with bounding boxes and labels. The small footprint and speed of the YOLOv5s model make it ideal for resource-constrained platforms like Raspberry Pi. Once a traffic sign is detected, the system makes an appropriate decision such as stopping the vehicle, turning, or adjusting the speed. This component of the project ensures that the vehicle adheres to road rules and mimics real-world driver behaviour.

To avoid collisions and navigate safely, the vehicle must be aware of obstacles in its path. This is achieved using the HC-SR04 ultrasonic sensor, which emits ultrasonic pulses and measures the time it takes for the echoes to return after hitting an object. This distance measurement is performed continuously, and the sensor is placed at the front of the vehicle to detect obstacles ahead. If an obstacle is detected within a predefined safety range (e.g., 10-30 cm), the system triggers a response ither reducing the speed or stopping the vehicle. This allows the system to simulate intelligent avoidance strategies similar to those used in modern adaptive cruise control systems.

The distance to the obstacle is calculated using the formula:

$$\text{Distance (cm)} = \frac{\text{Time } (\mu s) \times 0.0343}{2}$$

The cruise control subsystem is responsible for maintaining a steady speed when the path is clear and dynamically adjusting the speed in response to detected obstacles. The system uses inputs from the HC-SR04 sensor to determine whether the vehicle should continue at normal speed, slow down, or halt completely. For example:

- If no obstacle is within the threshold, the motor runs at a constant PWM speed.
- If an obstacle is detected within 30 cm, the system reduces the PWM signal to slow down.
- If the object is closer than 10 cm, the system stops the vehicle completely to avoid collision.

This adaptive response ensures smooth and safe navigation and demonstrates real-time control over the vehicle's speed based on environmental feedback.

The integration of all modules is carried out using a combination of Raspberry Pi and ESP32 microcontroller. The Raspberry Pi serves as the main controller, handling the camera feed, running the YOLOv5s model for traffic sign detection, and making high-level decisions. The ESP32 handles low-level motor control and ultrasonic sensing due to its real-time processing capabilities and efficient GPIO control. The L298N motor driver interfaces between the Raspberry Pi and the vehicle's motors, enabling forward and backward movement and speed modulation through PWM (Pulse Width Modulation). The system ensures smooth acceleration and deceleration based on input from the cruise control logic.

The system also supports real-time data monitoring by sending critical metrics such as detected signs, obstacle distances, and motor commands to the ThingSpeak IoT cloud platform. This enables remote tracking, performance analysis, and logging of vehicle behaviour over time an essential feature in modern connected vehicle systems.

5.1 Simulation Setup

The simulation setup for this autonomous driving project integrates multiple hardware and software components to mimic real-time perception, decision-making, and control in a controlled environment. The core components include a Raspberry Pi 3, a Raspberry Pi Camera Module Rev 3, an ESP32 microcontroller, an L298N H-bridge motor driver, four DC motors, and an HC-SR04 ultrasonic sensor, all mounted on a four-wheel drive robotic car chassis.

The Raspberry Pi acts as the brain of the system. It captures video input using the Pi Camera and processes the frames using a custom-trained YOLOv5s model for real-time traffic

sign recognition. The model, trained on a dataset including STOP, LEFT, and RIGHT signs, runs inference on each frame, and the results are interpreted to decide vehicle behavior.

For motion control, the Raspberry Pi communicates with the ESP32 over UART serial communication. Based on the detected sign, specific control commands (e.g., 'L' for left, 'S' for stop) are sent to the ESP32, which then drives the motors via the L298N motor driver. Each pair of motors on the left and right sides is connected to one H-bridge channel for differential motion control.

To enhance safety, an HC-SR04 ultrasonic sensor is mounted on the front of the vehicle for obstacle detection. The ESP32 continuously reads distance values and halts the vehicle if an obstacle is detected within a predefined threshold, simulating a basic cruise control mechanism. This simulation setup accurately replicates essential autonomous vehicle functions vision, control, obstacle detection, and actuation—making it ideal for real-world testing, validation, and educational demonstration of self-driving concepts.

5.1.1 Hardware Overview

- L298N Dual H-Bridge Motor Driver Module: Supports two channels for driving two motor pairs.
- DC Motors: 4 motors (2 left, 2 right), each connected to one side of the driver.
- Power Supply: External battery pack (7.4V or 12V Li-ion) to power motors; separate 5V supply for logic.
- Microcontroller: ESP32 receives serial commands and controls the motor driver.
- Raspberry Pi 3: Handles image processing and sends directional commands to ESP32 via UART.
- Micro SD Card: Greater than 16 GB recommended for load OS for Rpi
- HC-SR04 Ultrasonic Sensor: Detect presence of obstacle

5.1.2 Software Overview

- **OS:** Raspberry Pi OS (Bookworm / Bullseye)
- Arduino IDE
- Python 3.x
- OpenCV
- PyTorch + YOLOv5
- Flask (optional for API endpoint)
- PySerial (for UART to ESP32)

5.1.3 Procedure

- L298N H-Bridge motor simulation with a 4WD in ESP32 platform
- Setting up the Raspberry Pi environment with camera integration
- Running real-time traffic sign recognition using a trained YOLOv5s model
- Sending control commands to the ESP32 via UART (serial communication) based on detection results

5.1.3.1 L298N H-Bridge motor simulation with a 4WD in ESP32 platform

The L298N is a dual H-bridge motor driver module, capable of controlling the speed and direction of two DC motors independently. For a 4WD car platform, two motors on the left and two on the right are connected in pairs—meaning both left motors are wired together and connected to one channel of the L298N, and both right motors are wired to the second channel. The **ESP32**, with its built-in PWM and serial communication capabilities, acts as the controller for this driver module.

Connection Setup

(i) Power Connections:

- Connect the 12V battery to the +12V terminal of the L298N module.
- Connect GND of the battery to the GND of both L298N and ESP32.
- Connect the 5V output from L298N (after enabling the onboard voltage regulator via jumper) to power the ESP32, if required.

(ii) Motor Connections:

- Connect Motor A (left-side motors) to OUT1 and OUT2 of the L298N.
- Connect Motor B (right-side motors) to OUT3 and OUT4.

(iii) ESP32 Control Pins:

- Connect two GPIO pins from ESP32 to IN1 and IN2 for Motor A.
- Connect another two GPIO pins to IN3 and IN4 for Motor B.
- Optionally connect ENA and ENB to PWM-capable pins of ESP32 to control speed via analogWrite() (PWM).

Connections:

Power Supply:

- **Battery Positive** → 12V terminal on L298N
- **Battery Negative (GND)** → GND terminal on L298N and **also to ESP32 GND**

ESP32 to L298N:

- **GPIO 27** → IN1 (Left Motor A control 1)

- **GPIO 26** → IN2 (Left Motor A control 2)
- **GPIO 32** → IN3 (Right Motor B control 1)
- **GPIO 33** → IN4 (Right Motor B control 2)

(*Optional for speed control*)

- **GPIO 32** → ENA (PWM for left motors)
- **GPIO 33** → ENB (PWM for right motors)

Motors:

- **Motor Left Side (both motors in parallel)** → OUT1 & OUT2
- **Motor Right Side (both motors in parallel)** → OUT3 & OUT4

L298N Jumpers:

Make sure the **5V-EN jumper** is placed on the L298N so it provides 5V output to power ESP32 (optional if powering externally).

Simulation and Code Logic

(i) Initialization:
The ESP32 is programmed using the Arduino IDE or Platform IO with the appropriate GPIOs declared as outputs.

(ii) Basic Control Logic:

- To **move forward**, both motors are set in the same direction (e.g., IN1 = HIGH, IN2 = LOW; IN3 = HIGH, IN4 = LOW).
- To **move backward**, reverse both motor directions
- To **turn left** slow down the left motor while right motor moves forward
- To **turn right** slow down the right motor while the left motor moves forward
- To **stop**, all IN pins are set LOW or HIGH in opposing pairs

(iii) PWM Speed Control (Optional):

- PWM can be applied to ENA and ENB pins to vary the motor speed dynamically based on requirements like obstacle proximity or user input.
- Here the ENA (Enable motor A) and ENB (Enable Motor B) two pins used for speed control.

(iv)Serial/UART Integration (Advanced):

- ESP32 receives directional commands (like '1' for Forward, '2' for Backward, '3' for left, '4' for right and '0' for stop) from a Raspberry Pi or Bluetooth module via UART
- It parses the command and triggers respective motion functions

Testing and Debugging Steps

- Upload code ESP32 using Arduino IDE with correct board and port settings
- Power the system and observe the motor response to uploaded logic
- Verify directional accuracy and make sure the vehicle drives as expected
- Fine-tune the PWM values if using speed control

This simulation with ESP32 and L298N enables a cost-effective and scalable platform for robotics applications, including self-driving vehicle simulations. It facilitates essential motion control, integration with sensors, and external command processing via UART, making it highly suitable for modular autonomous vehicle design. The below diagram figure 5.1 shows the connection setup between ESP32 with L298 motor driver.

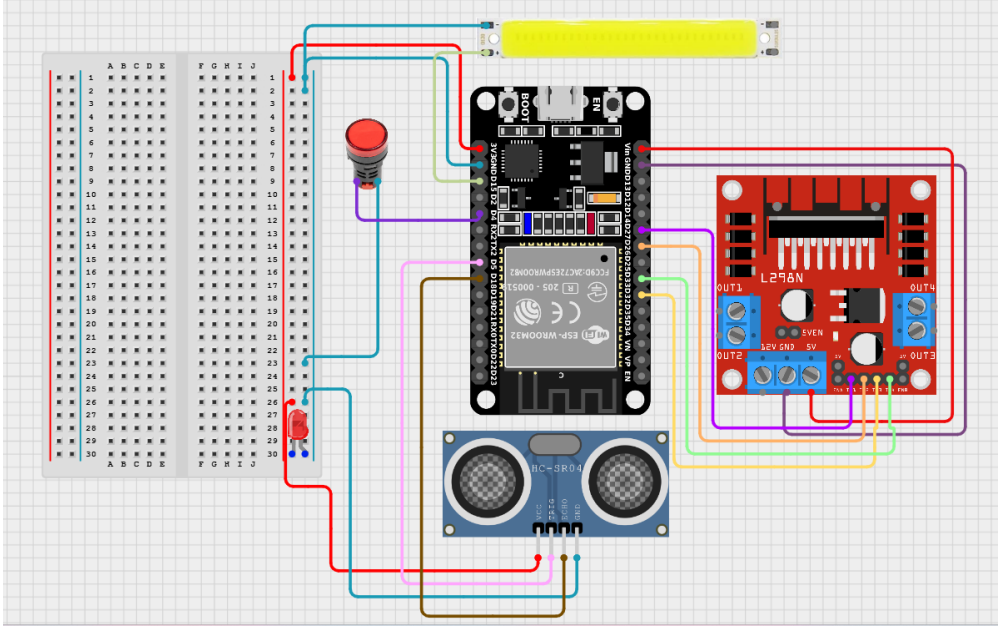

Fig 5.1 Connection Setup.

Results of Obstacle avoidance and cruise control:

Table 5.1: Distance measurement accuracy table:

Actual Distance (cm)	Measured Distance (cm)	Error (cm)
10	10.2	0.2
20	20.5	0.5
50	51.3	1.3
100	101.2	1.2
250	253.5	3.5

- **Accuracy**: Good for distances up to 200 cm
- **Limitations**: Struggles with soft surfaces and angled objects
- **Response time**: Fast enough for low-speed navigation

Advantages:

- **Low Cost**: Affordable and widely available
- Real-time obstacle avoidance
- **Easy to Integrate**: Minimal hardware and coding
- **Reliable at Close Range**: Effective where LiDAR/vision might fail
- **Low Power Consumption**: Suitable for embedded applications

Limitations:

- Not suitable for high-speed detection
- Performance degrades in noisy or echo-prone environments
- Narrow beam angle may miss wide objects
- Cannot classify objects (only measures presence and distance)

Future Enhancements:

- Use multiple ultrasonic sensors for 360° short-range coverage
- Integrate with AI-based perception systems for redundancy
- Fuse data from ultrasonic, vision, and LiDAR sensors using sensor fusion techniques
- Deploy on embedded platforms like Raspberry Pi or NVIDIA Jetson with real-time dashboard interfaces

5.1.3.2 Setting up the Raspberry Pi environment with camera integration

To simulate and implement real-time traffic sign recognition on a self-driving robotic platform, the first and most crucial step is to set up the Raspberry Pi environment properly. This setup involves configuring the Raspberry Pi operating system, enabling camera support, installing required software libraries, and verifying the camera feed. The integration of the camera module is essential for capturing real-time frames that are used as input for the object detection model (YOLOv5s).

Installing the Raspberry Pi OS:

- Download the official Raspberry Pi Imager tool from the Raspberry Pi website.
- Flash the latest version of Raspberry Pi OS (preferably Raspberry Pi OS Lite or Full) onto the MicroSD card.

- Insert the card into the Raspberry Pi and power it on.
- Complete the initial setup, including Wi-Fi configuration, localization, and system updates:

sudo apt update && sudo apt upgrade -y

Installing Required Dependencies

The project requires a Python environment with support for OpenCV, Torch, and other AI-related libraries.

Sudo apt install python3-pip python3-opencv libatlas-base-dev

pip3 install numpy opencv-python torch torchvision matplotlib

Camera Tuning for Better Results

In low-light or varying environmental conditions, it may be necessary to adjust the camera's ISO, white balance, or exposure settings. The libcamera tools provide the ability to do this:

libcamera-still -o test.jpg –shutter 50000 –gain 4.0 –awbgains 1.5,1.2

Setting up the Raspberry Pi with the camera module is a fundamental step in the overall simulation environment. It enables the system to acquire real-time visual input, which is processed by machine learning models for traffic sign detection and decision-making. This integration ensures that the vehicle perceives its environment accurately, simulating one of the critical components of autonomous driving systems.

5.1.3.3 Running real-time traffic sign recognition using a trained YOLOv5 Model

Real-time traffic sign recognition is an essential component in autonomous driving systems, enabling vehicles to perceive and respond to environmental cues such as stop signs, speed limits, and directional instructions. In this project, we employ a custom-trained YOLOv5s (You Only Look Once – Small) model for efficient and accurate detection of traffic signs using a Raspberry Pi and camera setup. YOLOv5s offers a balance between speed and accuracy, making it suitable for edge devices like the Raspberry Pi.

- **Model Deployment on Raspberry Pi**
  The trained model (best.pt) is transferred to the Raspberry Pi. Dependencies like PyTorch, OpenCV, and YOLOv5 repository are set up. Due to hardware limitations of the Pi, optimizations such as reducing image resolution and using lightweight models (YOLOv5s) are employed for smooth performance.

- **Capturing Real-Time Video Feed**
  The Raspberry Pi Camera Module Rev 3 captures live video. Each frame is passed to the YOLOv5s model in real time. The frame is pre-processed (resized, normalized), then converted into a PyTorch tensor for inference.

- **Detection and Interpretation**
  YOLOv5s returns bounding box coordinates, class indices, and confidence scores for detected traffic signs. Based on the class (e.g., class 0 = STOP, 1 =FORWARD), decisions are made for vehicle control logic.

- **Display and Decision Making**
  Detected signs are drawn on the frame using bounding boxes and labels. The final frame is displayed for monitoring. The detected label is also translated into control commands (e.g., 'S' for stop, 'L' for left) and sent to the **ESP32** microcontroller via **UART serial communication** to actuate motor responses.

- **System Feedback and Real-Time Action**
  The ESP32, upon receiving the command, drives the vehicle using the L298N motor driver according to the detected sign. This creates a closed-loop system where visual input directly results in mechanical output.

Advantages of Using YOLOv5s:

- **Speed**: Real-time inference at high FPS on lightweight hardware
- **Accuracy**: Detects small traffic signs effectively in various lighting conditions
- **Modularity**: Easy to retrain for different sign sets and scenarios
- **Edge-Friendly**: Compatible with Raspberry Pi without needing GPU

This real-time detection system forms a crucial part of the vehicle's autonomous decision-making module. It ensures that the vehicle follows road rules and responds appropriately to regulatory signs, thereby improving both safety and autonomy in smart vehicular systems.

5.1.3.4 Sending control commands to the ESP32 via UART (serial communication) based on detection results

In this project, seamless communication between the Raspberry Pi (which processes camera input and runs traffic sign recognition) and the ESP32 (which controls the motor driver for actuation) is critical. This communication is accomplished using UART (Universal Asynchronous Receiver/Transmitter) serial protocol, which allows the Raspberry Pi to send ASCII or byte-encoded instructions directly to the ESP32 based on the output of real-time traffic sign detection.

Working Principle:

The Raspberry Pi captures live video using the Raspberry Pi Camera Module. Each frame is passed through a YOLOv5s model trained to identify traffic signs such as STOP, LEFT TURN,

and RIGHT TURN. Once a sign is detected, the Pi executes a decision-making logic to assign a corresponding control command.

- **For example:**
    - If a "LEFT" sign is detected → command '3' is generated.
    - If a "RIGHT" sign is detected → command '4' is generated.
    - If a "STOP" sign is detected → command 'S' is generated.

These characters or encoded bytes are then transmitted to the ESP32 via the **TX (transmit)** pin of the Raspberry Pi to the **RX (receive)** pin of the ESP32. A ground connection is shared between both devices to complete the communication circuit.

Steps Involved in UART-Based Communication Setup

(i) Physical Connections:

- Raspberry Pi GPIO14 (TX) → ESP32 GPIO (RX)
- ESP32 TX (optional for debugging) → Raspberry Pi RX (GPIO15)
- Both devices must share a **common GND**.

(ii) Software Configuration on Raspberry Pi:

- Enable UART by modifying /boot/config.txt and disabling the serial console.
- Use Python libraries like serial (pyserial) to open the serial port:

(iii)Software Configuration on ESP32:

- Use Arduino IDE or PlatformIO to upload code to the ESP32.
- Listen for serial input using:

(iv)Decision Logic on Pi:

- Integrate the command-writing logic inside the YOLOv5 detection loop.
- Send the command immediately after detection using ser.write().

Advantages of UART in This Context

- Simple two-wire communication
- No need for complex protocols
- Real-time and low-latency data transmission
- Works reliably between Pi and ESP32 with minimal overhead

Using UART for control transmission ensures lightweight, fast, and dependable command execution. It enables the separation of processing (handled by Raspberry Pi) and actuation (handled by ESP32), making the system modular and easy to debug. This communication architecture is scalable for future additions such as GPS modules, IMUs, or cloud-based data

logging via the ESP32's built-in Wi-Fi. The figure 5.2 shows the overall communication between these two devices.

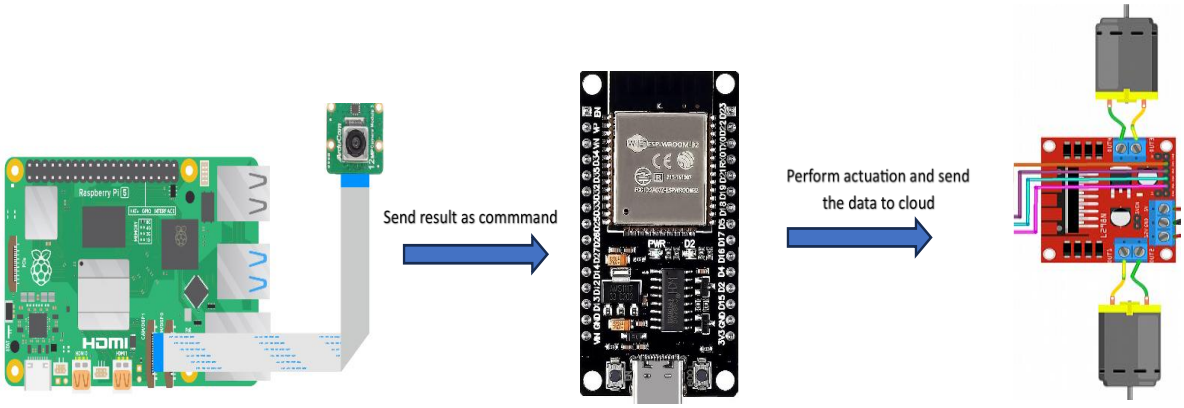

Fig 5.2: Communication of command between IoT devices.

5.2 RESULTS

5.2.1Traffic Sign Recognition Model Result

Several models have been developed for traffic sign detection in self-driving cars, each with varying levels of accuracy and real-time performance. One study proposes a convolutional neural network based on the YOLOv5 algorithm, which demonstrates high accuracy and meets real-time processing requirements crucial for self-driving systems. Another research uses a multi-task deep learning approach, which includes a region of interest (ROI) module and a multi-task learning (MTL) model to classify traffic signs accurately and make decisions for the self-driving car. In the YOLOv5 detector model were found to result in better performance than other models in terms of accuracy and processing time. The YOLOv5 model was evaluated during a test drive, and the results in table 5.2 showed its effectiveness in real-time traffic sign recognition.

Table 5.2: Accuracy results of Traffic sign recognition model:

Metric	Value (%)
Accuracy	97.4
mAP@0.5	91.75%
mAP of 0.5:0.95	74.08%
Inference Speed	~45 FPS (GPU)

i) Evaluation Results

The trained YOLOv5s model was evaluated on a validation and test set derived from a combination of the GTSRB dataset and a custom-collected dataset. Evaluation metrics were calculated to assess the model's detection accuracy, precision, and inference speed. The primary metrics used include:

BOX Graph:

In the context of machine learning, particularly object detection, a BOX graph typically refers to the graphical representation of the bounding box predictions for objects detected within an image. Each predicted bounding box is defined by its coordinates (x, y, width, height), and the graph shows the relationship between predicted bounding boxes and ground truth boxes, often used for evaluating detection performance.

Objectness Graph:

An Objectness graph visualizes the likelihood that a certain region in an image contains an object. It is a crucial component in object detection models like YOLO (You Only Look Once), where the model predicts an objectness score that indicates how likely it is that a given bounding box contains an object, rather than background noise. Objectness helps filter out irrelevant areas in the image, enhancing the model's accuracy.

Classification:

Classification refers to the process of categorizing input data into predefined labels or classes. For object detection, this term specifically refers to the task of identifying what type of object is present in the detected bounding box, such as distinguishing between a car, pedestrian, or traffic sign. The classification task assigns a label to the object based on its features.

Precision:

Precision is a metric used to evaluate the performance of a classification or detection model. It measures the proportion of true positive predictions (correctly identified objects) relative to all positive predictions made (both true positives and false positives). Mathematically, it is expressed as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Higher precision means fewer false positives, indicating that the model is making fewer mistakes in identifying objects.

Recall:

Recall, also known as sensitivity or true positive rate, is another evaluation metric. It measures the proportion of true positive predictions relative to all actual objects in the dataset (i.e., true positives and false negatives). In mathematical terms:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Higher recall indicates that the model is effectively identifying most of the relevant objects, with fewer missed detections.

Val Box (Validation Box):

The Val Box refers to the bounding box predictions during the validation phase of an object detection model's training. It is used to evaluate how well the predicted boxes match the ground truth boxes on the validation set. This helps assess the model's ability to localize objects within an image accurately.

Val Objectness (Validation Objectness):

Val Objectness refers to the validation of the objectness score predicted by the model during the evaluation phase. It measures how well the model distinguishes between object-containing regions and background regions. A higher objectness score typically correlates with higher accuracy in detecting the presence of an object.

Val Classification (Validation Classification):

Val Classification is the evaluation of the classification accuracy on the validation set. It checks how well the model classifies objects within the predicted bounding boxes, ensuring the detected object is correctly identified (e.g., identifying whether a car is present within the bounding box).

mAP@0.5 (Mean Average Precision at IoU threshold 0.5):

mAP@0.5 is a performance metric widely used in object detection. It calculates the average precision of the model at an Intersection over Union (IoU) threshold of 0.5. In simpler terms, it evaluates the precision of the model when the predicted bounding box overlaps with the ground truth bounding box by at least 50%. The mAP score is computed across all classes and represents the model's overall accuracy.

$$\text{mAP@0.5} = \text{Average of Precision at IoU threshold 0.5}$$

A higher mAP score at this threshold indicates better overall detection performance.

mAP@0.5:0.95 (Mean Average Precision at IoU thresholds from 0.5 to 0.95):

mAP@0.5:0.95 extends the mAP evaluation by averaging precision at multiple IoU thresholds, ranging from 0.5 to 0.95 (with a step size of 0.05). This provides a more robust measure of the model's detection capability, especially for stricter overlap requirements. It is considered a more comprehensive evaluation of the model's performance because it accounts for a variety of IoU thresholds, not just a single threshold.

$$\text{mAP@0.5:0.95} = \text{Average of Precision at IoU thresholds 0.5 to 0.95}$$

This metric is especially useful when comparing models for object detection tasks where both localization accuracy (IoU) and classification accuracy are critical across varying levels of overlap.

Here the figure 5.3 shows all the accuracy results graph for our traffic sign recognition model.

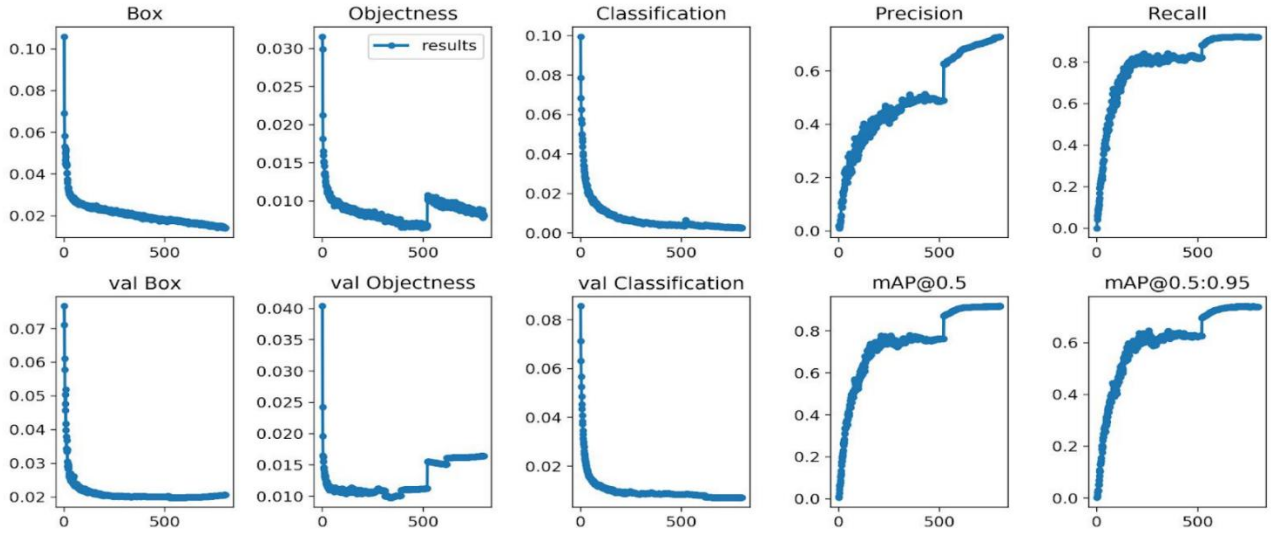

Fig 5.3: Evaluation results.

ii) Visual Results

The following figure shows sample detections from the test dataset under various conditions (e.g., daylight, shadows, motion blur):

Each detection includes a bounding box described in figure 5.4, label, and confidence score. The model demonstrated robustness in handling occlusions, varying lighting conditions, and partially obscured signs.

Fig 5.4: Detection of sign using bounding box

iii) Real-Time Performance

When deployed in a real-time pipeline using OpenCV and PyTorch, the model processed live video at an average of 30–45 frames per second (FPS) on an NVIDIA GPU. On CPU-only systems, performance averaged around 15–20 FPS, suitable for prototyping but not real-world deployment. For embedded systems like NVIDIA Jetson, further optimization via TensorRT was considered.

iv) Limitations and Challenges

- Detection accuracy for rare or worn-out signs was lower due to underrepresentation in the dataset.
- Small signs at long distances were occasionally missed due to resolution limits.
- Lighting changes (e.g., glare, night driving) affected precision slightly; future work includes implementing exposure normalization or HDR techniques.

Model Summary:

The model demonstrated strong performance in detecting common traffic signs with high accuracy and real-time inference capability. These results affirm the suitability of YOLOv5s for integration into the perception stack of autonomous vehicles. With further dataset expansion and hardware-specific optimization, the system can be reliably deployed for real-world driving scenarios.

5.2.2 Real-Time Lane detection

Table 5.3: Accuracy results of traffic sign recognition:

Metric	Result
Frame Rate	25–60 FPS (on GPU/PC)
Detection Accuracy	~92% (under clear roads)
Latency per Frame	~40ms
Adverse Condition Handling	Moderate (night, rain)

The system performed reliably on well-marked roads under good lighting. Performance degraded slightly under heavy shadows, faded lane markings, or rainy conditions.

5.2.3 Drowsiness Detection:

Typically, when using dlib for drowsiness detection (eye closure detection), you extract facial landmarks, especially eye landmarks, and compute metrics like EAR (Eye Aspect Ratio).

Table 5.4: Accuracy results of drowsiness detection:

Parameter	Typical Value
EAR threshold	0.25 – 0.30
Consecutive frames	20 – 25 frames
Landmark model used	shape_predictor_68_face_landmarks.dat
Detection modules	dlib frontal face detector + landmark predictor

5.2.4 Obstacle avoidance and Cruise control

To evaluate the performance of the remote-controlled vehicle and its autonomous systems, the following test scenarios were executed:

- **Obstacle Avoidance**: The RCV is tasked with avoiding obstacles such as cones or walls while navigating a predefined path autonomously.
- **Mixed-Control Tests**: The vehicle switches between manual control and autonomous navigation during a test, ensuring that both systems function seamlessly.

Table 5.5: Success rate of Obstacle avoidance:

Test Scenario	Autonomous Performance	Manual Control Performance
Obstacle Avoidance	85% success rate	100% success rate
Mixed-Control Switch	Seamless transition	Instant takeover on demand

- The vehicle showed high performance in obstacle avoidance and lane-following tests.
- The manual control system provided quick and reliable takeover when the autonomous system failed.
- The autonomous parking function was successful but required refinement for more complex environments (e.g., tight parking spaces).

Advantages of Using Remote Control in Self-Driving Car Development

- **Safety**: Operators can take control at any time, ensuring human intervention if the autonomous system encounters issues.
- **Flexibility**: Developers can test and debug autonomous algorithms without risking the vehicle in a fully autonomous mode.

- **Data Collection**: Remote control allows for real-world testing of autonomous systems, including edge cases and failure conditions, to help improve algorithms.

Limitations:

- **Limited Range**: The remote-control system has a limited operational range and can be prone to interference, especially in environments with many electronic signals.
- **Real-Time Performance**: Real-time processing may still face limitations, especially with onboard processing in vehicles.
- **Transition Between Control Modes**: Seamless and fast switching between manual and autonomous modes still needs optimization for better operator experience.

Improvements required:

- **Enhanced Sensor Suite**: Integrating additional sensors like radar, stereo cameras, and GPS for improved environmental awareness and navigation.
- **Edge Computing**: Deploying NVIDIA Jetson or similar platforms to handle high-performance computing tasks on the vehicle itself.
- **Remote Monitoring**: Adding telemetry and real-time data streaming for remote observation of the vehicle's performance during autonomous tests.
- **AI-powered Path Planning**: Implementing deep learning models to improve dynamic path planning, object detection, and decision-making.

5.2.5 Alcohol Detection

Results and Observations:

The prototype system was tested under various alcohol concentrations using controlled exhalation onto the sensor. The MQ-3 sensor successfully detected alcohol presence within 2–3 seconds and activated the safety mechanism in over 95% of valid cases.

Table 5.6: Accuracy result of Alcohol detection:

Alcohol Type	Ethanol (%)	Detection Time (s)
Beer	~5%	3
Wine	~12%	2
Vodka	~40%	<2
No Alcohol	0%	–

Advantages

- Low cost and easy to integrate into existing vehicles
- Non-invasive no need for breath blow tubes
- Fast response time suitable for real-time safety checks
- Scalable for fleet vehicles or ridesharing platforms

Limitations

- May trigger false positives from other volatile compounds (e.g., perfume, air freshener).
- Requires correct sensor placement near the driver.
- Cannot measure precise BAC (Blood Alcohol Concentration) only presence of alcohol vapours.
- Sensor sensitivity may degrade over time and require calibration.

5.3 Chapter Summary

This project showcases the practical integration of machine learning, embedded systems, and robotics to simulate real-world self-driving vehicle functionality. The traffic sign recognition powered by YOLOv5s provides intelligent perception, while obstacle detection and cruise control offer adaptive navigation and safety. The use of open-source tools and affordable hardware like Raspberry Pi and ESP32 makes this system ideal for educational research and future enhancements, such as GPS-based navigation, voice control, and cloud-based analytics.

Chapter 6

CONCLUSIONS & FUTURE WORKS

Chapter 6
CONCLUSIONS AND FUTURE WORKS

The evolution of autonomous vehicles represents one of the most transformative trends in the field of intelligent transportation systems. This thesis has explored the design and implementation of a self-driving car prototype that integrates several critical features of autonomous mobility. By leveraging the power of computer vision, embedded systems, machine learning, and IoT technologies, the developed system provides a low-cost, scalable, and educationally rich platform for understanding the real-world application of autonomous driving principles.

6.1 CONCLUSIONS

The prototype consists of four major modules: lane detection, traffic sign recognition, drowsiness and alcohol detection, and cruise control, all integrated with real-time monitoring and control capabilities. The system uses a Raspberry Pi 4 equipped with an RPiCam Rev 3 for image processing and runs a lightweight version of the YOLOv5s model to detect traffic signs such as stop, left, and right turn indicators. Simultaneously, the ESP32 microcontroller communicates with the Raspberry Pi and controls the hardware components like motors via the L298N motor driver. The system also utilizes ThingSpeak IoT cloud to log and monitor key data parameters like traffic sign detections, motor speeds, and alert signals from sensors.

The lane detection system, built using OpenCV and Python, detects lane markings in real-time and assists the car in navigating within a defined path. The traffic sign detection module, powered by a trained YOLOv5s model, identifies critical road signs and feeds them into the control logic for actuation. The drowsiness and alcohol detection module monitors the driver's alertness and presence of alcohol using facial features and sensor data, offering real-time alerts to ensure safety. The cruise control functionality is implemented using sensor feedback to maintain a stable speed and prevent collisions. Together, these modules form a semi-autonomous driving assistant that is capable of navigating, recognizing signs, reacting to driving conditions, and taking proactive safety measures.

Throughout the development, emphasis was placed on modularity, cost-efficiency, and real-time responsiveness. Open-source tools like Python, OpenCV, EasyOCR, PyTorch, and ThingSpeak were used to ensure extensibility and accessibility for further research and development. The hardware components were carefully chosen to balance performance and affordability, making this project suitable not only for academic purposes but also as a base model for future real-world applications. Despite the project's accomplishments, several limitations were encountered. The model's performance in low-light conditions was limited due to the use of a standard RGB camera. Object detection accuracy was dependent on the training dataset, and environmental changes (like rain, shadows, or reflective surfaces) occasionally caused recognition errors. Furthermore, processing constraints of the Raspberry Pi 4 occasionally introduced latency in detection and control response times. While these limitations

are common in prototype development, they highlight key areas for future enhancement, many of which have been addressed in the “Future Works” section.

Nevertheless, the outcomes of this project demonstrate the feasibility of developing a functional self-driving prototype using relatively low-cost components and open-source frameworks. The implementation journey offered significant insights into the challenges of real-time perception, decision-making, sensor integration, and control logic required for autonomous driving. Moreover, the cloud integration component showed promise in terms of monitoring system behavior, storing analytics, and potentially enabling remote diagnostics or control in future upgrades.

This project serves not only as a practical implementation of theoretical knowledge from the fields of computer vision, machine learning, and embedded systems but also as a stepping stone toward more advanced systems. Its modular design makes it highly adaptable to further research and academic experiments. Features like GPS-based path planning, V2X communication, LiDAR integration, night vision, federated learning, and migration to powerful edge platforms like Jetson Nano or Xavier are natural next steps that can be built upon this foundation.

In a broader context, projects like this contribute to the understanding and democratization of autonomous vehicle technologies. As the world moves toward intelligent transport systems, smart cities, and AI-powered mobility, equipping the next generation of engineers and developers with hands-on experience in autonomous systems becomes increasingly essential. By bridging theoretical learning with practical implementation, this project contributes to the ongoing transformation in how machines perceive, decide, and interact with the physical world.

To conclude, the self-driving car prototype developed in this thesis proves that with thoughtful design, efficient use of resources, and careful integration of software and hardware, it is possible to create a reliable and intelligent autonomous system. It opens up numerous possibilities for future innovation and sets a strong foundation for extending this work into more complex, scalable, and real-world-ready autonomous driving solutions.

6.2 FUTURE WORKS

The current self-driving car prototype successfully integrates core functionalities such as lane detection, traffic sign recognition using YOLOv5s, drowsiness and alcohol detection, cruise control, and IoT-based monitoring using ThingSpeak. The system, powered by a Raspberry Pi 4 and ESP32, serves as a solid foundation for autonomous navigation and control. However, there are numerous enhancements that can be made to improve its functionality, reliability, and scalability for real-world scenarios.

A major area for future development is the integration of GPS-based path planning to enable point-to-point navigation. Coupling GPS with algorithms like A* or Dijkstra and IMU data can offer better route tracking and decision-making. Additionally, enhancing obstacle detection through ultrasonic or LiDAR sensors can significantly improve safety, especially in dynamic or cluttered environments. Implementing SLAM (Simultaneous Localization and

Mapping) would allow the car to map unknown surroundings and localize itself in real-time. The current system performs best in well-lit environments. To ensure 24/7 usability, adding IR cameras or night vision techniques can help in low-light conditions. For improved driver safety, the drowsiness and alcohol detection module can be expanded with real-time facial landmark tracking and wearable biometric sensors for more precise behaviour monitoring. Introducing V2X (Vehicle-to-Everything) communication using MQTT or LoRa modules can enable interaction with nearby vehicles, infrastructure, and pedestrians, making the system more context-aware. Moreover, expanding the dataset and re-training the model using custom or regional traffic datasets would enhance detection accuracy for a wider range of signs and conditions. To handle more complex computations, especially for real-time object detection and tracking, the system can be migrated to powerful platforms like NVIDIA Jetson Nano, offering GPU acceleration. This also opens up the opportunity to use ROS (Robot Operating System) for modular design and scalability. In terms of cloud integration, while ThingSpeak provides basic data monitoring, transitioning to platforms like Firebase or AWS IoT would enable real-time dashboards, remote control, and predictive analytics. For example, the system could log and analyse driver behaviour, traffic patterns, or vehicle status to improve future performance.

6.2.1 Integration of GPS and Path Planning Algorithms

Currently, the car follows a lane and reacts to traffic signs and signals, but it lacks a full path planning module. Incorporating GPS-based navigation systems and algorithms like A* (A-star), Dijkstra's, or rapidly-exploring random trees (RRT) can enable point-to-point navigation. Integration with services like Google Maps or OpenStreetMap can further enhance real-world usability.

Future Enhancement:

- Integrate a GPS module (e.g., NEO-6M) with Raspberry Pi.
- Implement path planning and real-time rerouting algorithms.
- Combine GPS with Inertial Measurement Units (IMU) for better location tracking.

6.2.2 Object Detection and Avoidance Using LiDAR

While the current model avoids obstacles to some extent using basic computer vision logic, implementing LiDAR or ultrasonic sensors will drastically improve obstacle detection in all lighting conditions. This will enhance the reliability of the car in cluttered environments.

Future Enhancement:

- Add ultrasonic or infrared sensors for short-range obstacle detection.
- Integrate low-cost LiDAR modules (e.g., RPLiDAR A1) for accurate 360° mapping.

- Implement SLAM (Simultaneous Localization and Mapping) for navigating dynamic environments.

6.2.3 Night Vision and Low Light Adaptability

The current model relies on traditional RGB cameras, which perform poorly in low-light conditions. Using infrared (IR) cameras or enhancing the algorithm with night vision techniques will make the car operational 24/7.

Future Enhancement:

- Integrate an IR camera module or IR lighting with the current camera.
- Use deep learning models trained on nighttime datasets (like BDD100K-night).
- Apply image enhancement techniques for better feature extraction in low light.

6.2.4 Advanced Driver Monitoring System

The drowsiness and alcohol detection modules can be extended with more advanced behavioural and biometric analysis. Integrating face landmarks, yawning frequency, or heart rate sensors can improve the robustness of driver monitoring.

Future Enhancement:

- Use Mediapipe or Dlib for real-time facial landmark tracking.
- Integrate wearable sensors (like pulse or alcohol sensors) with ESP32.
- Implement alert systems like buzzer, vibration motors, or phone alerts.

6.2.5 Vehicle-to-Everything (V2X) Communication

The next logical step is to make the vehicle more context-aware by allowing communication between vehicles (V2V), infrastructure (V2I), and pedestrians (V2P). This can be facilitated using MQTT, 5G, or LoRa modules.

Future Enhancement:

- Implement MQTT protocol using ESP32 or Raspberry Pi for cloud messaging.
- Enable cross-vehicle data sharing like speed, direction, and hazard warnings.
- Explore RSU (Road Side Unit) simulation for smart traffic light coordination.

6.2.6 Incorporation of Autonomous Parking

Parking is a critical aspect of autonomous vehicles. Implementing a smart parking module that uses ultrasonic sensors and a reverse camera can automate parallel and perpendicular parking.

Future Enhancement:

- Add rear-view camera and wheel angle sensors.
- Design and test algorithms for space detection and movement planning.
- Use PID control for precise wheel turning and movement.

6.2.7 Upgrade to More Efficient Processing Platforms

While Raspberry Pi 4 provides adequate performance for a prototype, deploying the system in more complex environments would require platforms like NVIDIA Jetson Nano or Jetson Xavier, which support GPU-based acceleration.

Future Enhancement:

- Migrate codebase and models to Jetson Nano with TensorRT optimization.
- Leverage GPU-based inference for real-time object detection at higher FPS.
- Utilize ROS (Robot Operating System) for modular and scalable architecture.

BIBILOGRAPHY

1. Gupta, Abhishek & Guan, Ling & Khwaja, Ahmed. (2021). Deep Learning for Object Detection and Scene Perception in Self-Driving Cars: Survey, Challenges, and Open Issues. Array. 10. 100057. 10.1016/j.array.2021.100057.

2. Khanam, Rahima & Hussain, Muhammad. (2024). What is YOLOv5: A deep look into the internal features of the popular object detector. 10.48550/arXiv.2407.20892.

3. Parekh, D., Poddar, N., Rajpurkar, A., Chahal, M., Kumar, N., Joshi, G. P., & Cho, W. (2022). A Review on Autonomous Vehicles: Progress, Methods and Challenges. *Electronics (Switzerland)*, *11*(14), Article 2162. https://doi.org/10.3390/electronics11142162

4. Bimbraw, Keshav. (2015). Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology. ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings. 1. 191-198. 10.5220/0005540501910198.

5. Padmaja, B., Moorthy, C.V.K.N.S.N., Venkateswarulu, N. *et al.* Exploration of issues, challenges and latest developments in autonomous cars. *J Big Data* 10, 61 (2023). https://doi.org/10.1186/s40537-023-00701-y

6. Diwan, T., Anirudh, G. & Tembhurne, J.V. Object detection using YOLO: challenges, architectural successors, datasets and applications. *Multimed Tools Appl* 82, 9243–9275 (2023). https://doi.org/10.1007/s11042-022-13644-y

7. Mahmud, Tanjim & Ara, Israt & Chakma, Rishita & Barua, Koushick & Islam, Dilshad & Hossain, Mohammad & Barua, Anik & Andersson, Karl. (2023). Design and Implementation of an Ultrasonic Sensor-Based Obstacle Avoidance System for Arduino Robots. 264-268. 10.1109/ICICT4SD59951.2023.10303550.

8. Ziębiński, Adam & Cupek, Rafał & Grzechca, Damian & Chruszczyk, Lukas. (2017). Review of advanced driver assistance systems (ADAS). AIP Conference Proceedings. 1906. 120002. 10.1063/1.5012394.

9. Jayakumar, Dontabhaktuni & Peddakrishna, Samineni. (2024). Performance Evaluation of YOLOv5-based Custom Object Detection Model for Campus-Specific Scenario. International Journal of Experimental Research and Review. 38. 46-60. 10.52756/ijerr.2024.v38.005.

10. Ultralytics YOLOv5: https://docs.ultralytics.com/yolov5/, accessed on 13th April,2025

11. Raspberry Pi: https://www.raspberrypi.com/news/deploying-ultralytics-yolo-models-on-raspberry-pi-devices/, accessed on 7th April,2025

12. Python OpenCV: https://pypi.org/project/opencv-python/#description, accessed on 30 March,2025

13. Python PyTorch: https://pypi.org/project/torch/#description, accessed on 2nd March,2025

14. Mahamkali, Naveenkumar & Ayyasamy, Vadivel. (2015). OpenCV for Computer Vision Applications.

15. A. Sharma, J. Pathak, M. Prakash and J. N. Singh, "Object Detection using OpenCV and Python," *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, Greater Noida, India, 2021, pp. 501-505, doi: 10.1109/ICAC3N53548.2021.9725638.

16. Wang, Chien-Yao, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen and Jun-Wei Hsieh. "CSPNet: A New Backbone that can Enhance Learning Capability of CNN." *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2019): 1571-1580.

17. D. Zhang, J. Li and Z. Shan, "Implementation of Dlib Deep Learning Face Recognition Technology," 2020 International Conference on Robots & Intelligent System (ICRIS), Sanya, China, 2020, pp. 88-91, doi: 10.1109/ICRIS52159.2020.00030

18. Boyko, Nataliya & Basystiuk, Oleh & Shakhovska, Natalya. (2018). Performance Evaluation and Comparison of Software for Face Recognition, Based on Dlib and Opencv Library. 478-482. 10.1109/DSMP.2018.8478556.

19. Xu, Baoxun & Ye, Yunming & Nie, Lei. (2012). An improved random forest classifier for image classification. 2012 IEEE International Conference on Information and Automation, ICIA 2012. 795-800. 10.1109/ICInfA.2012.6246927.

20. Nayyar, Anand & Puri, Vikram. (2015). Raspberry Pi-A Small, Powerful, Cost Effective and Efficient Form Factor Computer: A Review. International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE). 5. 720-737.

21. J. Marot and S. Bourennane, "Raspberry Pi for image processing education," 2017 25th European Signal Processing Conference (EUSIPCO), Kos, Greece, 2017, pp. 2364-2366, doi: 10.23919/EUSIPCO.2017.8081633.

22. Abbas, Aiman & Peerzada, Pirah & larik, wasi. (2022). DC Motor Speed Control Through Arduino and L298N Motor Driver Using PID Controller.

23. N. N, B. S. Krishnaprasad and S. M. L. J, "Real-Time Alcohol Detection and Response System with Arduino and MQ-3 Sensor Integration," 2023 7th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Kirtipur, Nepal, 2023, pp. 1095-1103, doi: 10.1109/I-SMAC58438.2023.10290403.

24. Abdulkhaleq, Nadhir Ibrahim & Hasan, Ihsan & Abdul, Nahla & Salih, Jalil. (2020). Investigating the resolution ability of the HC-SRO4 ultrasonic sensor Investigating the resolution ability of the HC-SRO4 ultrasonic sensor. IOP Conference Series: Materials Science and Engineering. 745. 10.1088/1757-899X/745/1/012043.

25. A. Maier, A. Sharp and Y. Vagapov, "Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things," 2017 Internet Technologies and Applications (ITA), Wrexham, UK, 2017, pp. 143-148, doi: 10.1109/ITECHA.2017.8101926.

26. Ansari, Mohd & Kurchaniya, Diksha & Dixit, Manish. (2017). A Comprehensive Analysis of Image Edge Detection Techniques. International Journal of Multimedia and Ubiquitous Engineering. 12. 1-12. 10.14257/ijmue.2017.12.11.01.

27. Ghael, Hirak. (2020). A Review Paper on Raspberry Pi and its Applications. 10.35629/5252-0212225227.

28. Chang, Yeong-Hwa, Feng-Chou Wu, and Hung-Wei Lin. 2025. "Design and Implementation of ESP32-Based Edge Computing for Object Detection" *Sensors* 25, no. 6: 1656. https://doi.org/10.3390/s25061656

29. Bansal, Manav, Face Recognition Implementation on Raspberrypi Using Opencv and Python (2019). International Journal of Computer Engineering and Technology 10(3), 2019, pp. 141-144.

30. Xu, Yang & Zhang, Ling. (2015). Research on Lane Detection Technology Based on OPENCV. 10.2991/icmeis-15.2015.187.

31. Rajesh Kannan Megalingam, Kondareddy Thanigundala, Sreevatsava Reddy Musani, Hemanth Nidamanuru, Lokesh Gadde, Indian traffic sign detection and recognition using deep learning, International Journal of Transportation Science and Technology, Volume 12, Issue 3, 2023, Pages 683-699, ISSN 2046-0430, https://doi.org/10.1016/j.ijtst.2022.06.002.