**ADAS**

Almost all vehicle accidents are caused by human error which can be avoided with Advanced Driver Assistant System also known as ADAS. The role of ADAS is to prevent deaths and injuries by reducing number of car accidents and the serious impact of those that cannot be avoided. The ADAS system are passive and active safety system designed to remove the human error when operating vehicles of many types. The ADAS system uses advanced technology to assist the driver during driving and thereby improve driver performance. It uses various sensor in the vehicle such as radar and cameras to perceive the world around it and then either provides information to the driver or takes automatic actions based on what it perceives. The modernization of ADAS applications is the first step towards realizing autonomous vehicles.

**How the ADAS system works?**

Self-driving cars use a variety of technologies to gain 3600 vison both near and far. The ADAS system consists of sensors various chips called system on a chip interface and a powerful computer processor that integrates all the data and makes decision in real-time. The implementation of cameras in vehicles involves a new AI function that uses Sensor fusion to identify and process objects. Sensor fusion is similar to how the human brain process the information. It combines large amount of data with the help of image and object recognition software, ultrasonic sensor, LIDER and RADAR. This technology can physically respond faster than a human driver ever could.

**What ADAS do that human can’t?**

The ADAS equipped vehicles have an array of advanced sensors that augment the eyes, ears and decision making of the human driver. We know well that a human driver cannot see in the dark very well but the RADAR can. A human driver cannot determine what’s behind the car but SONAR sensor can. A human driver cannot see all the direction at a time but cameras and LIDAR sensor can.

**ADAS Levels:**

Advanced Driver Assistance System Levels are technical features that are intended to improve vehicles safety. They are categorized into different levels based on the amount of automation and the scale provided by the society of Automative Engineers. The ADAS system can be divided into six levels.

**In Level 0** the driver is entirely responsible for managing vehicles including steering, breaking, accelerating and slowing down.

**In Level 1** the vehicle has at least one driver support system that provides assistance in for example Adaptive cruise control, which maintains a safe following distance behind traffic ahead without intervention from the driver.

**In Level 2** the vehicle can perform steering control and acceleration. In this driver can monitor all tasks and can take control at any time.

**A level 3** ready autonomous vehicle is capable of driving itself in particular conditions during which it will take control of all safety critical systems nonetheless when the system asks it the driver is supposed to take control.

**In level 4** we can take an app while riding in the vehicle. Level 4 driving automation technology is for use in driverless taxies and for transportation services such as vehicle will be programmed from point A to point B and restricted to specific geographic boundary by geofencing technology.

**In level 5** the vehicle performs all driving tasks under all driving tasks under all conditions. In this level zero human attention and interaction is required.

1. **Traffic Sign Recognition (TSR):**

Traffic sign recognition is a technology by which vehicle can recognize the traffic signs on a road, such as speed limits, turn ahead, left and right turn etc. For this technology a forward sensing camera is used to capture the image frame from the real-time video and detect the sign by using Machine learning and deep learning technologies.

For real-time intelligent transportation systems, fast and accurate traffic sign recognition plays a crucial role in autonomous driving. The proposed model leverages the efficiency of the YOLOv5s (You Only Look Once, version 5 small) architecture for real-time traffic sign detection, trained on the widely used German Traffic Sign Recognition Benchmark (GTSRB) dataset. This model is specifically optimized for edge devices like Raspberry Pi and NVIDIA Jetson Nano, enabling low-latency inference and intelligent decision-making on embedded systems that control vehicle navigation and response. YOLOv5s is ideal due to its balance between speed and accuracy, and our experiments demonstrate that the model, trained over 800 epochs, achieved a classification accuracy of 97.4%, making it highly reliable for deployment in dynamic environments such as roads and highways.

The development process began with data preparation using the GTSRB dataset, which contains thousands of labelled images representing a wide variety of German traffic signs captured in real-world scenarios. These images were first pre-processed by resizing and augmenting them to enhance variability and improve model generalization. Next, we converted the dataset into the YOLO format, which involves creating annotation text files for each image with normalized coordinates of bounding boxes and corresponding class IDs. Once the data was ready, we configured the training pipeline by modifying the YAML configuration file to include custom class names, the number of classes, and the path to the training and validation datasets. We selected YOLOv5s due to its lightweight architecture, allowing faster inference with minimal computational load—ideal for Raspberry Pi and Jetson Nano deployment.

Model training was conducted using PyTorch on a high-performance GPU setup, training the model for 500 epochs. During training, the model continually optimized its parameters using the Stochastic Gradient Descent (SGD) optimizer and calculated loss based on objectness score, class probability, and bounding box regression. After training, the model's performance was evaluated on a separate validation set, where it achieved 97.4% accuracy, confirming its robustness and ability to generalize to unseen images. The final trained model was exported as a .pt file, which is compatible with PyTorch and can be easily loaded for inference.

For deployment, the .pt file was transferred to a Raspberry Pi 4 and Jetson Nano setup, where we installed a minimal Python environment with OpenCV and PyTorch libraries optimized for ARM architectures. The real-time detection pipeline uses frames captured from a connected camera, which are passed to the YOLOv5s model. The model detects and classifies traffic signs in each frame, drawing bounding boxes and labels in real-time. Based on the detected sign, control commands are sent to the vehicle's onboard microcontroller (such as Arduino or ESP32) using serial communication or GPIO signalling to trigger actuators like motors or LED indicators. This seamless end-to-end system allows the vehicle to interpret road signs dynamically and act accordingly, such as slowing down, stopping, or turning, making it suitable for self-driving applications and smart traffic systems.

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.

1. **Lane Detection:**

This technology proactively keeps the vehicle centered within a lane it is travelling. It utilizes automatic steering functionality to make constant adjustments based on road marking information from the front mounted camera. It also used to send warnings to driver if vehicle is going out from the lane.

Lane detection is a fundamental task in autonomous driving systems. Accurate lane detection ensures that a vehicle remains within its driving lane, enabling path planning, collision avoidance, and adaptive cruise control. The first stages of lane detection involve preparing the input image through a series of computer vision techniques: grayscale conversion, Gaussian blurring, edge detection via the Canny algorithm, and defining a Region of Interest (ROI). Each of these steps contributes to isolating relevant lane markings from other visual data. Below is a detailed technical walkthrough and analysis of these steps.

1. **Colour Space Conversion: RGB to Grayscale**

Original video frames from the car's onboard camera are typically in RGB (Red, Green, Blue) format. However, lane markings are primarily distinguished by contrast rather than colour. Hence, colour information becomes redundant. The RGB image is converted to grayscale using a weighted average method:

Gray = 0.2989 x R + 0.5870 x G + 0.1140 x B

This formula emphasizes green, to which the human eye is more sensitive. The grayscale image reduces computational complexity (from 3 channels to 1) and allows subsequent edge-detection algorithms to focus on intensity gradients rather than chromatic variations.

Python (OpenCV):

*gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)*

Why we need to convert the real-time frame to grayscale

* Improves edge-detection accuracy
* Reduces computational overhead, which is critical for real-time applications on embedded devices like Raspberry Pi

1. **Gaussian Blurring:**

Real-world images often contain noise due to lighting variations, shadows, or texture on the road. To reduce noise and false edges, a Gaussian blur is applied. It uses a Gaussian kernel to smooth the image by averaging neighbouring pixel values, giving higher weight to central pixels.

Mathematical Formulation:

G(x, y) = (1 / 2πσ^2) \* e^(-(x^2 + y^2) / 2σ^2)

Python (OpenCV):

*blur = cv2.GaussianBlur(gray, (5, 5), 0)*

Why we need Gaussian blurring

* reduces high-frequency noise
* Ensures Canny edge detection does not identify irrelevant edges
* Maintains the structure of important features like lane lines

1. **Canny Edge Detection:**

The Canny algorithm is a multi-stage edge detection technique. It detects rapid intensity changes (edges), which correspond to boundaries of objects—in this case, lane lines.

Steps involved:

* Gradient calculation (Sobel filter)
* Non-maximum suppression (keeps thin edges)
* Double thresholding (filters weak and strong edges)
* Edge tracking by hysteresis (connects strong and weak edges)

Python (OpenCV):

*edges = cv2.Canny(blur, threshold1=50, threshold2=150)*

Why we need Canny edge detection:

* Detects lane line edges sharply
* Thresholds can be tuned depending on lighting and environment
* Works well in combination with Gaussian blur

1. **Region of Interest (ROI) Masking**

Not all edges detected by Canny are relevant. For instance, edges from trees, buildings, or other cars are irrelevant to lane detection. Thus, we apply a mask to retain only the bottom portion of the image where lanes typically appear—forming a trapezoidal ROI.

Python (OpenCV):

*mask = np.zeros\_like(edges)*

*height, width = edges.shape*

*polygon = np.array([[*

*(int(0.1 \* width), height),*

*(int(0.45 \* width), int(0.6 \* height)),*

*(int(0.55 \* width), int(0.6 \* height)),*

*(int(0.9 \* width), height),*

*]], np.int32)*

*cv2.fillPoly(mask, polygon, 255)*

*roi = cv2.bitwise\_and(edges, mask)*

Why we need to find region of interest:

* Reduces false positives
* Focuses computation on road region
* Improves performance in real-time systems by processing only essential data

1. **Object detection and Adaptive Cruise Control (ACC):**

Adaptive cruise control is designed to help road vehicles to maintain a safe following distance and stay within the speed limit. This system automatically adjusts to car’s speed so drivers don’t have to reduce speed manually. Advanced cruise control can automatically accelerate slow down and sometimes stop the vehicle depending upon the action’s other objects in the immediate area. It is constructive on the highway where driver drivers can find it challenging to monitor their speed and other cars over a long period.

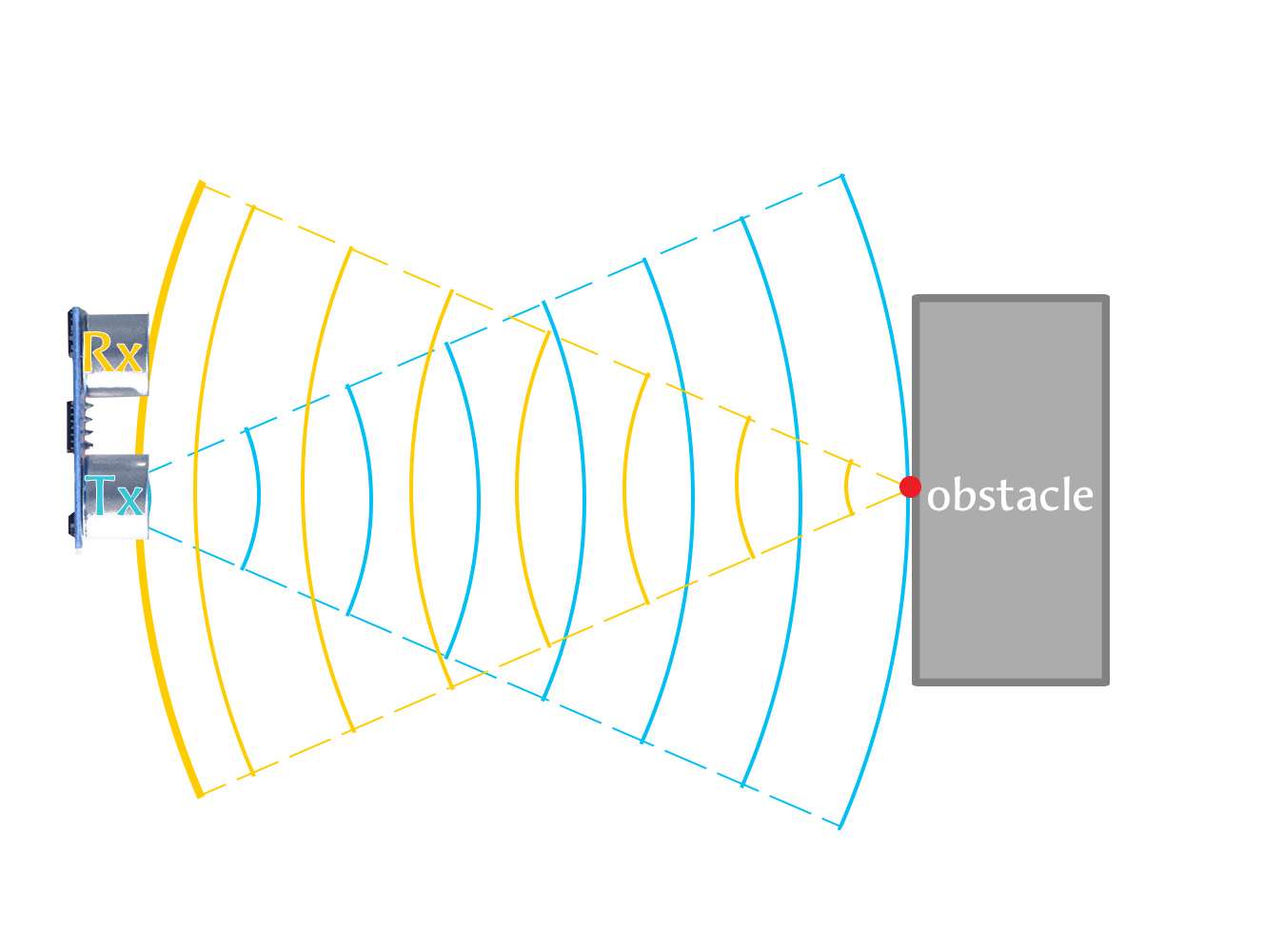
This project implements a basic prototype of adaptive cruise control using the HC-SR04 ultrasonic sensor, which is capable of measuring the distance to objects in front of the vehicle. The system is ideal for small-scale autonomous vehicles, robotics platforms, or DIY model cars, and demonstrates the core principle behind commercial ACC systems. HC-SR04 typically works within a range of 2 cm to 400 cm.

**Working Principle:**

1. Distance Measurement:  
   The HC-SR04 ultrasonic sensor emits high-frequency sound waves and listens for the echo reflected from an object. By measuring the time interval between the emission and reception, the distance to the object is calculated.
2. Speed Adjustment:  
   The microcontroller continuously reads the distance measured by the ultrasonic sensor. If the object in front is at a safe distance (e.g., >50 cm), the vehicle maintains or increases speed. If the distance decreases (e.g., <50 cm), the system reduces speed or stops the vehicle to avoid collision.
3. Real-Time Response:  
   The ACC system uses a feedback loop to constantly monitor and react to changes in distance. This enables smooth deceleration and acceleration, mimicking the behaviour of real-world adaptive cruise control.

**Algorithm / Logic Flow:**

1. Initialize the HC-SR04 and set up the trigger and echo pins.
2. Continuously measure the distance to the object ahead.
3. Use if-else conditions or a proportional control logic to adjust the motor speed:
   * **Distance > 8 cm**: Full speed.
   * **5 cm < Distance ≤ 8 cm**: Medium speed.
   * **3 cm < Distance ≤ 5 cm**: Stop motor.
4. Send PWM signals to the motor driver to change speed accordingly.



(Obstacle detection using HC-04 Ultrasonic Sensor)

1. **Driver Drowsiness Detection:**

Most of the sources suggested that about 20% of road accidents are due to fatigue. Driver Drowsiness Detection aims to prevent collision due to driver fatigue. The vehicle obtains information such as facial patterns and eye movement to monitor driver’s activities correspond with drowsy driving. If drowsy driving is suspected then the vehicle will typically sound awful out alert and may vibrate the driver’s seat.

The Driver Drowsiness Detection system is a real-time computer vision-based solution designed to monitor the eye activity of a driver using a webcam and raise an alert if drowsiness is detected. The algorithm utilizes **facial landmark detection** to track the **eye state** (open or closed) over time, and uses a carefully selected threshold to identify prolonged eye closure, which is one of the most common symptoms of drowsiness. The system leverages libraries like **dlib**, **OpenCV**, **imutils**, and **SciPy**, combining lightweight computational geometry with real-time video processing.

**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.

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

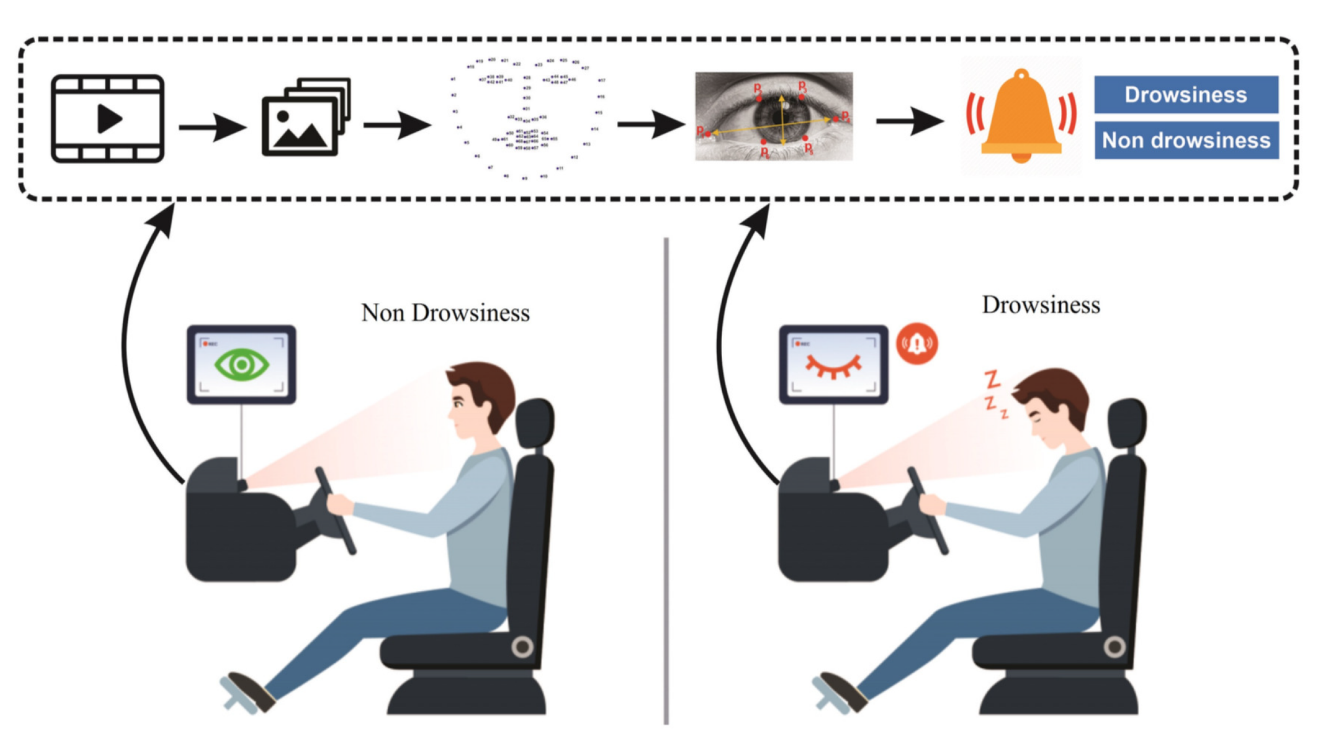
Since occasional blinks are normal, a single low EAR reading is not sufficient to declare drowsiness. Instead, the system monitors the EAR over a sliding window of frames. If the EAR falls below a predefined threshold (typically around 0.25) for a continuous number of frames (e.g., 25 frames), the driver 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.



**Real-time drowsiness detection**