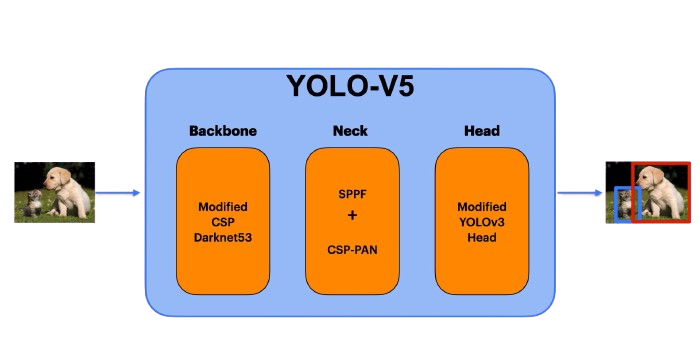
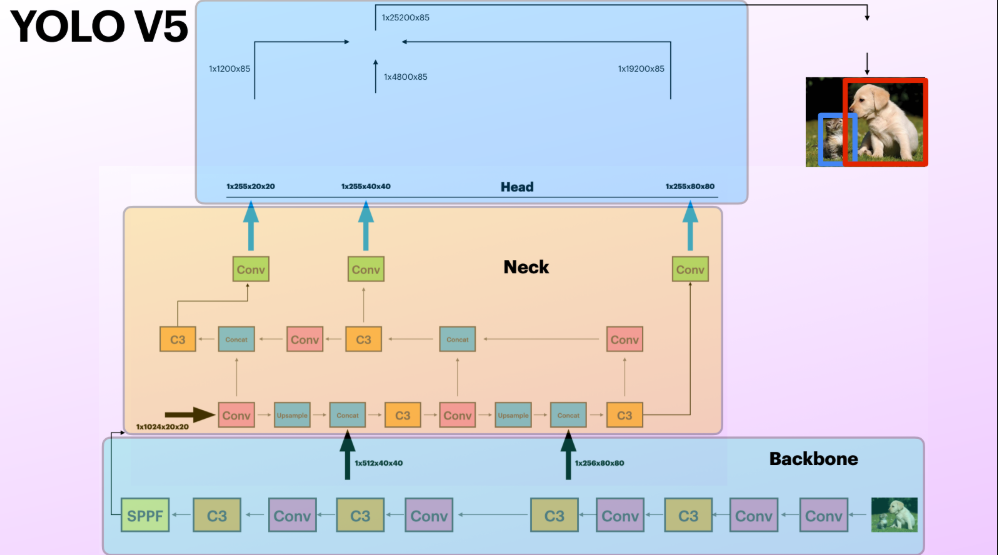
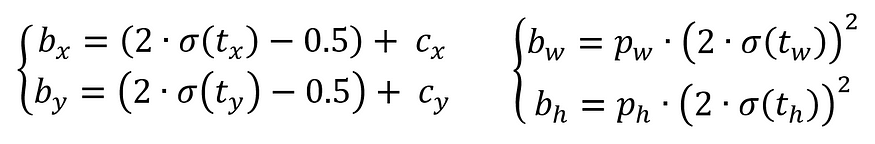
**Connected Autonomy**

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.

YOLO v5 is a popular real-time object detector, it is a PyTorch implementation of YOLO Single Stage Detector (SSD) which is an object detection algorithm which draws the boundary boxes over the image and predicts the probability of class. YOLOv1 to YOLOv4 are implemented on Darknet, whereas YOLOv5 is built on top of PyTorch implementation of YOLOv3 for which it gives better speed and accuracy and best suited for real-time applications.   
It has 5 different Models YOLOv5n, YOLOv5s, YOLOv5m, YOLOv5l, and YOLOv5x and each model is different in size and has a different use case. YOLO v5s is ideal for running inference on the CPU. We specifically chose YOLO v5s for its accuracy and speed in real-time light weight application and model exchange for.





Algorithm for finding boundary box in YOLOv5: 

**Steps for Traffic sign recognition:**

1. Download the Traffic sign Dataset
2. Annotate the dataset using roboflow
3. Install perquisites and setup environment
4. Train using YOLOv5s
5. Export the model
6. import the trained model for real-time detection (PyTorch / Ultralytics YOLO)
7. Perform real-time detection using OpenCV (Threshold)
8. Capture the detection result and send it to Controller using Serial port
9. Receive the command and perform actuation
10. Store the actuation data to cloud

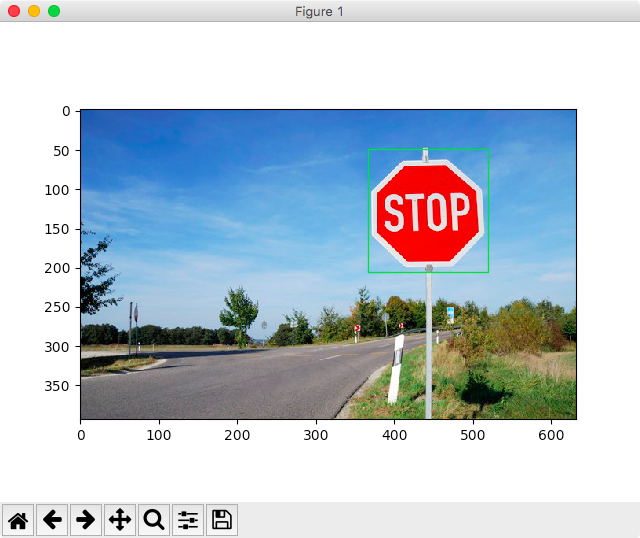
**Step 1:** Download the Traffic sign Dataset

In the process of data collection, we use the most popular dataset from Kaggle named GTRSB traffic sign dataset which is an open-source dataset with 43 different classes of traffic sign images used for training, testing image classification and traffic sign recognition models. This dataset contains total of 51840 (39,209 images of training with 43 different classes and 12,631 for testing) traffic sign images in different angel and different weather condition and different time including day and night. For all the images the dataset provides the labels in separated excel files.

**Step 2:** Annotate the dataset using roboflow

Roboflow is a platform which provides various tools for simplifying data collection and annotation, model training, model deployment with dev tools and API and also use of deployed trained model API mostly in computer vision application. Roboflow makes easy to deploy models in various environments such as cloud devices and edge devices.

In this project we use roboflow for annotation of our traffic sign images. The annotation tool of roboflow simply draw a boundary box region over image. It simply finds the region of interest (roi) or labels from an image by the help of boundary box annotation and save the labels in a .xml file (.json and .txt are also available). The annotation is in a separate file according to the images and will map with the images during training. The annotation is simply a set of coordinates where the boundary boxes will be drawn or in which region the classification task will be performed. After annotation we must ensure that we have a **.yaml** file which specifies the paths of train, test folders and class names.



Boundary box annotation

20 0.45552884615384615 0.5552884615384616 0.6778846153846154 0.6105769230769231

(Coordinates of that region)

**Step 3: Install perquisites and setup environment**

For model training we need to install and setup the tech stack which will be used in training our model.

For model training we use google colab T4 GPU. We need to import our dataset to google colab then install the libraries which will be used for training

* Ultralytics (Can be install using git or directly with pip)
* torch
* torchvision
* Clone YOLOv5s from git

**Step 4: Train model using YOLOv5s**

Run this command for training:

*python train.py --img 640 --batch 16 --epochs 500 --data data/traffic\_sign.yaml --weights yolov5s.pt --device 0*

--img 640 → Image size.

--batch 16 → Batch size.

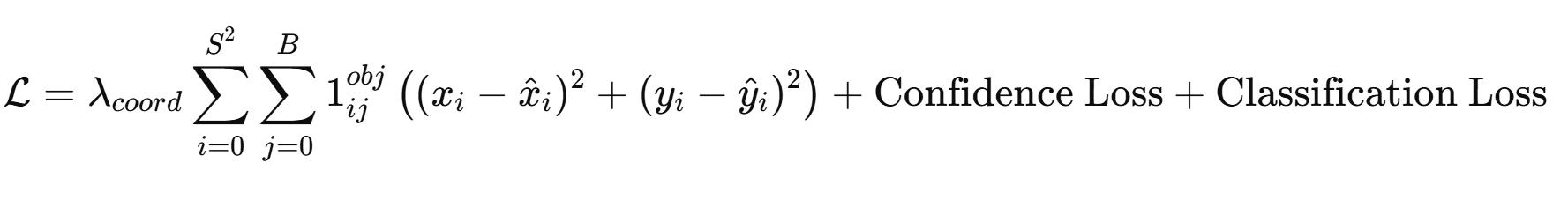
--epochs 50 → Number of training epochs.

-- traffic\_sign.yaml → Dataset configuration file.

--weights yolov5s.pt → Pretrained YOLOv5 model.

Loss Function:

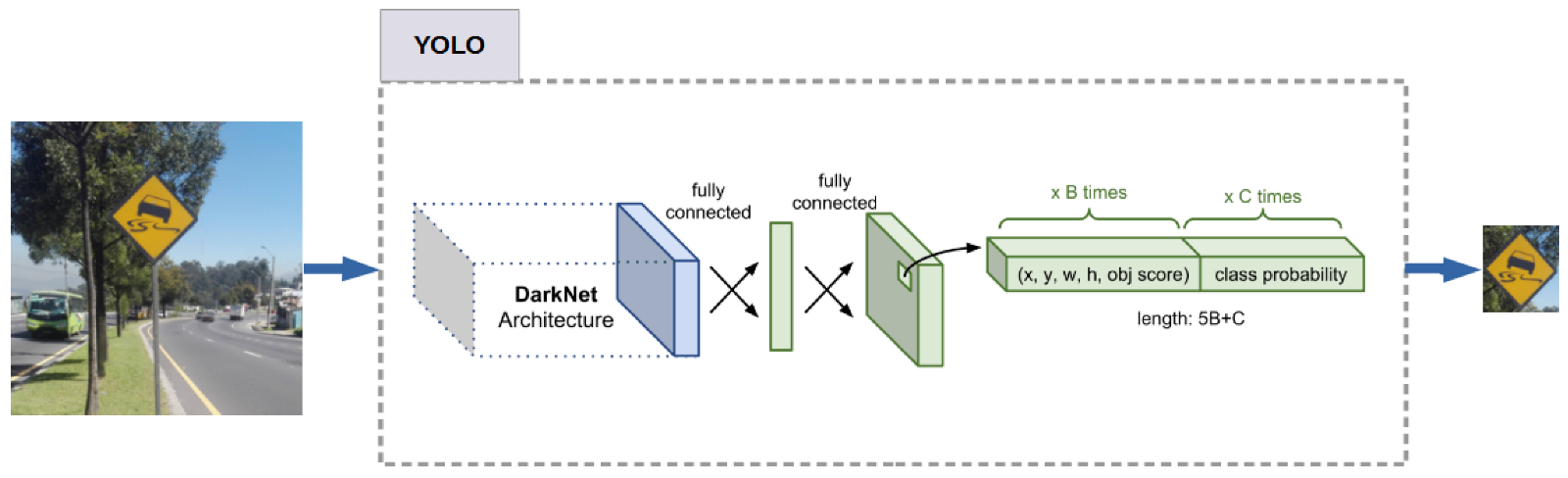
Loss = Localization loss + Confidence loss + classification loss



Where S - Grid size

BBB - Number of bounding boxes per grid cell

1ijobj Indicator if an object is present

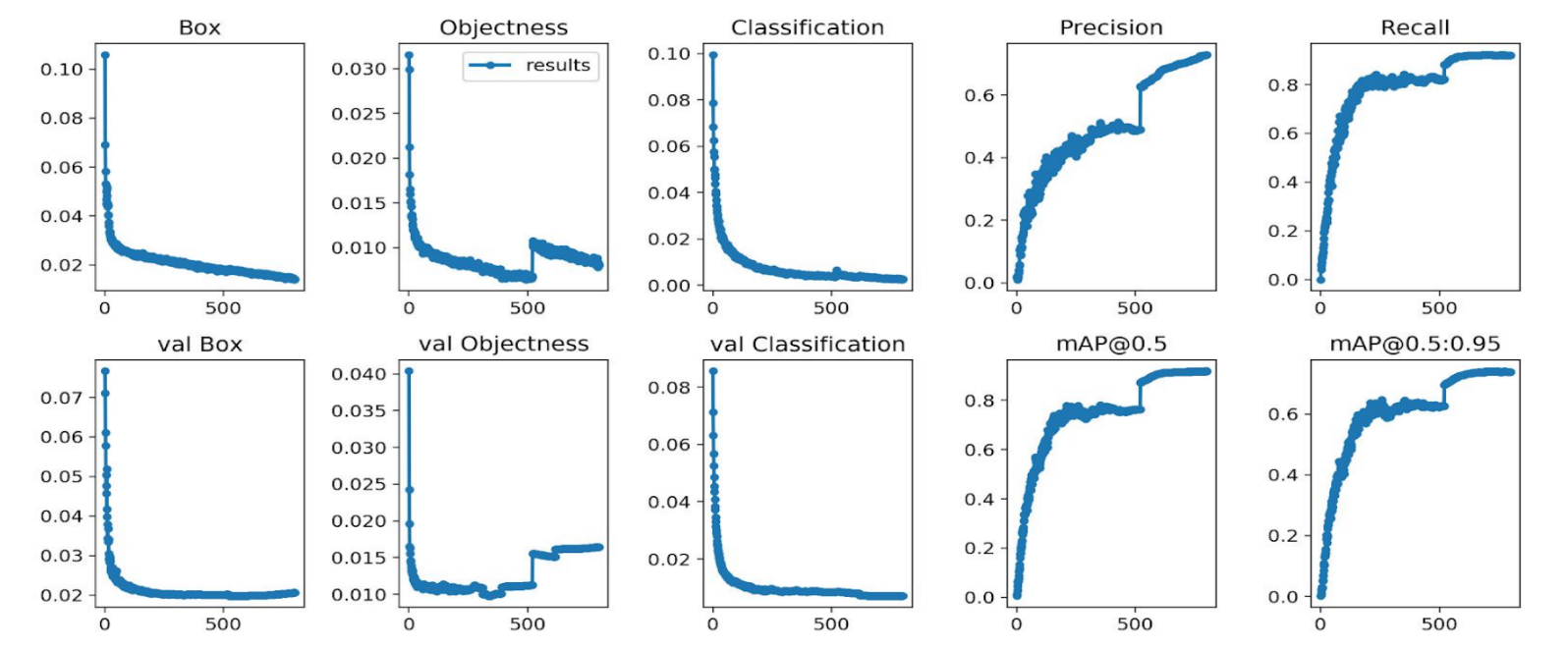


Bounding boxes are predicted using (x, y, w, h, c)

Where (x, y) are the centre of the bounding box (w, h) being the width and height and (c) is the presence of the object. If c is 0 then there is no object is present and if it is 1 then there must be at least one object

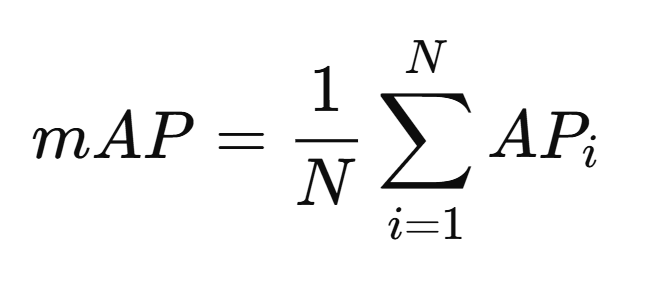
YOLO uses Intersection over Union to calculate the overlap between predicted and actual bounding boxes:

**Intersection over Union (IoU)=Area of Intersection / Area of Union**



Model classification reports

**Mean Average Precision:**

****

Where

-NNN → Number of object classes

-mAPi​ → Average precision for class iii

**-mAP@0.5** → Calculates mAP using an **IoU threshold of 0.5**.

**-mAP@0.5:0.95** → Calculates mAP using multiple IoU thresholds from **0.5 to 0.95** with a step of **0.05**.

**Precision:** Measures the accuracy of correct prediction

**Precision = True Positive / True Positive + False Positive**

**Recall:** Out of all positive, how many are correctly identified

**Recall = True Positives / (True Positives + False Negatives)**

**Step 5: Export the trained model in best.pt format or best.onnx format.**

Capabilities:

1. Real-Time detection and classification up to 45 fps

2. Accuracy as high as 98%

3. Robust model - detects in all weather conditions

4. Trained on 43 classes

5. Able to detect a wide variety of signs (for ex. Indian, European, American traffic signs)

**Step 6:** Import required libraries for real-time detection

* argparse → For handling command-line arguments.
* time → For measuring inference time.
* Path → For managing file paths.
* cv2 → OpenCV for image and video processing.
* torch → For deep learning model operations using PyTorch.
* cudnn → CUDA for GPU acceleration.
* random → For generating random colors for bounding boxes.

**Step 7:** Perform real-time detection

For testing this model in real-time we use OpenCV library of python which use system camera for real-time detection. First load the pre trained model which is stored in .pt format or **.onnx** format (**.onnx** for light weight) and then perform these tasks

- **Real-time object detection** using YOLOv5.

- **Visualization** with bounding boxes and FPS.

- **Saving results** for further analysis.



**Step 8: Capture the result and send it to serial port**

Python provides a library named Serial which helps to send the classification result to the Serial port. Below three lines of codes are help to send the data to serial port and also we can store the result in a file for future analysis.

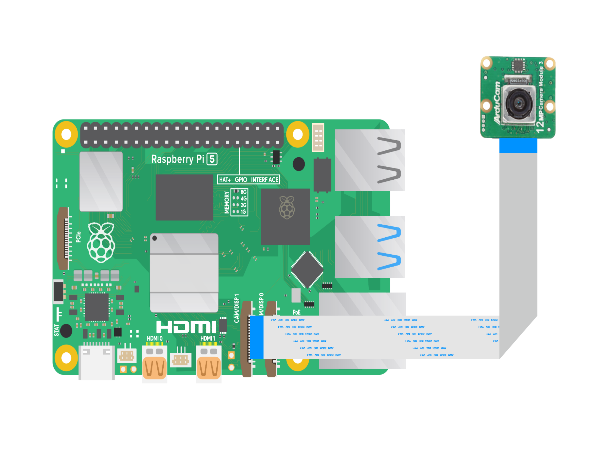
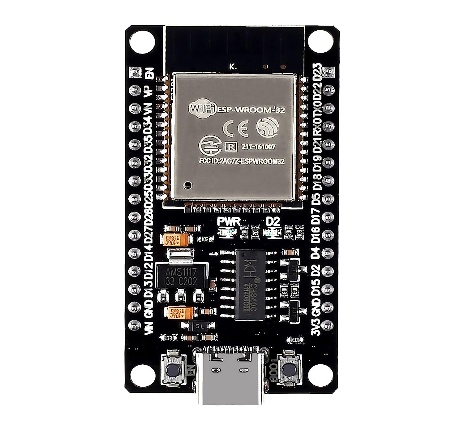
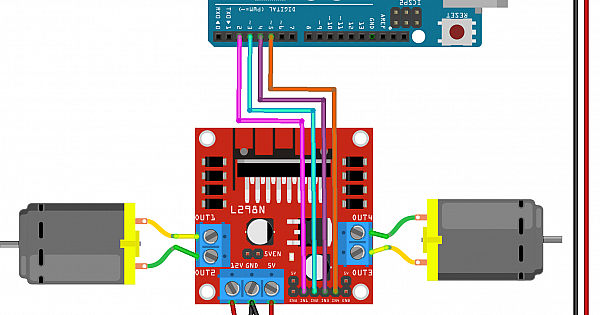
*import serial*

*Controller = serial.Serial('COM5', 9600)*

*Controller.write(b”Stop”)*

**Step 9: Receive the command and perform actuation**

The controller receives the command generated and sent by the primary device which is used for detection of signs and according to that command it will perform actuation through gear motor or LED. Suppose if the camera detected that it is a red sign or stop signal then the primary device (raspberry pi or Jetson Nano) sends the STOP command to the controller and according to that command controller will perform actuation. Here we add a secondary controller ESP32 for smooth conduct of cloud operation with low overhead on raspberrypi. To send the data from raspberry pi to ESP32 we use **UART serial communication** which is quite faster with no overhead.

Perform actuation and send the data to cloud

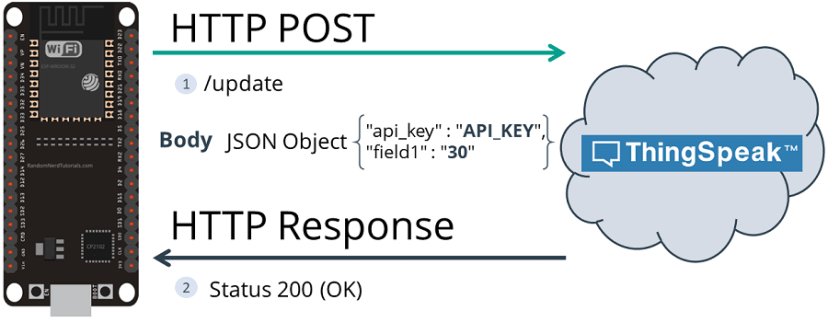
Send result as commmand

SS

Communication of command between IoT devices

**Step 10: Store the actuation data in cloud**

For storing data into cloud, we use Thingspeak cloud which is easy and simple to use. Thingspeak provides API based communication with internet enabled device. It also provides data visualisation facilities of collected data which helps in future analysis.



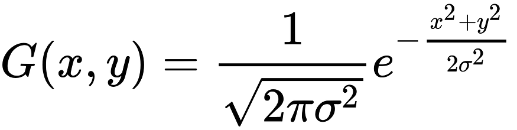
Logging sensor data to Thingspeak cloud

**Realtime Lane Detection**

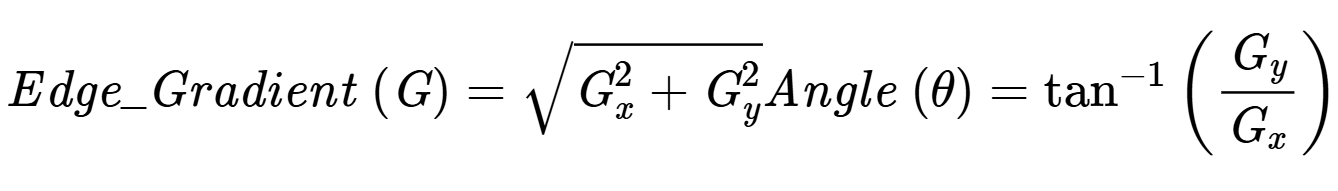
Realtime lane detection is a computer vision application used to detect the track and lane by considering the road marking or straight line from the road. Realtime lane detection is very crucial feature in autonomous car and driver assistant system. Lane detection is used to ensure correct navigation and to prevent accidents. Lane detection can be achieved by using various sensor like IR sensor which uses infrared technology to detect changes in environment. We can use IR sensor for detecting a line which will not be fruitful always. Another approach to achieve this is use computer vision algorithm and tools and libraries for detecting lane in real-time.

In this project we use python OpenCV which is a popular python library use in computer vision application and offers various algorithm for tasks like object detection, face recognition, image classification. In our project we use two dominant algorithms of OpenCV and these are Gussian Blur and Canny for Edge Detection. The Gaussian Blur algorithm of OpenCV is used for reduce unwanted noise from our image frame which helps in smooth edge detection. And the Canny algorithm is used for detecting changes in pixel coordinates to find the edges which helps us to draw the lane and this algorithm is also known as Canny Edge detection algorithm.

Equation behind Gaussian Blur:



Equation behind Canny edge detection:



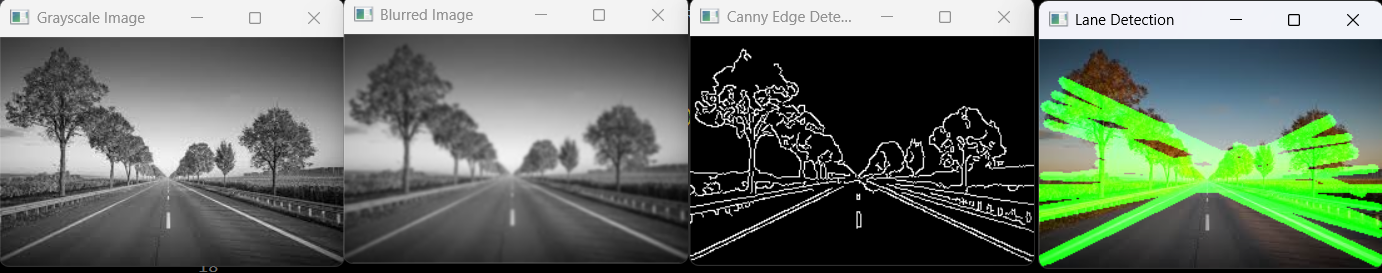
**Steps**

1. Capture the real-time video frame and calculate the FPS
2. Convert the image frame to grey image
3. Apply Gaussian Blur over the grey image frame
4. Apply canny edge detection over the blurred image
5. Find the region of interest where lane will be drawn
6. Draw the lines using the edge co-ordinates
7. Calculate the width between two lines and convert into meter

Let’s try over a sample lane image



The results after applying these steps



Gray Image Blur Image Image with edges Lines according to edges

**Step 1: Capture real-time image frame**

In the first step it read the rea-time frame from the camera by using OpenCV library of python and calculate the FPS by calculating the time difference between two frames.

**FPS = 1/ (curr\_time – prev\_time)**

**Step 2: Grayscale**

In the second step we apply OpenCV functionality to remove the colours from the image which helps the algorithm to find the edges of lines. For this we use ***cvtColor(frame, cv2.COLOR\_BGR2GRAY)*** method of python.

**Step 3: Gaussian Blur**

We use Gaussian blur over the image for smoothing image and to reduce noise from the image which will gives a productive result to us. We always prefer to use gray image with no colour to apply noise reduction techniques like Gaussian blur. OpenCV provides a popular noise reduction technique and smoothing technique named Gaussian blur and we can access it by writing code **cv2.GaussianBlur()** method.

**Step 4: Canny Edge detection**

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.

[[0 0 0 ... 0 0 0]

[0 0 0 ... 0 0 0]

[0 0 0 ... 0 0 0]

...

[0 0 0 ... 0 0 0]

[0 0 0 ... 0 0 0]

[0 0 0 ... 0 0 0]]

**Step 5: Region of interest**

Region of interest finds the relevant area of road for detecting lane, for this it will only consider the lower half of the image which contains road. Using region of interest removes the unnecessary objects like buildings and sky. We use ***cv2.fillPoly()*** which cover the area by drawing a triangle or polygon-like structure where lanes are expected.

**Step 6: Draw the lane over the image**

Drawing the lines of lane using the co-ordinates using opencv library cv2.line().

**Step 7: Calculate the width**

Calculating the width between two detected lines and if there is line detected then returns the length in meter an if there is no line then returns zero.

----------------------------------------------------------------------------------------------------------------

**3.1 SYSTEM MODEL**

The self-driving car is an advanced cyber-physical system that seamlessly integrates sensor fusion, machine learning, computer vision, control theory, and actuation mechanisms to enable autonomous navigation without human intervention. At its core, the vehicle is built upon a layered architecture comprising perception, decision-making, control, and actuation modules, each intricately connected through both hardware and software components. The perception system relies on a suite of sensors—including cameras, ultrasonic sensors, Camera modules, MQ3 sensor to gather real-time environmental data, which is then processed using computer vision techniques and deep learning models like YOLO for traffic sign recognition and lane detection. Machine learning further enables object detection, obstacle avoidance, and behavioural analysis such as drowsiness or alcohol detection. These plans are translated into motion commands by control systems employing PID controllers, which are then executed by actuators such as DC motors and servo motors through an L298N motor driver. The hardware backbone includes a Raspberry Pi or Jetson Nano for processing and an ESP32 microcontroller for real-time actuation and sensor interfacing. Communication between modules is facilitated via serial communication and cloud platforms like ThingSpeak for data monitoring and remote control. The tightly coupled integration of hardware and intelligent software results in a prototype capable of recognizing traffic signs, maintaining lanes, avoiding obstacles, and making context-aware navigation decisions, embodying the principles of autonomy and intelligent control in a dynamic driving environment.

**Simulation environment:**

The self-driving car is modelled using a modular architecture, dividing the system into the following components:

* + - * Perception Module: Gathers data from sensors (camera, ultrasonic, GPS, MQ3, etc.)
      * Decision-Making Module: Processes input using algorithms for traffic sign recognition, object detection, and lane detection.
      * Control Module: Converts high-level decisions into low-level actuator commands (motor speed, direction).
      * Communication Module: Handles data transmission between cloud services (ThingSpeak), remote devices (BLE / WLAN), and internal subsystems.

**Software Development**

The software development of our self-driving car project involved integrating multiple modules using Python and embedded C. Key components included real-time lane detection using OpenCV, traffic sign recognition with a custom-trained YOLOv5 model, and sensor-based safety features like drowsiness and alcohol detection. Communication between Raspberry Pi and ESP32 was achieved using serial protocols, while data was transmitted to the ThingSpeak cloud for monitoring. Custom control algorithms managed motor operations via the L298N driver. Additionally, a React Native mobile app was developed for manual override and testing. The modular architecture ensures scalability and seamless interaction between hardware and software components.

Operating System and Languages:

* **OS**: Raspberry Pi OS (Raspbian Strech)
* **Languages**: Python (OpenCV, PyTorch), Arduino C++, MATLAB (ThingSpeak)

|  |  |
| --- | --- |
| **Module** | **Technologies Used** |
| Lane Detection | OpenCV, Hough Transform, Edge Detection |
| Object Detection | YOLOv5 (Ultralytics), PyTorch |
| Alcohol Detection | MQ3 with ADC → threshold-based logic |
| Cloud Communication | ThingSpeak (UDP or HTTP POST) |
| Remote Control | Bluetooth serial interface |
| Drowsiness Detection | Python, dlib, EAR |

Control Algorithm (Example: Lane Following)

1. Capture video frame from camera.
2. Apply grayscale conversion + Gaussian blur.
3. Use Canny edge detection.
4. Apply Hough transform to detect lane lines.
5. Determine steering angle using line slope.
6. Send motor command to microcontroller.
7. **Hardware Requirements**

Core Components:

|  |  |
| --- | --- |
| **Component** | **Description** |
| **Microcontroller** | Raspberry Pi or Jetson Nano, ESP32 |
| **Motor Driver** | L298N Dual H-Bridge for motor control |
| **Motors** | DC BO Gear motors with encoders |
| **Camera Module** | Raspberry pi cam rev 3 |
| **Ultrasonic Sensor** | HC-SR04 for obstacle avoidance |
| **MQ3 Sensor** | For alcohol detection (driver safety module) |
| **Cloud System** | ThingSpeak for telemetry and monitoring |

Sensor Integration:

* **Ultrasonic sensors** detect proximity to obstacles.
* **MQ3 sensor** detects alcohol in the environment (e.g., from the driver).
* **Camera module** provides input for computer vision tasks like lane and traffic sign detection.
* **GPS module** sends location data to the cloud via UDP.

1. **System Integration and Workflow**
2. Start the system – Raspberry Pi initializes all sensors and modules.
3. Perception begins – camera captures video, ultrasonic and MQ3 begin sensing.
4. Real-time analysis – lane lines and obstacles are detected.
5. Decision logic – determines whether to steer, accelerate, or stop.
6. Actuation – microcontroller sends PWM(HIGH/LOW) signals to motor driver.
7. Logging – data sent to ThingSpeak cloud for remote visualization.
8. Optionally, remote user can control vehicle via Bluetooth in manual mode.
9. **Cloud and Remote Communication**

* **UDP Protocol** is used to send lightweight telemetry data to ThingSpeak (speed, GPS, sensor values).
* **Bluetooth** allows remote driving control through a smartphone app using serial commands like '1', '2', '3', '4','0'.

1. **Challenges Encountered**

* **Synchronization** of sensor data in real-time.
* **Latency** in cloud communication (ThingSpeak update interval).
* **Power management** for running multiple modules on Raspberry Pi.
* **Camera calibration** for consistent lane detection.

The system modelling and implementation of the self-driving car follow a modular and scalable approach. By combining sensor data processing, cloud connectivity, real-time control, and machine learning-based perception, the vehicle is capable of navigating a structured environment with minimal human intervention. This prototype lays the foundation for more advanced autonomous systems and provides a testbed for future improvements such as V2X communication, LiDAR integration, and autonomous fleet coordination.

--------------------------------------------------------------------------