Detection of Traffic Signs and Traffic Lights Using YOLOv8 and MQTT

Vincenzi Fabio, Iorio Matteo, Furi Stefano

December 2024

Contents

1 Introduction

1.1 Abstract

This project focuses on the development of a module capable of detecting and displaying *traffic signs* and *traffic lights* in a simulated environment. To enhance realism and usability, the module also identifies the color of traffic lights, offering users a more comprehensive understanding of their surroundings. The Carla simulator, along with its extensive library, served as the foundation for this endeavor, enabling the integration of cameras onto vehicles within the simulation. This approach allowed for the creation of a robust system that mirrors real-world scenarios, paving the way for advanced traffic detection and analysis solutions.

1.2 State of the Art

The detection of traffic signs and traffic lights is a critical component of autonomous driving systems and advanced driver-assistance systems (ADAS). This task involves the use of onboard cameras mounted on vehicles to identify and interpret traffic signs and lights in real time. The ability to accurately and efficiently detect these elements is essential for ensuring safety, compliance with traffic laws, and effective navigation.

1.2.1 Core Methodologies

Modern approaches for detecting traffic signs and traffic lights typically leverage deep learning, specifically convolutional neural networks (CNNs), due to their superior performance in image recognition tasks. The task can be divided into two primary stages:

- **Detection**: Identifying the location of traffic signs and lights within an image.
- **Classification**: Determining the type and meaning of the detected traffic signs or the current state (e.g., red, yellow, green) of the traffic lights.

1.2.2 Traffic Sign Detection

Traffic sign detection often involves object detection frameworks such as:

- `YOLO` (You Only Look Once): This real-time object detection system predicts bounding boxes and class probabilities directly from full images. YOLO's speed makes it well-suited for real-time applications;
- `Faster R-CNN`: This framework uses region proposal networks (RPN) to generate candidate object regions and then classifies these regions. It offers high accuracy but can be computationally demanding;
- `SSD` (Single Shot Multibox Detector): Similar to YOLO, SSD combines speed and accuracy, making it another popular choice.

For traffic signs, the datasets often include thousands of labeled images with signs under various conditions, such as occlusion, poor lighting, and weather changes. Notable datasets include:

- The German Traffic Sign Detection Benchmark (GTSDB).
- The Mapillary Traffic Sign Dataset, which provides a diverse set of signs from different countries.

1.2.3 Traffic Light Detection

Traffic light detection shares similarities with sign detection but has unique challenges, such as detecting small objects (traffic lights are often smaller relative to the image size) and identifying color and state. The detection process typically combines:

- *Image segmentation*: To isolate the traffic light and its components;
- *Feature extraction and classification*: To determine the light's state. Techniques such as color thresholding and CNN-based classification are widely used.

2 Domain

As described inside the *Abstract* part, this project is mainly focused on the **object detection** of signs and traffic lights. There are many ways to accomplish this task, by using Deep Neural Networks or specific Computer Vision strategies. This was our first point of choice, because depending on which strategy we can obtain different results with a different computation times. The time complexity must be very short, in order to provide real time feedback. Below we will show how we faced the problem, what we considered to reach our goal and how we managed to decide what was the best overall architecture for this project.

3 Analysis

Here we will listed all the specific and information of our system. First of all, we needed to find a simulator that was able to create cars, create cameras and attach the camera to the specific car, in this way we were able to explore the roads using the mounted camera. The camera specific, is simple, It must be a *RGB* camera, where we could have also used a *RGB-D* camera, but four our task depth information was to necessary, in this way in an optic real life situation It is also possible to use less expensive cameras to attach to our vehicle. The selected simulator is **Carla**.

Carla CARLA is an open-source simulator designed specifically for autonomous driving research. Built from the ground up, it serves as a highly modular and flexible platform capable of addressing a wide range of tasks related to autonomous driving. These include, but are not limited to, learning driving policies, training perception algorithms, and testing decision-making strategies. The simulator leverages the power of Unreal Engine to deliver realistic and high-fidelity simulations, while its road and urban settings are defined using the ASAM OpenDRIVE standard (currently version 1.4). CARLA offers extensive control over the simulation environment through an ever-evolving API available in Python and C++, ensuring adaptability to the growing needs of the research community. Its robust design makes it an essential tool for advancing the field of autonomous vehicle technology.

Our Task With the simulator chosen and its advantages clarified, the primary objective of this project can be outlined as follows: to *detect and recognize traffic signs and traffic lights*. By leveraging Carla's APIs, we can spawn vehicles in the simulation and attach cameras to them. This setup enables real-time feedback of the simulated road environment, allowing us to analyze the surroundings, identify traffic regulations, and better understand the dynamics of the environment around the car. Beyond detection and recognition, this project also aims to explore the integration of these capabilities into advanced driving systems, paving the way for decision-making algorithms that prioritize safety and efficiency. Further enhancements may include implementing data augmentation for training models, validating results through benchmarking, and assessing performance under various weather and lighting conditions.

4 Tech Stack

To accomplish our objectives, we leveraged a diverse set of technologies, each contributing to the development and refinement of our system. We began by selecting `Python` as our primary programming language due to its simplicity and extensive ecosystem. Next, we incorporated essential libraries for image processing and numerical computation, such as `OpenCV` and `Numpy`, which provided robust tools for handling and analyzing visual data. Additionally, we utilized `Ultralytics` to integrate the YOLO (You Only Look Once) module, enabling state-of-the-art object detection capabilities. This combination of technologies allowed us to build a powerful and efficient system for achieving our project goals.

4.1 Plain Computer Vision algorithms

We started by exploring the `OpenCV` library, in order to see if It was possible to find some groups of algorithms that were useful to detect the different objects. By doing some researches we managed to find two different possible ways to achieve our goal.

4.1.1 Object Shape Detection

By using Open CV as library, It is possible to detect specific geometric shapes in an Image, and we know that Traffic signs and Traffic lights have shapes recognized by all the globe (squares, circles, triangles and so on). At first this strategy can be seen as the best option, It would be definitely faster than a DNN and also more accurate, but It is not like that, we know that the car where the camera is mounted is moving, and this creates problems for the camera distortion. Also because we have to identify 3/4 different shapes at the same time, we need to start different iterations of the same algorithm on the same image, and we do not have the complete security in detecting traffic signals/lights, because we could for example identify also other objects with the same geometric shapes we are looking for (tires, etc etc). So we tried to switch with another strategy.

4.1.2 Template Matching

The second idea that we had was to use the so what called *Template Matching* where a template of an Image (with different sizes) is passed over the current image, and If the template matches with an interval of pixels, then we found the Traffic light/sign. But also this time It is not so easy as we say because in total we know that there are over 300 traffic signs in Italy, and by making a small heuristic we can say that there are at least:

$$TotalSigns = 195 \times 300 = 585000 \tag{1}$$

where 195 is the number of all the different countries in the world and 300 is the average number of traffic signs for a single country. So It is quite impossible to give a real time feedback if we have to check in each frame at least 58 500 templates.

4.2 Deep Neural Networks

Now that we decided that It was not possible to use plain computer vision algorithms, we decided to switch with Deep Neural Networks, because they are definitely easier to use and also much faster than *Template Matching* or *Shape Detection*. As we said inside the domain description, our task is labeled as an **Object Detection** task. In order to accomplish this task, we know that there are already different existing architectures for doing this. Among all the different architectures, we were deciding between `YOLO`, `EfficientDet` and `RetinaNet`.

4.2.1 YOLO

YOLO (You only look once) is a real-time object detection system widely used in computer vision tasks such as autonomous driving. Unlike traditional object detection methods that involve region proposal and classification in multiple stages, YOLO treats object detection as a simple regression problem. It tries to predict the bounding boxes and classes probabilities directly from the input image in one pass through the use of Its network. This makes YOLO exceptionally **fast** and suitable for **real-time** applications.

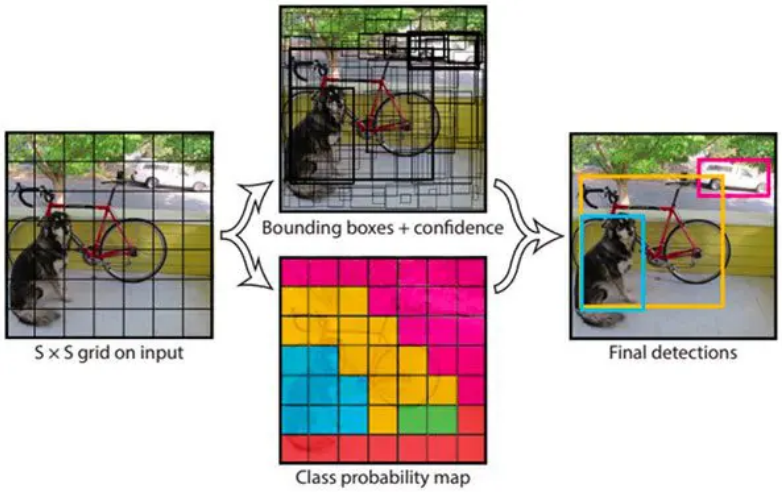

Figure 1: How YOLO works

Architecture YOLO's architecture is inspired by `Convolutional Neural Networks` and the YOLOv8 version (the one we used) has three main parts:

- **Backbone**: extracts essential features from the input image;
- **Neck**: aggregates features at different scales for better detection of objects of varying sizes;
- **Head**: predicts bounding boxes, class probabilities and optionally masks for segmentation

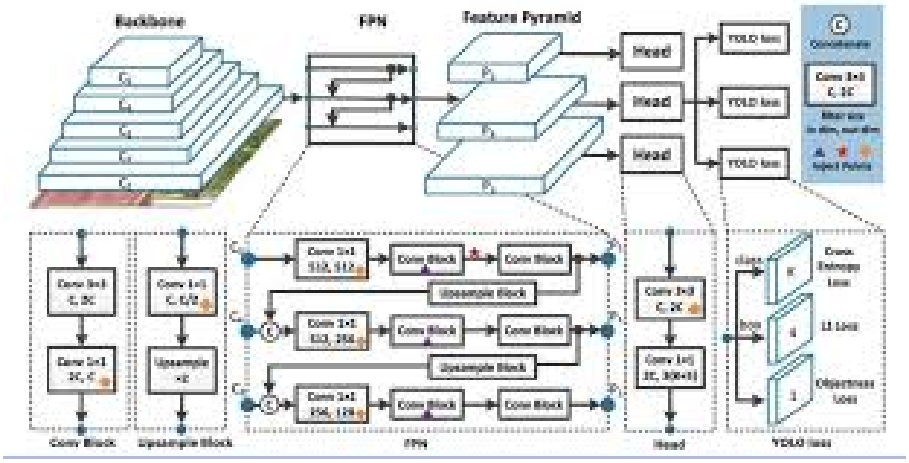

Figure 2: YOLO Architecture

Components - Backbone The **backbone** in YOLOv8 is responsible for feature extraction and employs a highly optimized Convolutional Neural Network (CNN) structure:

- **Cross Stage Partial Networks**: YOLOv8 uses `CSP` modules to improve gradient flow and reduce the computation. These modules split feature maps into two paths-one undergoes transformations, and the other bypasses them-before merging;
- **Focus Layer**: Replaces earlier versions' initial layers to better preserve spatial information;
- **Depthwise Convolutions**: Lightweight convolutions are used to reduce parameter count while maintaining efficiency;
- **Feature Pyramid**: The backbone processes the input at multiple scales, capturing fine-grained details and abstract patterns.

Components - Neck The neck in YOLOv8 bridges the backbone and the head, enhancing the detection capability for objects of varying sizes:

- **Path Aggregation Network**: Combines low-and high-level features for multi-scale detection;
- **FPN-like Strucutr**: The neck aggregates information across layers to improve small object detection, similar to `Feature Pyramid Networks`;
- **Bidirectional Feature Pyramid**: YOLOv8 introduces bidirectional connections to refine multi-scale feature vision.

Components - Head The head generates predictions, combining outputs from the neck. YOLOv8's detection head is lightweight and refined:

- **Decoupled Head**: YOLOv8 separates classification and localization tasks for better performance;
- **Anchor-Free Design**: YOLOv8 adopts an anchor-free detection approach, reducing the reliance on predefined anchor boxes and simplifying the architecture. This makes it more efficient and easier to tune;
- **Objectness Score**: For each detection, the head predicts a confidence score, indicating whether the bounding box contains an object.

4.2.2 EfficientDet

`EfficientDet` is a state of the art object detection model family that balances accuracy and computational efficiency. It builds on the principles of the `EfficientNet` backbone and introduces a novel Bidirectional Feature Pyramid Network for efficient feature fusion, making it scalable for different and resource constraints. `EfficientDet` is used for *object detection* tasks across a variety of domains, including autonomous driving and real-time applications. It achieves a balance between *speed* and *accuracy* by:

- **Leveraging EfficientNet as a backbone**: for feature extraction;
- **Using Bidirectional Feature Pyramid Network for feature fusion**: to aggregate multi-scale features efficiently;
- **Employing a compound scaling method**: to uniformly scale the network's resolution, depth and width based on available resources.

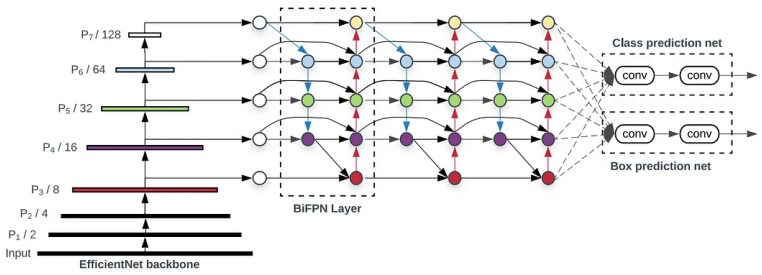

Figure 3: EfficientDet overall architecture

Architecture - Backbone `EfficientDet` uses **EfficientNet** as Its backbone. EfficientNet is a family of convolutional neural network optimized for both performance and efficiency using compound scaling method:

- **EfficientNet** is built using `Mobile Inverted Bottleneck Convolution` which combines depthwise separable convolutions and squeeze-and-excitation blocks;
- The backbone outputs multi-scale feature maps, which are passed to the Bidirectional Feature Pyramid Network for further processing. EfficientNet's different versions are used to scale EfficientDet models, allowing users to select the model size based on they hardware constraints.

Architecture - Bidirectional Feature Pyramid Network The **Bidirectional Feature Pyramid Network** is a key innovation in EfficientDet. It improves on the traditional Feature Pyramid Network by:

- **Bidirectional Connections**: unlike Feature Pyramid Network, which only aggregates features in a top-down manner, BiFPN fuses features bidirectionally.
- **Learnable Weights**: BiFPN assigns learnable importance weights to each feature map, allowing the network to adaptively prioritize useful features;
- **Efficient Fusion**: Redundant nodes are removed, and lightweight operations are used to minimize computational cost.

The **BiFPN** structure repeats several times to refine feature maps.

Architecture - Detection Head The detection head is lightweight and decoupled, predicting:

- **Bounding Boxes**: Using regression to output for object locations;
- **Class Probabilities**: For each bounding box, predicting the likelihood of different object classes. EfficientDet uses a shared prediction network for both bounding box regression and classification, reduaciang redundancy.

Architecture - Compound Scaling `EfficientDet` uses a compound scaling method to scale all parts of the network systematically.

- **Input Resolution**: Scales the image size for higher detail;
- **Network Depth**: Increases the number of layers in the backbone and `BiFPN`;
- **Newtwork Width**: Expands the number of channels in feature maps.

The compound scaling formula is parameterized by three factors:

$$Compound(\alpha, \beta, \gamma) \tag{2}$$

that determine how to allocate additional computation across these dimensions. This allows `EfficientDet` to be scaled from lightweight models to highly accurate but resource-intensive model

4.2.3 RetinaNet

`RetinaNet` is a **one-stage** object detection model introduced by *Facebook*. It bridges the gap between the accuracy of two stage detectors and the speed of one stage detectors. `RetinaNet` achieves this balance by introducing a novel Focal Loss that addresses the class imbalance problem which is common in one stage detectors.

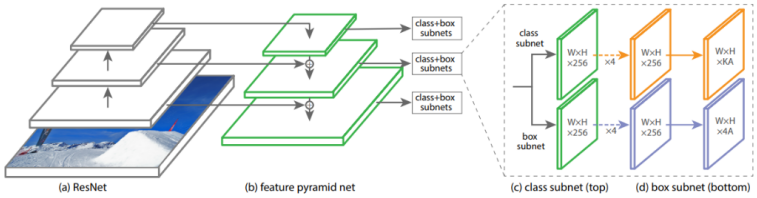

Figure 4: RetinaNet overall architecture

General Description `RetinaNet` is used for **object detection** tasks where speed and accuracy are both critical, such as autonomous driving, surveillance and robotics. It uses a single neural network to predict object bounding boxes and class probabilitiesd directly from an input image. The **key features** of `RetinaNet` are:

- **One-Stages Detector**: Unlike two-stages detectors, `RetinaNet` eliminates the region proposal step, resulting in a faster model;
- **Focal Loss**: A novel loss function that handles the class imbalance problem by focusing more on hard examples during training;
- **Feature Pyramid Network (FPN)**: `RetinaNet` uses an Feature Pyramid Network to generate multi-scale feature maps, allowing it to detect objects of different sizes effectively.

Architecture Overview `RetinaNet` consists of three main components:

- **Backbone**: Extracts feature maps from the input image using a pre-trained CNN;
- **Feature Pyramid Network**: Enhances the feature maps from the backbone for multi-scale detection;
- **Prediction Subnets**: Two lightweight sub-networks that predict class scores and bounding box regressions

Architecture - Backbone The **backbone** extracts *high-level features* from the input image. `RetinaNet` typically uses a `ResNet-50` or `ResNet-101` as Its backbone, pre-trained on `ImageNet`. The key characteristics are:

- The backbone generates feature maps at different scales corresponding to different levels in the `ResNet` architecture;
- These feature maps serve as inputs to the `Feature Pyramid Network`.

Architecture - Feature Pyramid Network The `Feature Pyramid Network` plays a critical role in `RetinaNet`'s ability to detect objects at multiple scales. It builds a **top-down pathway** with lateral connections to merge low-resolution, semantically rich features with high-resolution, low-level features:

- **Input**: Feature maps from different stages of the backbone;
- **Output**: Multi-scale feature maps for object detection.

Each level of the `Feature Pyramid Network` corresponds to a different scale in the image, enabling RetinaNet to detect objects of varying sizes effectively.

Architecture - Prediction Subnets `RetinaNet` uses two parallel lightweight sub-networks to process the multi-scale feature maps output bu the `Feature Pyramid Network`:

- **Classification Subnet**: Predicts the probability of object classes for each anchor box at every location on the feature map, It uses a **sigmoid activation** to produce class probabilities independently for each anchor and It operates on each feature map level;
- **Regression Subnet**: It predicts the **bounding box offsets** for each anchor box and it uses **smooth L1 loss** for bounding box regression.

Focal Loss One of `RetinaNet`'s innovations is the **Focal Loss**, which addresses the class imbalance between foreground and background during training. The problem with class Imabalance is that in one-stage detectors, the vast majority of anchor boxes correspond to the background, which overwhelms the learning process and reduces performance. Thanks to this loss, It allows to improve the `RetinaNet` ability fo detect small and challenging objects.

Anchor Boxes `RetinaNet` uses **anchor boxes** at each location on the feature maps. Each feature level generates anchors of different sizes and aspect ratios:

- **Sizes**: Anchor boxes are scaled according to the feature map resolution.

The network predicts offsets to adjust these anchors to match the ground truth boxes.

4.2.4 Model Decision

For our project, we selected the **YOLO** (You Only Look Once) architecture due to Its outstanding balance between *speed* and *accuracy*. Here, we compare **YOLO** with **RetinaNet** and **EfficientDet**, highlighting the key reasons for our decision.

Speed In order to provide a real-time outcome, It is really important to have a network capable of high speed results, by also maintaining and high accuracy in the outcome. Here we list the main speed aspects between the three models:

- `YOLO`: Designed as a **one-stage detector**, **YOLO** processes the image in a *single forward pass* through the network. This results in extremely fast inference times, making It deal for **real-time** applications;
- `RetinaNet`: Although **RetinaNet** is also a **one-stage detector**, It is computationally heavier due to Its use of **Feature Pyramid Network** and two separate prediction subnets, this makes `RetinaNet` slower compared to `YOLO`;
- `EfficientDet`: While **EfficientDet** is optimized for efficient through Its **BiFPN** and compute scaling, It tends to be slower than **YOLO** for real-time scenarios, especially on edge-devices.

In conclusion **YOLO**'s speed advantage is unmatched, making It the best choice for **real-time** application and resource constrained environments.

Accuracy vs Complexity Other than having a **real-time** feedback we want also that our application gives the correct result each time a detection occurs. Now that we have highlighted this need, we will follow the main reasons on why we chose **YOLO**.

- `YOLO`: **YOLO** achieves strong accuracy while mantaining speed, recent versions of YOLO, for example the one we used, offer impressive performance on benchmark datasets, with high `mAP` (mean Average Precision) scores and efficient bounding box predictions;
- `RetinaNet`: **RetinaNet** is highly accurate on smaller obect, thanks to Its **Focal Loss** that addresses class imbalance. However, this comes at the cost of increased computational complexity and slower inference;
- `EfficientDet`: While **EfficientDet** is more efficient than `RetinaNet` for Its accuracy, the compound scaling increases computational demands, especially for larger models.

In the end, `YOLO`'s resource efficiency makes It deal for real-time and edge-device deployment.

4.3 Training

Now that we know why we decided to choose for the `YOLO` architecture, we can move on the training phase. Here we will list all the different steps taken in order to provide knowledge at our **YOLO** model. We will discuss all the different steps used for accomplish our goal.

4.3.1 Carla Video

First thing first we started on **Carla**, we decided that in order to test our application It was necessary to have some real vehicle videos for testing our model. So in order to do that we decided to create and attach an *RGB* camera to a random vehicle and set the autonomous driving to `True`, in this way we were able to capture different street images (and crop them into a video) where It was possible to clearly see *Traffic Signs* and *Traffic Lights*. In order to do that, we defined two different functions: `spawn_vehicle` and `spawn_camera`. By using the `spawn_vehicle` function we are able to spawn a random vehicle in the map. So as to spawn a *vehicle* we first need to use the so called *blueprint* of the map.

Blueprint The **Blueprint** contains the information necessary to create a new **actor**. For instance, if the blueprint defines a car, we can change its color here, if it defines a lidar, we can decide here how many channels the lidar will have. A blueprints also has an ID that uniquely identifies it and all the actor instances created with it, an **Actor** is anything that plays a role in the simulation and can be moved around, examples of actors are vehicles, pedestrians, and sensors. (reference **Carla Blueprint**).

Then we moved to the implementation of the function `spawn_camera`. The main goal of this function is to create a `RGB` camera and attach It to the input vehicle, by doing this we are able to record and see what the vehicle is doing, how the vehicle is moving on the map and how the vehicle is driving and so on.

```
def spawn_vehicle(vehicle_index=0, spawn_index=0, pattern='vehicle.*'):
    blueprint_library = world.get_blueprint_library()
    vehicle_bp = blueprint_library.filter(pattern)[vehicle_index]
    spawn_point = world.get_map().get_spawn_points()[spawn_index]
    vehicle = world.spawn_actor(vehicle_bp, spawn_point)
    return vehicle

def spawn_camera(attach_to=None, transform=carla.Transform(carla.Location(x=1.2,
↪   z=1.2), carla.Rotation(pitch=-10)), width=800, height=600):
    camera_bp = world.get_blueprint_library().find('sensor.camera.rgb')
```

```
    camera_bp.set_attribute('image_size_x', str(width))
    camera_bp.set_attribute('image_size_y', str(height))
    camera = world.spawn_actor(camera_bp, transform, attach_to=attach_to)
    return camera
```

After recording four different videos, we started on working on the `YOLO` architecture, as we said before we based our application on `YOLOv8`, using the default network configuration and instruct It for our task.

4.3.2 Dataset Preparation

For this project, we utilized a labeled traffic sign dataset containing **4,969 samples**, we managed to find the dataset by using the `Roboflow Universe` community. Our dataset can be accessed by using the following link **Self Driving Dataset**. The `Roboflow Universe` is the largest collection of open source computer vision datasets and APIs. The dataset was well-structured and divided into three subsets:

- **Train**: All the images used only for the model training;
- **Validation**: For hyperparameter tuning and preventing overfitting;
- **Test**: For final evaluation and performance metrics, this Is used after validating the model.

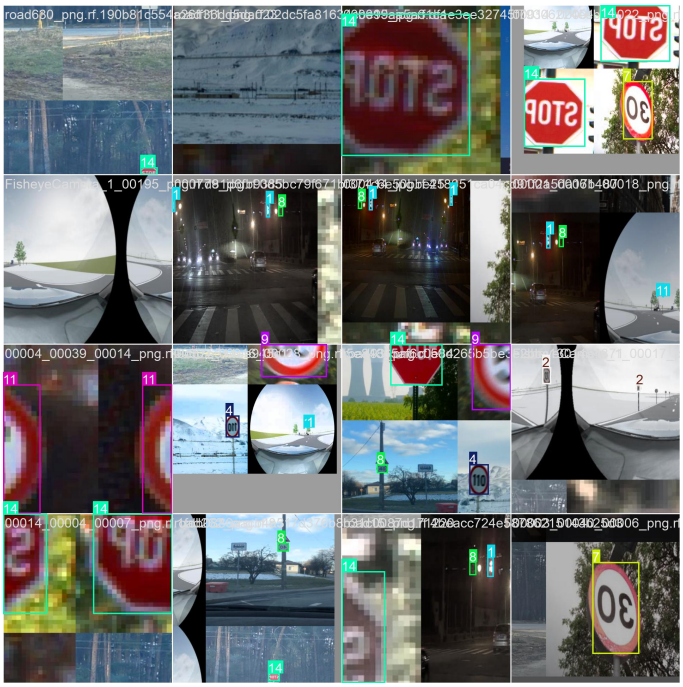

Figure 5: Train batch

Figure 6: Validation batch

The dataset includes **15 traffic sign classes**, covering both traffic lights and speed limit signs:

- **Traffic Lights**: Green Light, Red Light
- **Speed Limits**: Speed Limit 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120
- **Stop Sign**: Stop

To prepare the dataset for training, the images were preprocessed and stored following the structure required by the **YOLOv8** framework. The `data.yaml` configuration file defined the class names and paths to the training, validation, and test datasets.

```
Image_dir = './car/train/images'
num_samples = 9
image_files = os.listdir(Image_dir)
rand_images = random.sample(image_files, num_samples)

fig, axes = plt.subplots(3, 3, figsize=(11, 11))
for i in range(num_samples):
    image = rand_images[i]
    ax = axes[i // 3, i % 3]
    ax.imshow(plt.imread(os.path.join(Image_dir, image)))
    ax.axis('off')
plt.tight_layout()
plt.show()
```

4.3.3 Training Phase

We trained the **YOLOv8** model using the pre-trained weights (`yolov8n.pt`) provided by the **Ultralytics** library. Training was conducted for **30 epochs** with an automatic batch size and optimizer selection.

The training code is as follows:

```python
from ultralytics import YOLO
final_model = YOLO('yolov8n.pt')
results = final_model.train(data="./car/data.yaml", epochs=30, batch=-1,
↪  optimizer='auto')
```

During training, the YOLOv8 framework generated real-time performance metrics and saved outputs in the directory `./runs/detect/train3`. These metrics include:

Confusion Matrix The **Confusion Matrix** is a performance evaluation tool commonly used in machine learning and statistics to assess the accuracy of a classification model. It provides a summary of prediction results on a classification problem, helping to understand the performance of a model in terms of correct and incorrect predictions across various classes.

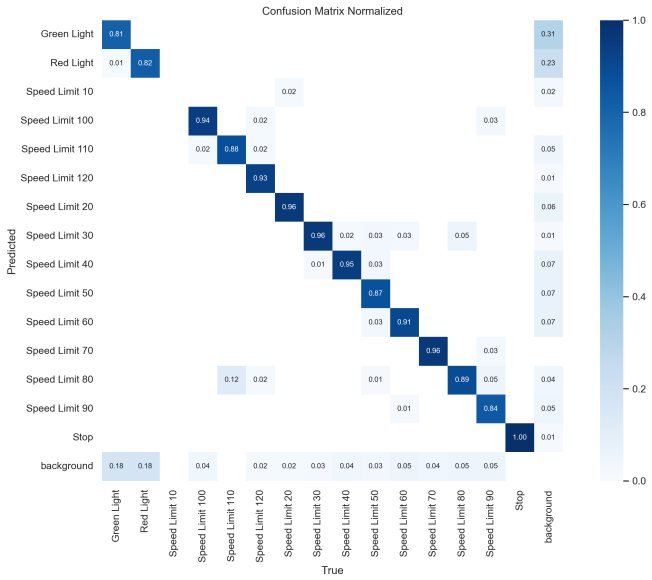

Figure 7: Confusion Matrix Normalized

The image represents a normalized Confusion Matrix for the YOLOv8 model, illustrating its classification performance across different traffic sign classes. Each cell in the matrix indicates the proportion of predictions made for a given class relative to the total occurrences of that class. The cells along the main diagonal represent correct predictions, while off-diagonal values highlight misclassifications, showing potential confusion between classes. For instance, the model accurately recognizes most of the signs, but as we can see there is some confusion between classes such as *Green Light* and *Red Light*.

Precision-Recall (PR) Curve The **Precision-Recall (PR) Curve** The Precision and Recall are performance metrics that apply to data retrieved from a collection, corpus or sample space, **Precision** (also called positive predictive value) is the fraction of relevant instances among the retrieved instances, instead the **Recall** (also known as sensitivity) is the fraction of relevant instances that were retrieved.

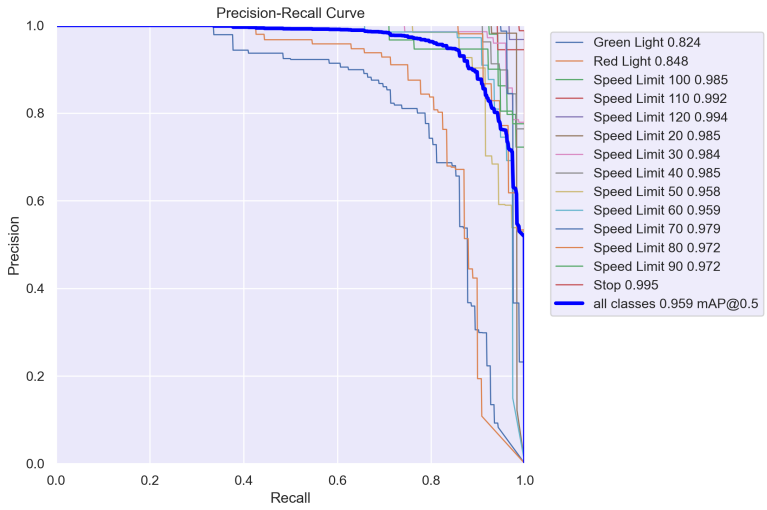

Figure 8: Precision-Recall Curve

Each curve represents the performance of a specific traffic sign class, with the "Stop" sign achieving the highest precision and recall (0.995), indicating excellent detection accuracy. Other classes, such as "Speed Limit 120" and "Speed Limit 110," also perform exceptionally well with mAP values above 0.99. The overall mAP@0.5 for all classes is 0.959, showcasing a strong balance between precision and recall across all categories.

F1 Score Curve The F-score or F-measure is a measure of predictive performance, It is calculated from the precision and recall of the test, the F1 score is the **harmonic mean** of the precision and recall. It thus symmetrically represents both precision and recall in one metric.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

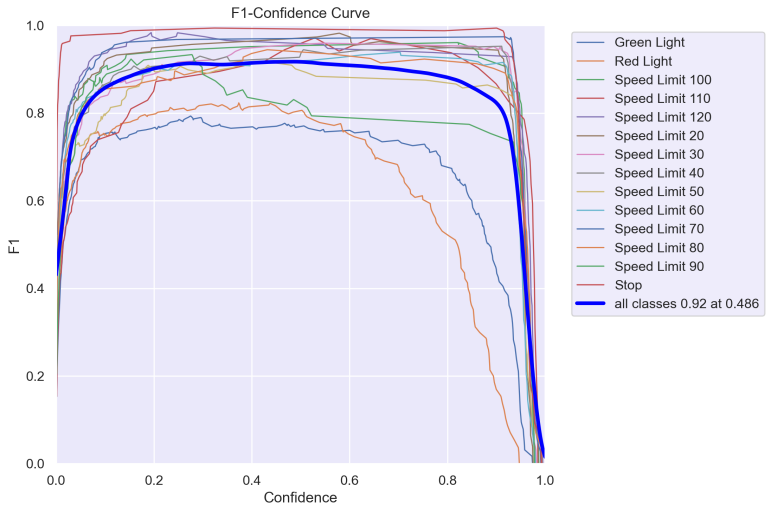

Figure 9: F1 Curve

The thick blue line represents the overall performance, with a maximum F1 score of 0.92 at a confidence of 0.486. Classes like "Stop" and "Speed Limit 120" maintain high F1 scores

consistently, while others, like "Green Light" and "Red Light," show greater variability at higher confidence levels.

Precision Curve The **Precision** (also called positive predictive value) is the fraction of relevant instances among the retrieved instances:

$$Precision = \frac{Relevant\_retrieved\_instances}{All\_retrieved\_instances} \tag{4}$$

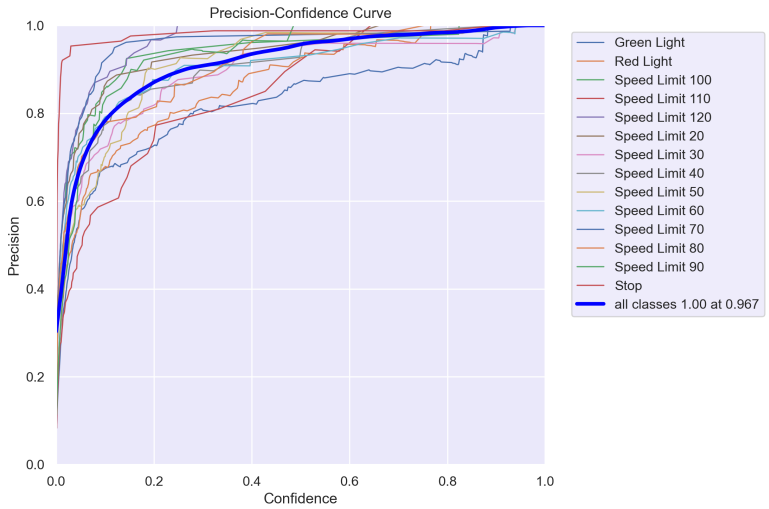

Figure 10: Precision Curve

The thick blue line represents the overall precision, reaching a perfect score of 1.00 at a confidence level of 0.967. Most classes exhibit high precision across the confidence spectrum, with "Stop" and "Speed Limit 120" maintaining consistently strong performance, while "Green Light" and "Red Light" show slightly more variability at lower confidence levels.

Recall Curve The **Recall** (also known as sensitivity) is the fraction of relevant instances that were retrieved:

$$Precision = \frac{Relevant\_retrieved\_instances}{All\_relevant\_instances} \tag{5}$$

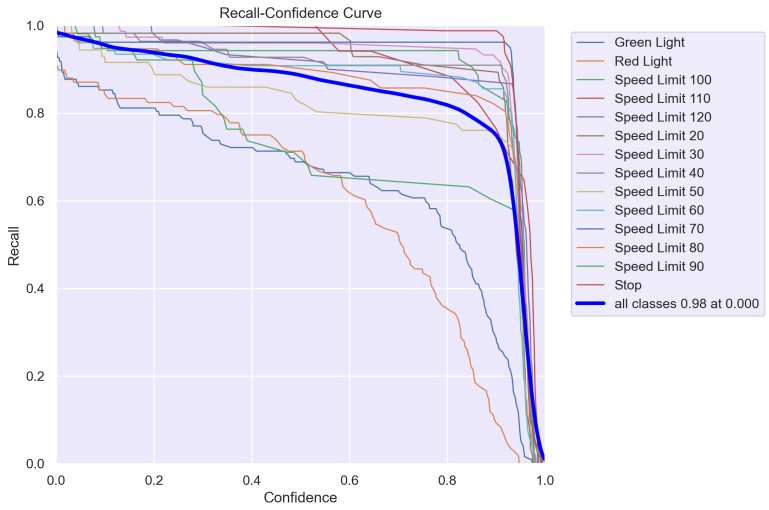

Figure 11: Recall Curve

The thick blue line indicates the overall recall, achieving a value of 0.98 at a confidence of 0.0. Higher recall values are maintained across most classes at lower confidence levels, with a noticeable decline as confidence increases. Classes like "Stop" and "Speed Limit 120" exhibit strong recall performance, while "Green Light" and "Red Light" display greater sensitivity to increasing confidence thresholds, causing more drops in recall.

4.3.4 Results and Evaluation

The YOLOv8 model was evaluated on the validation dataset using the best-performing weights after completing the training phase. Below are the metrics obtained during the evaluation process:

- **Precision (B)**: 94.78%
- **Recall (B)**: 89.27%
- **mAP@50 (B)**: 95.94%
- **mAP@50-95 (B)**: 83.30%

Class	Precision	Recall	mAP@50	mAP@50-95
Green Light	85.3%	71.3%	82.4%	48.4%
Red Light	90.7%	71.8%	84.8%	53.7%
Speed Limit 100	96.4%	94.2%	98.5%	89.8%
Speed Limit 110	89.5%	100%	99.2%	92.5%
Speed Limit 120	100%	92.3%	99.4%	91.6%
Speed Limit 20	95.6%	98.2%	98.5%	87.6%
Speed Limit 30	95.0%	95.9%	98.4%	92.7%
Speed Limit 40	92.4%	92.7%	98.5%	88.8%
Speed Limit 50	98.4%	84.9%	95.8%	85.9%
Speed Limit 60	92.7%	90.8%	95.9%	87.6%
Speed Limit 70	97.8%	96.2%	97.9%	89.6%
Speed Limit 80	98.1%	90.6%	97.2%	86.1%
Speed Limit 90	96.4%	71.2%	97.2%	79.4%
Stop	98.8%	99.6%	99.5%	92.5%

Table 1: Per-class performance metrics of the YOLOv8 model.

Analysis The YOLOv8 model performed exceptionally well across most metrics and classes. Below are some key observations:

- **Overall Performance:** The model achieved a high overall **precision (94.78%)** and **recall (89.27%)**, ensuring reliability in both minimizing false positives and detecting most relevant objects.
- **Detection Accuracy:** The **mAP@50 (95.94%)** score reflects strong accuracy in detecting and classifying traffic signs. The slightly lower **mAP@50-95 (83.30%)** is expected due to stricter IoU thresholds.
- **Class Performance:** The **Stop** sign class exhibited the highest detection accuracy with near-perfect scores. Conversely, the **Green Light** and **Red Light** classes showed comparatively lower performance, likely due to occlusions or subtle variations.

4.3.5 Exporting the Model

Finally, the trained model was exported in the **ONNX** format. `ONNX` provides an *open source format* for AI models, both deep learning and traditional ML (`ONNX Format`). It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types to ensure compatibility with deployment platforms:

```
Valid_model.export(format='onnx')
```

In this way we are able to use the new trained network for out task, by doing so the new network knows what It has to look for in an image, and how to put the best bounding box for a specific object. So now if we want to use the new network we can import It as:

```
model = YOLO("./best.pt")
```

As we can see from the above code, now we are able to load our weights for the network, that It is specifically designed to detect *Traffic Signs* and *Traffic Lights*.

5 In-Vehicle Communication

5.1 Vehicle Network

Modern vehicles are marvels of technological integration, with a plethora of components such as cameras, sensors, and control units working in harmony to ensure safety, efficiency, and comfort. However, this seamless interaction is the result of a significant evolution in how in-vehicle networking is structured.

5.1.1 Legacy In-Vehicle Networking

n the early days of automotive technology, each component within a vehicle was interconnected through dedicated point-to-point wiring. For example, a camera system might be directly wired to a display unit, while sensors like proximity detectors or accelerometers would have their own separate connections to control modules or processing units. This design approach resulted in several challenges:

- **Complexity in Wiring**: As the number of components increased, the wiring harnesses became increasingly intricate and bulky. This not only added weight to the vehicle but also made assembly, maintenance, and troubleshooting more difficult;
- **Scalability Issues**: Adding new features or components required additional wiring, which further compounded the complexity and costs;

- **Signal Interference**: The proximity of multiple wires could lead to electromagnetic interference, potentially degrading signal quality and reliability;
- **Integration Challenges**: Components often used proprietary communication protocols, making it difficult for different systems to exchange data effectively.

This decentralized and fragmented approach made it clear that a more efficient and standardized solution was needed.

5.1.2 Modern In-Vehicle Networking

The adoption of Ethernet-based networking has revolutionized in-vehicle communication, addressing many of the challenges posed by traditional wiring systems. Ethernet, a technology well-established in IT and industrial applications, has been adapted to meet the stringent requirements of the automotive environment. Here's how this transformation has simplified and enhanced in-vehicle networking:

- **Single Network Backbone**: Instead of point-to-point connections, components like cameras, sensors, and control units can now communicate over a single Ethernet backbone. This drastically reduces the amount of wiring required, cutting down on weight and complexity;
- **High Bandwidth** Automotive Ethernet supports high-speed data transfer, enabling the seamless exchange of large data sets, such as high-definition video from cameras or real-time data from LiDAR sensors;
- **Interoperability**: Standardized protocols, such as Audio Video Bridging (AVB) and Time-Sensitive Networking (TSN), ensure compatibility and synchronization between components from different manufacturers;
- **Flexibility and Scalability**: Ethernet networks can be easily expanded to accommodate new features or devices without the need for extensive rewiring.

Modern vehicles often feature a centralized gateway that acts as the hub of the Ethernet network. This gateway connects various subsystems, such as advanced driver assistance systems (ADAS), infotainment, and body electronics. Data flows efficiently across this network, enabling real-time decision-making and enhancing the overall driving experience. For instance, a camera can send high-definition video data to both the driver's display and the ADAS control unit simultaneously. Similarly, sensors can share critical information with multiple systems, such as collision avoidance and adaptive cruise control, without needing duplicate connections.

5.2 Using MQTT for inter vehicle communication

In modern vehicle networks, the evolution of Ethernet-based systems opens the door for advanced communication protocols like **MQTT** (Message Queuing Telemetry Transport). By leveraging **MQTT**, it is possible to create a robust, lightweight, and asynchronous communication system that interconnects various components and systems inside a vehicle.

5.2.1 How does MQTT work

MQTT is a publish-subscribe messaging protocol designed for efficient communication in environments with limited bandwidth or high latency. Its lightweight nature makes it ideal for vehicle networks. Instead of direct, synchronous communication between components (e.g., point-to-point messaging), **MQTT** allows devices to exchange messages indirectly through a central broker. It uses a Publish-Subscribe Model where different components act as **publishers** or **subscribers**, and in order to use this protocol It is strictly necessary to us a **Central Broker**. The *broker* acts as an intermediary that routes messages from **publishers** to all intended subscribers, It ensures that messages are delivered only to components that need them. All the subscribed devices do

not need to wait for a direct response or maintain constant connections with other components, this allow to all the different devices to operate independently and then react to new information when they arrive.

5.2.2 Our device structure

Now that we have a clearer understanding of what MQTT is and how it works, we will describe how we achieved our goal by implementing this technology in our system. To use the MQTT protocol, we need a broker. There are several ways to set this up. One option is to use a generic cloud provider, rent a server, and install the MQTT broker, along with the necessary configurations for managing the message queue, topics, and so on. Alternatively, we could use a free, open broker that does not require any complex setup for managing the queue. We chose to use a free broker, specifically the Eclipse Broker. This broker allows different clients to connect and subscribe automatically, facilitating the seamless delivery of messages to all subscribed users.

Client - The Driver In the current architecture, the client (the driver) is subscribed to an MQTT broker and is actively listening for messages under the topic **vehicle/detections**. The client understands that any information regarding detected objects will be sent under this topic by the YOLO architecture. So the client connects to the MQTT broker and It subscribes under the specific topic.

```
MQTT_BROKER = "mqtt.eclipseprojects.io"
MQTT_PORT = 1883
MQTT_TOPIC = "vehicle/detections"

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT broker!")
        client.subscribe(MQTT_TOPIC)
    else:
        print(f"Connection error: {rc}")
```

Now that the driver is subscribed to a specific topic, we need to define a handler for all the input messages received under the `MQTT_TOPIC`. This method will be called each time a new message is received. Under the subscribed topic, we will receive all the information about the objects detected by YOLO. Since the messages are transmitted over a network, we require a specific format for them. We have chosen to use the **JSON** format, which allows us to easily **load** and **dump** the messages using the `json` library in Python. After receiving a message, we parse it from **JSON** into a **dictionary**. We can then access the key `label`, which contains the class of the object detected by the YOLO network. In this way the client will be informed in real time about all the information captured by the RGB camera and If any traffic signs or traffic light is detected, the client will be instantly informed.

```
def on_message(client, userdata, msg):
    try:
        message = json.loads(msg.payload.decode('utf-8'))
        print(message['label'])
    except json.JSONDecodeError as e:
        print(f"Parsing error: {e}")

mqtt_client = mqtt.Client()

mqtt_client.on_connect = on_connect
mqtt_client.on_message = on_message
mqtt_client.connect(MQTT_BROKER, MQTT_PORT, 60)
```

Camera Module Now that the client is connected to the broker at the right topic, It is now fundamental to set up the Camera Model. The Camera defines a camera call back, where each captured image is sent inside this function to handle a possible callback. Inside this function we take the input image, apply some operations on the computer image and then use the YOLO model for detecting a possible object inside the image by using the `model.predict` method, If a class is detected, we **marshall** the message in a JSON format and we then call the `send_mqtt_message` for sending the **JSON** message to the MQTT broker under a specific topic, in this way all the clients subscribed at that broker under the `MQTT_TOPIC` will receive this message.

```
def camera_callback(image):
    img_array = np.frombuffer(image.raw_data, dtype=np.uint8)
    ...
    results = model.predict(img_array, conf=0.6)
    detections = []
    for box in results[0].boxes:
        label = model.names.get(box.cls.item())
        detection = {"label": label}
        detections.append(detection)
   if detections:
        for detection in detections:
            send_mqtt_message(detection
```

Before defining the camera callback, It is important to establish the Camera module's connection to the broker by calling the `client.connect` method. This connection should be made under the same **MQTT** broker and specific **MQTT Topic**. By doing this, all clients will be notified of any new detections made by the YOLO model. Once the client connection is set up, we implemented a function called `send_mqtt_client`. This function takes the input data to be sent, formats it in **JSON** format, and then uses the publish method to send the **JSON** data to the MQTT broker. This function is called each time a new detection is done, In this way we can provide real time feedback to the user.

```
MQTT_BROKER = "mqtt.eclipseprojects.io"
MQTT_PORT = 1883
MQTT_TOPIC = "vehicle/detections"

mqtt_client = mqtt.Client()
mqtt_client.connect(MQTT_BROKER, MQTT_PORT, 60)

def send_mqtt_message(data):
    global last_message
    try:
        if data != last_message:
            payload = json.dumps(data)
            mqtt_client.publish(MQTT_TOPIC, payload)
            last_message = data
            print(f"Sent Message: {data}")
    except Exception as e:
        print(f"Error for MQTT: {e}")
```

By organizing the entire architecture in this manner, we can separate the different modules based on their specific functions. This design enhances the system's extensibility; for instance, if we want to change the camera type or add functionality with a camera that points to the rear of the vehicle, we simply need to create a new camera module tailored for that purpose. Using the MQTT protocol, the driver can subscribe to a different topic to receive information about the rear camera. This approach makes the entire vehicle system very flexible. Currently, a single vehicle can be equipped with hundreds of sensors. By employing this architecture, we can distinguish

and operate each sensor based on its functionality, allowing them to work independently without causing issues for the other sensors.

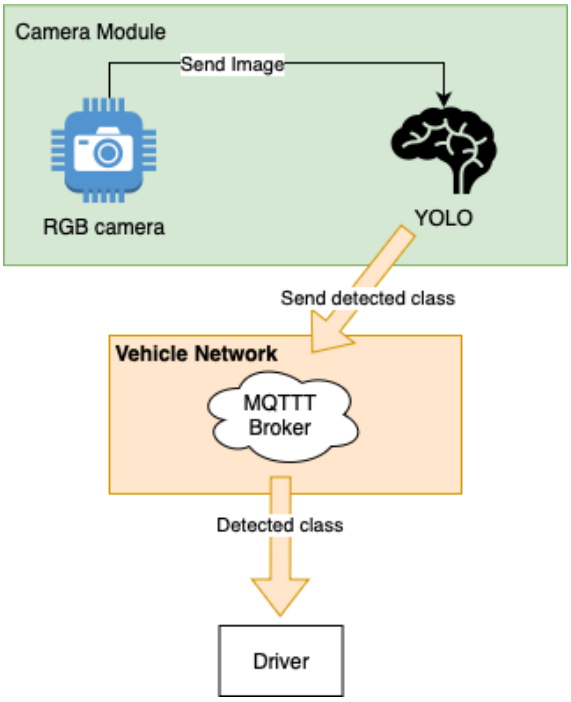

Figure 12: MQTT connection between Sensor and Driver

6 Conclusion

In conclusion, our project successfully achieved its goal of real-time detection and communication of traffic-related objects using advanced technology. By integrating `YOLOv8` with an **RGB** camera, we developed a system capable of processing images in **real time** to detect various traffic signs and signals with remarkable accuracy. The detected objects were communicated seamlessly to the driver using the `MQTT` protocol, ensuring an efficient and asynchronous flow of critical information. Our primary objective was to establish reliable real-time detection while ensuring robust communication between the image processing system and the driver's interface. We accomplished this by leveraging YOLOv8's fast and efficient detection capabilities alongside MQTT's lightweight and decoupled messaging framework. The system demonstrated excellent performance, delivering near-instantaneous detection and feedback, which is essential for applications in traffic safety and autonomous vehicle systems. This project highlights the potential of combining state-of-the-art AI models with modern communication protocols to create intelligent and responsive systems that address real-world challenges.