Smart Vehicular Systems
Pedestrian Protection System Report

Mert Akpinar
mert.akpinar@studio.unibo.it

Benedetta Pacilli
benedetta.pacilli@studio.unibo.it

Valentina Pieri
valentina.pieri5@studio.unibo.com

March 2025

Contents

Chapter 1

Introduction

1.1 The Project

This report details our **Pedestrian Protection System** developed within the CARLA simulation environment. The system's primary goal is to detect pedestrians in front of the vehicle and activate safety measures to reduce collision risks.

The system is implemented through a Python Jupyter notebook that sets up a simulated world in CARLA with random walking pedestrians and a vehicle. Two cameras are mounted at the front of the vehicle:

- **RGB Camera**: Continuously captures video frames. These frames are processed in real time by a YOLOv8 model to detect pedestrians.
- **Depth Camera**: Positioned at the same location as the RGB camera, this camera computes the distance from the vehicle to the detected pedestrians.

Based on the pedestrian's proximity, the system takes the following actions:

1. When a pedestrian is detected at a moderate distance, the system commands the vehicle to *slow down* while allowing the driver to maintain control over both direction and speed.
2. If a pedestrian is detected within a critical threshold, the vehicle *stops* completely. In this situation, the vehicle permits only reverse motion to safely walk away from the pedestrian.

Each of these scenarios triggers distinct warning messages that are sent over MQTT, ensuring that real-time alerts are available for monitoring and logging purposes.

1.2 Motivations

Road safety is a critical global challenge, with pedestrians being among the most vulnerable road users. According to the *Global Status Report on Road Safety*

2023 by the World Health Organization, an estimated 1.19 million people died in road traffic crashes in 2021, corresponding to a fatality rate of approximately 15 deaths per 100,000 population [4]. These figures underscore the urgent need for systems that can help mitigate such tragedies by protecting those on foot.

In Italy, the situation remains equally pressing. Data from the *Road Accidents Base 2023* report indicate that in 2023 there were:

- 166,525 road accidents,
- 3,039 fatalities, and
- 224,634 injuries.

While there was a slight decrease in fatalities (a reduction of 3.8% compared to the previous year), the numbers of accidents and injuries have shown a minor increase (+0.4% and +0.5%, respectively) [1]. Moreover, the fatality rate in Italy stands at 51.5 deaths per one million inhabitants, which is significantly higher than the EU average of 45.4 deaths per million inhabitants.

These alarming statistics motivate the development of our Pedestrian Protection System, which aims to:

- Enhance pedestrian safety by providing early warnings and automatic deceleration.
- Reduce the likelihood of collisions through timely intervention based on real-time perception.
- Contribute to the broader efforts in developing Advanced Driver Assistance Systems (ADAS) that can ultimately lower road traffic fatalities and injuries.

Thanks to a lightweight and efficient YOLOv8-based detection algorithm combined with depth estimation, our system demonstrates how modern computer vision techniques can be integrated into vehicular safety applications to address road safety challenges.

Chapter 2

Requirements

This project's requirements can be split into two main categories: functional requirements (what the system has to do) and non-functional ones (requirements the system has to have).

2.1 Functional Requirements

The Pedestrian Protection System must fulfill the following functional requirements:

1. **Real-Time Environment Setup:**
   - Establish a connection with the CARLA simulator.
   - Spawn a vehicle and some pedestrians (walkers) at random spawn points.
2. **Image Acquisition and Processing:**
   - Capture real-time video streams using an RGB camera.
   - Capture depth information using a Depth camera.
   - Pre-process images (resizing, normalization, and padding) for input to the detection model.
3. **Pedestrian Detection:**
   - Use a YOLO model to detect pedestrians in the RGB image stream.
   - Only consider detections with a confidence level above a certain threshold (0.4 in our case as explained in Section 4.1.1) and with the class corresponding to `person`.
4. **Distance Estimation:**

- Calculate the distance to detected pedestrians using data from the Depth camera.
- Determine the proximity of a pedestrian by mapping depth values to metric distances.

5. **Vehicle Control and Safety Interventions:**
   - The system continuously computes a Time-to-Collision (TTC) metric based on the distance to a detected pedestrian and the current vehicle speed.
   - When the TTC indicates that the available reaction time is moderately low, the vehicle gradually decelerates.
   - If the TTC falls below a critical threshold, the system engages emergency braking and restricts forward control, permitting only reverse motion.
6. **MQTT Event Publishing:**
   - Publish distinct warning messages over MQTT based on the level of danger (e.g., soft warning for slowing down, emergency warning for braking).
7. **User Input Integration:**
   - Support both keyboard input and a dedicated controller (e.g., Logitech steering wheel) for manual override.

2.2 Non-Functional Requirements

In addition to the core functionalities, the system must meet the following non-functional requirements:

1. **Real-Time Performance:**
   - Ensure low-latency processing for real-time detection and vehicle control.
   - Synchronize image acquisition, processing, and control loops using CARLA's tick mechanism.
2. **Robustness and Fault Tolerance:**
   - Implement error handling for external connections (CARLA and MQTT).
   - Ensure the system can gracefully handle missing data (e.g., absence of images) or sensor failures.
   - Ensure adequate operation regardless of the weather (sun, fog, rain) and light conditions (day/night) encountered.

3. **Scalability and Modularity:**

   - Structure the code into modular functions for easy updates and debugging.
   - Enable easy expansion (e.g., additional sensor types or new warning modes) without major code restructuring.

4. **User Interaction and Safety:**

   - Provide clear visual feedback for detected pedestrians via overlayed bounding boxes.
   - Ensure that warning messages and vehicle interventions do not unexpectedly override driver control except in emergency conditions.

Chapter 3

Design of Proposed Solution

This chapter presents an in-depth description of the design and implementation of the Pedestrian Protection System. The solution is divided into several key modules, including sensor configuration, image processing, pedestrian detection and distance estimation, vehicle control, and MQTT-based event publishing. A block diagram illustrating the overall architecture is provided in Figure 3.1.

3.1 System Architecture Overview

The system architecture consists of the following main components:

- **Simulation Environment:** The CARLA simulator provides a realistic urban scenario with vehicles and pedestrians.
- **Sensors:** An RGB camera and a Depth camera are mounted on the vehicle to capture visual and depth data.
- **Computer Vision Pipeline:** A YOLOv8-based model processes the RGB images to detect pedestrians, while depth data is used to estimate distances.
- **Vehicle Control Module:** Based on detection results, depth estimation, and vehicle speed, the system commands the vehicle to either decelerate, apply emergency braking, or maintain normal operation.
- **Event Communication:** MQTT is used to publish warning messages for real-time monitoring.
- **User Input Module:** The system supports both keyboard control and controller input for manual driving.

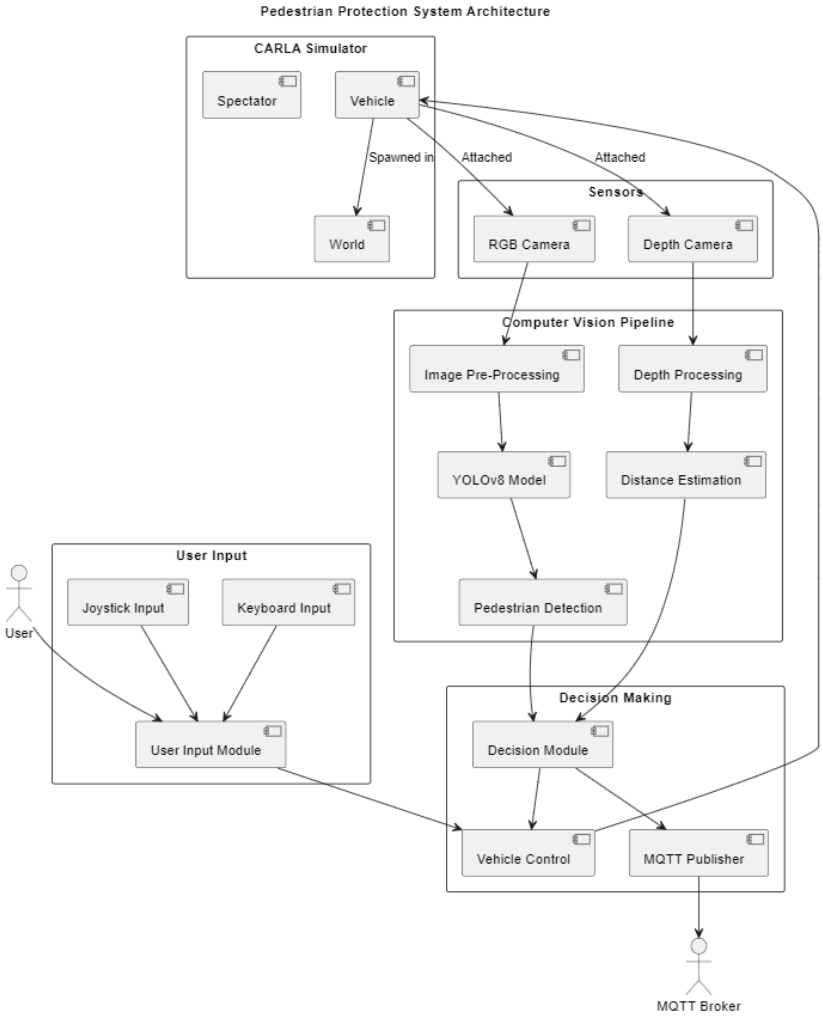

Figure 3.1: High-Level Block Diagram of the Pedestrian Protection System Architecture

3.2 Cameras Used

To perceive the environment in front of the vehicle, the system relies on two key sensors: an RGB camera and a Depth camera. Both cameras are mounted at the front of the vehicle, as shown in Figure 3.2, and share the same calibration parameters. This setup enables the system to detect pedestrians using the RGB stream while simultaneously estimating distances through the depth information.

Mounting and Transform

The cameras are attached to the vehicle using a transform defined as:

- **Location:** `x=1.0`, `y=-0.4`, `z=1.2`
- **Rotation:** Default `(pitch=0, yaw=0, roll=0)`

This positioning ensures a slightly forward and offset view from the driver's seat level, simulating a front-facing camera on the passenger side of the windshield. The transform is created via:

```
camera_transform = carla.Transform(
    carla.Location(x=1, y=-0.4, z=1.2),
    carla.Rotation()
)
```

Both cameras share the same image size, defined at runtime by:

```
rgb_bp.set_attribute('image_size_x', str(VIEW_WIDTH))
rgb_bp.set_attribute('image_size_y', str(VIEW_HEIGHT))
depth_bp.set_attribute('image_size_x', str(VIEW_WIDTH))
depth_bp.set_attribute('image_size_y', str(VIEW_HEIGHT))
```

where `VIEW_WIDTH` and `VIEW_HEIGHT` are global variables that can be set to the desired size.

Blueprint Selection and Resolution

CARLA provides a library of sensor blueprints that specify how sensors behave. In this project, the following two blueprints are used:

- **RGB Camera Blueprint:** `sensor.camera.rgb`
- **Depth Camera Blueprint:** `sensor.camera.depth`

Field of View and Calibration

Both cameras operate with a horizontal Field of View (FoV) set to:

- **FoV = 90 degrees**

This wide FoV allows the system to capture a broad, forward-facing scene, essential for detecting pedestrians who may appear on either side of the vehicle's path, as captured by Figure 3.4.

Intrinsic Calibration To map 2D image coordinates to real-world metrics (and vice versa), each camera is assigned a 3×3 calibration matrix:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

In the code, `f` (focal length) and c_x, c_y (principal point) are derived from `VIEW_WIDTH`, `VIEW_HEIGHT`, and the FoV using:

```
calibration = np.identity(3)
calibration[0, 2] = VIEW_WIDTH / 2.0
calibration[1, 2] = VIEW_HEIGHT / 2.0
calibration[0, 0] = calibration[1, 1] = VIEW_WIDTH / (
   2.0 * np.tan(VIEW_FOV * np.pi / 360.0)
)
```

This ensures that both the RGB and Depth cameras share the same intrinsic parameters, simplifying the correspondence between color and depth pixels.

RGB Camera Specifications

- **Sensor Type:** `sensor.camera.rgb`
- **Channels:** 4-channel BGRA output (though the alpha channel is often unused)
- **Frame Rate:** By default, synchronized with the CARLA world's tick
- **Usage:** Provides color frames for the pedestrian detection pipeline (YOLOv8)
- **Data Processing:** The raw image buffer is converted into a NumPy array, reshaped, and passed to the detection model after resizing/normalization

Depth Camera Specifications

- **Sensor Type:** `sensor.camera.depth`
- **Channels:** 4-channel BGRA, where the RGB components encode depth
- **Frame Rate:** Also synchronized with CARLA's tick
- **Depth Encoding:** The depth camera outputs a normalized depth value, which is then scaled to real-world meters using:

$$\text{depth\_in\_meters} = \text{normalized\_depth} \times 1000 \tag{3.2}$$

- **Usage:** Computes the distance to detected pedestrians by sampling the depth value at the bounding box's centroid in image space

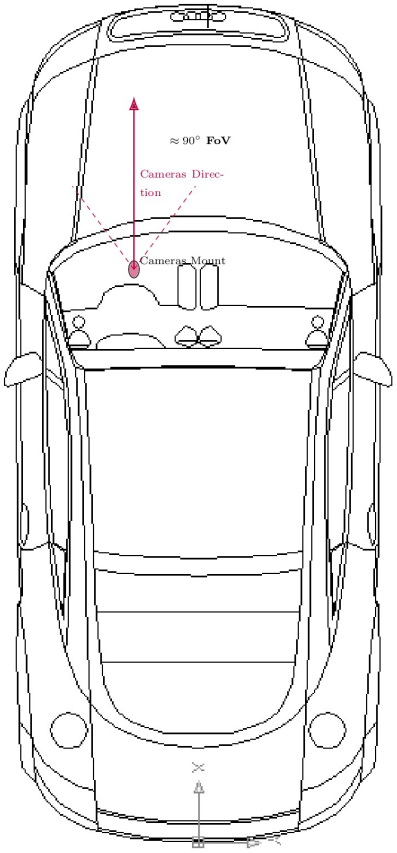

Figure 3.2: RGB and Depth cameras positioning on the vehicle.

Data Flow and Listeners

After spawning the cameras, each is instructed to listen for new frames:

```
rgb_camera.listen(lambda image: set_rgb_image(image))
depth_camera.listen(lambda image: set_depth_image(image))
```

This `listen()` method registers a callback function that processes incoming frames. As soon as a new image arrives, the system converts it to a NumPy array and stores it for further processing in the pedestrian detection and distance estimation pipelines.

RGB Camera View

Figure 3.3 illustrates the real-time view captured by the front-facing RGB camera mounted on the vehicle. This perspective closely resembles a driver's field

of vision, enabling the system to detect pedestrians who may enter the vehicle's path. As seen in Figure 3.4, the system draws a **green bounding box** around each identified pedestrian, allowing for clear visual confirmation of detections.

Figure 3.3: Real-time RGB camera view in the CARLA environment.

Figure 3.4: Wide FoV of camera resulting into broad visualization scene. Green bounding box for clear visual confirmation of detections.

Pedestrian Detection in Various Weather and Light Conditions

The Pedestrian Protection System is designed to operate reliably under diverse environmental scenarios. By using the YOLOv8 model and continuous depth sensing, the system can accurately detect pedestrians even when visibility is compromised or multiple pedestrians appear simultaneously. Figure 3.5, Figure 3.6, Figure 3.7, Figure 3.9, and Figure 3.10 illustrate examples of detection results under different weather and lighting conditions.

In cases of foggy weather, visibility and detection capabilities depend highly on how dense the fog is, as it can be seen in Figures 3.8 and 3.9.

Figure 3.5: Pedestrian detection under clear, sunny conditions.

Figure 3.6: Pedestrian detection at night. A green bounding box outlines a single pedestrian despite low-light conditions.

Figure 3.7: Pedestrian detection during rainfall.

3.3 Computer Vision Pipeline

For pedestrian detection, the system employs the **YOLOv8n** model provided by *Ultralytics*[1].

YOLO (*You Only Look Once*) is a family of real-time object detection mod-

[1] `https://github.com/ultralytics/ultralytics`

Figure 3.8: Pedestrian detection in a dense fog.

Figure 3.9: Pedestrian detection in a light fog.

els first introduced by Redmon et al. [5]. Over successive iterations (YOLOv2, YOLOv3, YOLOv5, etc.), the architecture has been refined for speed and accuracy. YOLOv8 represents one of the latest generations, offering:

- **Improved Backbone and Neck:** Incorporates CSP (Cross-Stage Partial) connections and PANet for feature aggregation.
- **Lightweight Variants:** *Nano (n), Small (s), Medium (m),* etc. The

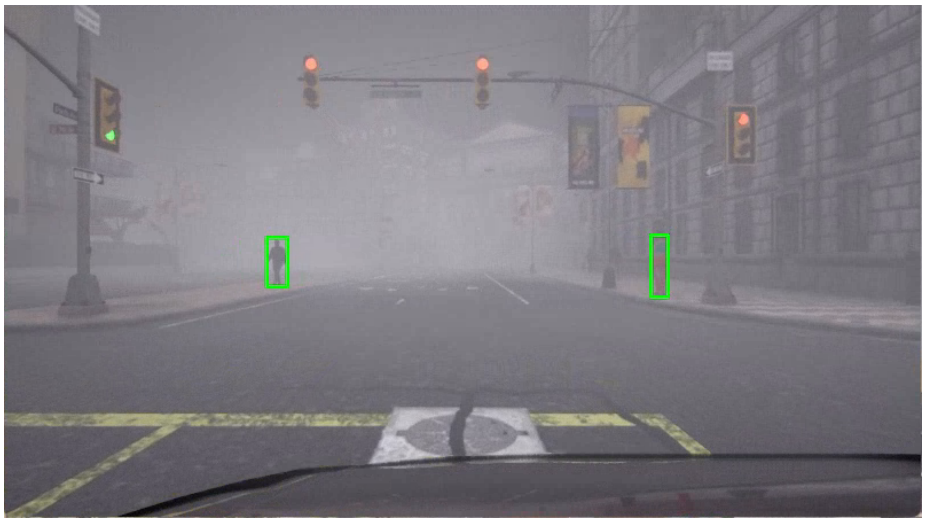

Figure 3.10: Multiple pedestrian detection in foggy conditions. Each detected individual is highlighted by a separate bounding box.

`n` variant used here (YOLOv8n) is optimized for real-time inference on limited hardware.

- **Flexible Input Sizes:** Can be trained or fine-tuned on various resolutions, with 640×640 as a common default.

While experimental weight files for YOLOv10 and YOLOv11 are available, we have chosen to use YOLOv8 in this project for several reasons:

- **Stability and Maturity:** YOLOv8 is a well-established model that has undergone extensive testing in numerous real-time applications. Its performance and accuracy have been validated by the community, making it a reliable choice for a safety-critical system.
- **Official Support and Compatibility:** The current version of the Ultralytics library (v8.3.82) fully supports YOLOv8. In contrast, weight files for YOLOv10 and YOLOv11 expect model definitions that are not present in the official library release, leading to compatibility issues.
- **Real-Time Efficiency:** YOLOv8, particularly the nano variant (YOLOv8n), strikes an excellent balance between computational efficiency and detection accuracy, making it suitable for real-time pedestrian detection and vehicle control in dynamic environments.
- **Ease of Integration:** Using the officially supported YOLOv8 model minimizes integration challenges, allowing us to focus on the overall system performance and safety.

Model Retrieval and Integration: The model weights (`yolov8n.pt`) can be downloaded from Ultralytics' repository[2] and loaded into the Python script via:

```
from ultralytics.models.yolo import YOLO
model = YOLO("yolov8n.pt")
```

This enables direct inference on incoming frames using:

```
results = model(image)
```

where `image` is the pre-processed NumPy array described in Section 3.2.

Detection Process

1. **Inference:** The YOLOv8n model outputs bounding boxes, class labels, and confidence scores for each detected object.
2. **Filtering:** Only predictions with the `person` class (index 0) and a confidence score above 0.2 are retained.
3. **Centroid Extraction:** For each retained bounding box (x_1, y_1, x_2, y_2), a centroid is computed as:

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

   This pixel location is then used to estimate distance from the depth camera.

3.3.1 Distance Estimation and TTC Calculation

In parallel with the RGB stream, a Depth camera provides a 4-channel BGRA image, where the RGB channels encode the depth information. The system maps each detected pedestrian's centroid to the corresponding pixel in the depth image and calculates the distance in meters using the following process:

```
blue  = depth_array[y, x, 0]
green = depth_array[y, x, 1]
red   = depth_array[y, x, 2]

normalized_depth = (red + green * 256 + blue * 256**2) / (256**3 - 1)
depth_in_meters = normalized_depth * 1000.0
```

This computation converts the 24-bit encoded depth into a physical distance. However, rather than relying solely on this raw distance, our implementation calculates the Time-to-Collision (TTC) as:

$$\text{TTC} = \frac{\text{distance (m)}}{\text{vehicle speed (m/s)}}$$

[2] `https://github.com/ultralytics/assets/releases/download/v8.1.0/yolov8n.pt`

The TTC metric integrates both distance and vehicle speed, providing a dynamic indicator of collision risk. The control module then uses defined TTC thresholds to determine the appropriate intervention:

- **Emergency Braking:** When TTC is less than 1.0 second, the system engages full emergency braking.
- **Soft Braking:** When TTC is between 1.0 and 2.0 seconds, a softer braking action is applied as an early warning.

Unlike raw distance measurements, Time-to-Collision (TTC) provides a dynamic metric that incorporates the vehicle's current speed.

For a given distance, a higher vehicle speed results in a lower TTC, indicating a more urgent risk of collision. Conversely, if the vehicle is moving slowly, the same distance yields a higher TTC, suggesting that there is more time for the driver to react. TTC allows the control system to adapt its braking strategy based on the actual time available before a potential collision. This leads to more appropriate and timely interventions, such as soft braking when there is still time to decelerate gradually and full emergency braking when the available time is critically short.

Integration with Detection For each pedestrian bounding box centroid (c_x, c_y), we perform:

1. **Validity Check:** Ensure (c_x, c_y) lies within the image bounds.
2. **Depth Retrieval:** Convert the BGRA channels at (c_x, c_y) into a distance in meters using the method above.
3. **TTC Calculation and Thresholding:** If the computed TTC is below critical thresholds (e.g., less than 1.0 s for emergency or between 1.0 s and 2.0 s for soft braking) and the vehicle speed is above zero, the control module triggers the corresponding safety intervention. If the depth exceeds 100 m or is invalid, the detection is ignored.

3.4 Vehicle Control and Safety Interventions

3.4.1 Control Logic

The vehicle control module receives input from both the computer vision pipeline and the user input module. The key control decisions include:

- **Normal Driving:** When no pedestrian is within a critical distance.
- **Soft Braking:** When a pedestrian is detected within a safe but reduced distance, the system reduces the throttle and applies a small brake force to lower the chances of impact.

- **Emergency Braking:** When a pedestrian is detected at a very close distance, the system commands full braking and switches to a mode where only reverse motion is permitted. Once the car is far enough from the pedestrian, the driver will again be able to fully control the car in terms of both speed and direction.

3.4.2 Control Input Modes

Two distinct functions handle the vehicle control in our system, each catering to a different input device. Both functions ultimately command the vehicle to accelerate, decelerate, steer, or brake based on user input and safety requirements, ensuring consistent behavior regardless of the input mode.

1. **Keyboard Control:** The function `control_car` manages vehicle control when a dedicated controller is unavailable. It listens for key press events using the Pygame event loop and adjusts the vehicle's control parameters accordingly:
   - *Directional Input:* The keys `W`, `A`, `S`, and `D` are mapped to forward, left, reverse, and right commands, respectively.
   - *Braking:* Since moving forward and backward last until the opposite mode key is pressed, the key `B` is dedicated to braking to facilitate controlled deceleration.
   - *Emergency Handling:* In scenarios where a pedestrian is too close, the function is triggered in a *backwards-only* mode. Here, only reverse and braking commands are allowed, ensuring that the vehicle does not move forward in dangerous situations.
2. **Controller (Joystick) Control:** The function `control_car_with_wheel` is designed for use with dedicated controllers such as a Logitech steering wheel. It retrieves analog and digital inputs from the joystick:
   - *Steering:* The horizontal axis of the joystick (axis 0) controls the steering. The value is clamped between -1.0 and 1.0 to determine the degree of left or right turn.
   - *Throttle and Brake:* Throttle input is read from axis 3, and its maximum value is capped at 0.85 to prevent over-acceleration. Brake input is taken from axis 4; if the brake value falls within a defined range (between 0.1 and 1.0), it is applied as a braking command.
   - *Reverse Mode:* A digital button (button 5) is used to toggle the reverse mode. When active, the controller allows reverse driving by setting the reverse flag and applying the corresponding throttle value.
   - *Backwards-Only Mode:* Similar to the keyboard control, the backward-only mode is engaged under emergency conditions. In this mode, if the reverse button is pressed, the vehicle applies throttle in reverse; otherwise, normal forward control is disabled.

In both control modes, the function `move_spectator_to` is called to adjust the spectator camera, ensuring that the user maintains a clear view of the vehicle's current position and direction.

By providing an equivalent control logic across both keyboard and joystick inputs, the system maintains a consistent user experience and ensures safe, responsive control of the vehicle under varying conditions.

3.4.3 Motivations behind our choices

Our controller implementations have been designed with pedestrian safety as the foremost priority. In real-world traffic incidents, rapid driver reaction time is one of the critical factors between a minor accident and a catastrophic collision.

Based on this, our system adopts a two-tier intervention strategy, with gradual deceleration when a pedestrian is detected at a moderate distance and emergency braking with consequential partial control removal if a pedestrian remains within a critical distance.

In the less critical situation, the braking serves as an early warning to the driver, allowing for human oversight and intervention while still reducing the kinetic energy of the vehicle. Research in traffic safety indicates that early deceleration can significantly mitigate the impact force during a collision, thereby reducing the severity of injuries [6]. Meanwhile, our choice of partially and temporarily removing the driver's ability to control the car was made to prevent delayed or inappropriate responses in a high-risk situation. Accident data shows that, in severe emergency scenarios, even a split-second delay in reaction can result in fatal outcomes [3, 2].

This phased intervention strategy is also employed in several modern vehicles. For instance, Volvo's City Safety system and Toyota's Pre-Collision System both initiate a gentle deceleration when a potential collision with a pedestrian is detected, followed by full emergency braking if the threat persists. Such systems have been shown to reduce both the frequency and severity of collisions in urban environments.

3.5 MQTT Event Publishing

3.5.1 Implementation

The system uses the lightweight and low-latency MQTT protocol for real-time communication of critical events. To achieve this, we utilize the Paho MQTT client library, which provides robust features for both publishing and subscribing to topics.

- **MQTT Client Setup:** The client is configured to connect to a public broker (e.g., `test.mosquitto.org`) on the standard MQTT port (1883). During initialization, a callback function is set up to handle incoming messages, though in our application, the primary use case is publishing events.

- **Secure and Robust Connection:** Although we are using a public broker for demonstration purposes, the implementation includes managing connection failures. In production scenarios, it is advisable to use a broker with TLS support and authentication to secure the communication.
- **Event-Based Message Publishing:** Based on vehicle-predestrian distance, it publishes predefined messages, as shown in Figure 3.11:
  - **Emergency Events:** If a pedestrian is within the emergency distance threshold, the system publishes a message (e.g., "Pedestrian too close, emergency braking!"), designed to trigger immediate response actions in connected subsystems.
  - **Warning Events:** If a pedestrian is approaching but not yet critically close, a softer warning message (e.g., "Pedestrian approaching, slowing down the vehicle.").
- **Integration with the Safety System:** By publishing these event messages in real time, the MQTT mechanism enables the seamless integration of the pedestrian protection system with other vehicle subsystems or external monitoring dashboards. This communication protocol suits distributed systems where lightweight messaging is essential.

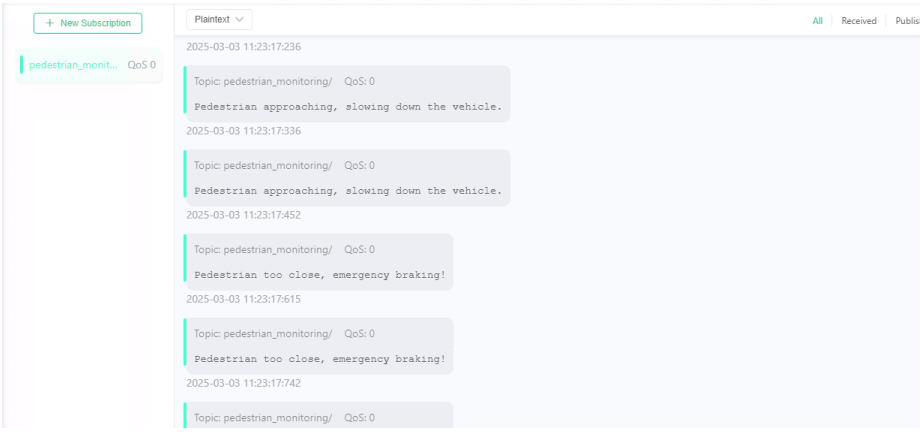

Figure 3.11: Different messages are published to the mqtt broker, based on the real-time situation.

3.5.2 Code Snippet for MQTT Publishing

Below is an excerpt of the key code segments illustrating the MQTT setup and event publishing:

```
import paho.mqtt.client as mqtt
```

```
# Define MQTT broker settings
BROKER = "test.mosquitto.org"
PORT = 1883
TOPIC = "pedestrian_monitoring/"

# Instantiate the MQTT client
mqtt_client = mqtt.Client()

# Define a callback function for incoming messages
def on_message(client, userdata, message):
    print(f"Received: {message.payload.decode()} on {message.topic}")

mqtt_client.on_message = on_message

# Attempt to connect to the broker with error handling
try:
    mqtt_client.connect(BROKER, PORT, 60)
    mqtt_client.loop_start()  # Start the network loop in a separate thread
    print(f"Listening to {TOPIC} on {BROKER}...")
except Exception as e:
    print("Connection failed:", e)

# Function to publish warning messages based on pedestrian detection
def send_warning(emergency):
    if emergency:
        mqtt_client.publish(TOPIC, "Pedestrian too close, emergency braking!")
    else:
        mqtt_client.publish(TOPIC, "Pedestrian approaching, slowing down the vehicle.")
```

Chapter 4

Testing and Results

4.1 Test Setup

To validate the performance of our Pedestrian Protection System, we conducted a series of test sessions in the CARLA simulator under four distinct weather conditions. Each test session was executed under one of the following conditions:

- **Night:**

```
night_scenario = carla.WeatherParameters(
    sun_azimuth_angle=180.0,
    sun_altitude_angle=-90.0
)
```

- **Sunny:**

```
sunny_weather = carla.WeatherParameters(
    cloudiness=0.0,
    precipitation=0.0,
    precipitation_deposits=0.0,
    wind_intensity=5.0,
    sun_altitude_angle=70.0
)
```

- **Rainy:**

```
rainy_weather = carla.WeatherParameters(
    cloudiness=80.0,
    precipitation=100.0,
    precipitation_deposits=80.0,
```

```
    wind_intensity=30.0,
    sun_altitude_angle=20.0
)
```

- **Foggy:**

```
foggy_weather = carla.WeatherParameters(
    cloudiness=70.0,
    precipitation=0.0,
    precipitation_deposits=0.0,
    wind_intensity=10.0,
    fog_density=80.0,
    fog_distance=10.0,
    fog_falloff=2.0,
    sun_altitude_angle=20.0
)
```

For each test, the system logged the following parameters in a consolidated CSV file:

- Timestamp
- Frame Number
- Vehicle Speed (m/s)
- Detection Distance (m) – the distance to the closest detected pedestrian
- Time-to-Collision (TTC, s)
- Intervention Type (e.g., Emergency Braking, Soft Braking, No Intervention)
- Number of Detections in the Frame
- Average Confidence of Detections
- Weather Condition (e.g., sunny, night, rainy, foggy)

4.1.1 Detection Confidence Threshold

During tests, we observed that using a low confidence threshold resulted in a high number of false-positive detections (i.e., non-pedestrian objects being detected as pedestrians). To improve detection reliability, we tested different threshold values and eventually set the threshold to 0.4. This change effectively filters out many non-pedestrian detections while still ensuring that all actual pedestrians in the scene are reliably detected.

4.2 Detection Distance Results

Analysis of the detection distances shows distinct trends across weather conditions, available in Figure 4.1:

- **Sunny Conditions:** Under sunny conditions, pedestrians are detected at moderate distances, thanks to high visibility.
- **Rainy and Foggy Conditions:** In these adverse conditions, detection distances are generally shorter. In the rainy case, reflections and water droplets can obscure or distort the scene, forcing the system to detect pedestrians only at closer ranges. Under the dense fog conditions we chose, visibility is highly hindered, so pedestrians are recognized only when they are near the vehicle.
- **Night Conditions:** Nighttime exhibits the widest spread of detection distances, including some very large outliers. Despite increasing the confidence threshold, the YOLO model easily mistakes non-pedestrian objects for pedestrians due to the low light level.

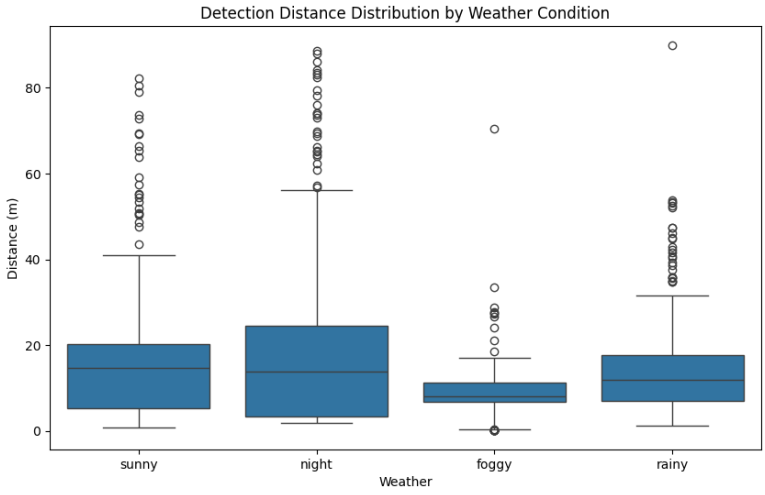

Figure 4.1: Detection distance distribution across different weather conditions.

4.3 Detection Confidence Results

Figure 4.2 presents the distribution of average detection confidence under each weather scenario:

- **Sunny:** The box plot for sunny conditions sits in a mid-to-high range. Sunny conditions provide a balance of good visibility and clear contrast; while this results in moderate-to-high confidence overall, harsh shadows or reflective surfaces can occasionally reduce confidence in some frames.
- **Night:** Paradoxically, the median confidence for night scenes can be quite high. This is because illuminated pedestrians stand out sharply against dark backgrounds, creating strong contrast that the model recognizes confidently. However, inconsistent lighting also introduces a large variance: certain frames yield very high confidence, while others produce false positives or lower confidence due to partial shadows or other objects mimicking pedestrian silhouettes.
- **Foggy:** As it can be seen from Figure 3.8, the heavy fog simplifies the usual detailed scenario of the city, so when a pedestrian is close enough for the model to detect it, the model is not usually confused by other background objects.
- **Rainy:** Rain introduces multiple visual disturbances—raindrops on the "camera" lens, reflections, and partial occlusions—degrading the model's ability to identify pedestrians reliably.

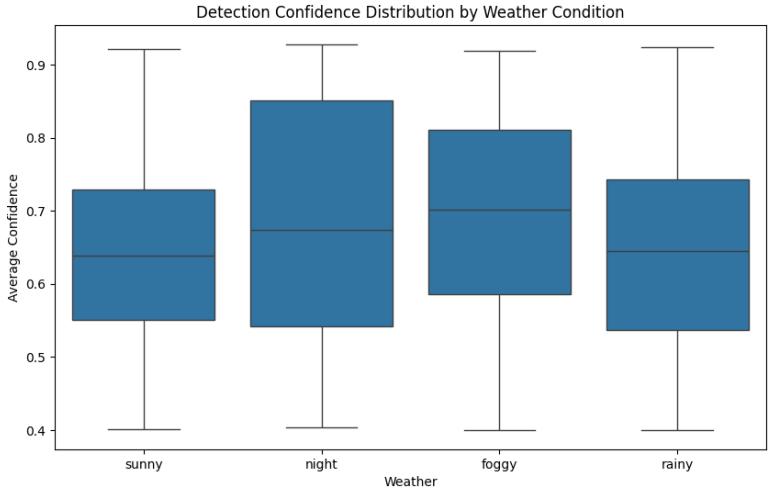

Figure 4.2: Detection confidence distribution across different weather conditions.

Chapter 5

Usage Info and Instructions

This chapter provides a comprehensive guide for setting up and running the Pedestrian Protection System. Follow these steps to ensure the system operates correctly in the CARLA simulation environment.

For any doubts regarding the usage or functionality of the functions, please refer to the in-code documentation. All functions are equipped with detailed docstrings to clarify their purpose and usage

5.1 Prerequisites

Before running the system, ensure that you have the following:

- **CARLA Simulator:** Install and launch your local machine's CARLA simulator (version 0.9.x recommended). The simulator should be accessible on `localhost` at port `2000`.
- **Python Environment:** Set up a dedicated Python environment (e.g., `carla-env`) with the following dependencies:
  - `carla`
  - `pygame`
  - `opencv-python` (cv2)
  - `numpy`
  - `ultralytics` (for YOLOv8)
  - `paho-mqtt`
  - `tkinter` (usually included with Python)
- **Code Repository:** Clone the repository containing the code by running:

```
git clone https://github.com/benedettapacilli/pedestrian-protection-system.git
```

5.2 Environment Setup

1. **Launch CARLA:** Start the CARLA simulator on your machine (e.g., by running `CarlaUE4.exe` on Windows or using the appropriate launch command on Linux). Ensure it runs on `localhost` with the default port `2000`.

2. **Install Required Dependencies:** Ensure that you have Python 3.7.x installed. Then, install the following necessary dependencies pip:

   - `carla` – for simulation control and environment access
   - `pygame` – for handling user input and creating display windows
   - `opencv-python` (cv2) – for image processing and display
   - `paho-mqtt` – for MQTT messaging and communication
   - `numpy` – for numerical operations and array manipulations
   - `ultralytics` – providing the YOLO model for object detection

5.3 Using the Notebook

1. Open the Notebook and Run the Cells Sequentially.

2. **Monitor the Camera Feed:** The notebook will display a real-time camera feed in an OpenCV window. This visual feedback assists in car manual driving and verifying that the detection pipeline is functioning correctly.

3. **Control the Vehicle:** The notebook supports both keyboard input (using Pygame) and joystick/steering wheel input (if connected), as previously explain in Sub-section 3.4.2.

Conclusions

In this project, we developed a Pedestrian Protection System within the CARLA simulation environment that integrates real-time pedestrian detection using a YOLOv8n model, depth estimation from an aligned depth camera, and a dynamic braking strategy based on Time-to-Collision (TTC). The system can detect pedestrians under various weather conditions and initiate timely interventions such as soft or emergency braking. The TTC metric, which incorporates both distance and vehicle speed, provides a more realistic measure of collision risk than raw distance alone. This approach enables the system to adapt its intervention strategy based on the available reaction time, thereby reducing the likelihood of a collision.

The system has been shown to operate reliably under various environmental conditions, although some challenges remain. For example, in adverse weather scenarios such as fog and rain, the detection range is reduced, and confidence scores can be lower, which may lead to variations in the braking interventions. Overall, the real-time performance and low-latency control demonstrated by the system highlight its potential for enhancing pedestrian safety in urban driving scenarios.

Future Works

Future work on this project should aim to extend the system's capabilities to more complex and realistic driving environments. Incorporating additional sensors, such as LiDAR or RADAR, could improve the robustness of pedestrian detection, especially in adverse weather conditions where the performance of cameras may be compromised. An interesting extension would be the integration of pedestrian trajectory forecasting, which would allow the system not only to react to current detections but also to predict pedestrian movement and adapt the intervention strategy accordingly.

Finally, while the system has been validated in the CARLA simulation, a logical next step would be to test the approach on real-world data or through hardware-in-the-loop experiments, ensuring that the benefits observed in the simulation can be translated to actual driving conditions. These improvements would contribute to the broader goal of developing advanced driver assistance systems that significantly enhance road safety and reduce collision risks.

Bibliography

[1] Istituto Nazionale di Statistica (ISTAT). Road accidents base 2023. `https://www.istat.it/en/`, 2023. Data on road accidents, fatalities, and injuries in Italy.

[2] Hoggatt Law Firm. How can driver reaction times affect your car accident case?, 2024.

[3] Anderson Scott Law. Distracted driving and delayed reactions pose a critical safety risk, 2024.

[4] World Health Organization. Global status report on road safety 2023. `https://www.who.int/teams/social-determinants-of-health/safety-and-mobility/global-status-report-on-road-safety-2023`, 2023. Available under the CC BY-NC-SA 3.0 IGO license.

[5] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[6] European Commission Road Safety. Speed and accident risk, 2024.