

Improving Self-Driving Performance using ML-MAS Framework

Abhijith Udayakumar

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Artificial Intelligence
of the
University of Aberdeen.



Department of Computing Science

2023

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

ABHIJITH UDAYAKUMAR

Date: 2023

Abstract

Self-driving cars represent a transformative technology with the potential to revolutionise transportation systems worldwide. These vehicles have the ability to operate autonomously, without human intervention, by leveraging advanced technologies such as Artificial Intelligence (AI) and Machine Learning (ML). Self-driving cars offer numerous benefits, including improved road safety, increased efficiency, reduced traffic congestion, and enhanced mobility for individuals who are unable to drive.

The success of self-driving cars heavily relies on the development of robust AI systems capable of perceiving, reasoning, and making decisions in real-time. Neuro-symbolic AI, the integration of sub-symbolic with symbolic reasoning, has emerged as a promising approach to address the complexities of autonomous driving. By combining the strengths of sub-symbolic ML and Multi-Agent Systems (MAS), Neuro-symbolic AI frameworks can achieve more reliable and intelligent decision-making in highly dynamic environments.

In this context, the current project investigates a neuro-symbolic AI framework using ML and MAS, with the aim to enhance the driving capabilities of autonomous vehicles. The ML component enables the vehicle to learn patterns and behaviours from vast amounts of data, while the MAS component provides rational decision-making and adaptive behaviour in response to complex driving scenarios.

Previous approaches to autonomous driving have predominantly focused on purely ML-based solutions ([Chen and Krähenbühl, 2022](#)) or rule-based systems. ML models, such as deep neural networks ([Dworak et al., 2019](#)), have shown remarkable capabilities in perception tasks like object recognition and lane detection. However, they often lack explicit reasoning and interpretability, making them prone to uncertainty and limited adaptability in dynamic environments. On the other hand, rule-based systems can provide rational behaviour ([Perry and Sevil, 2022](#)) but struggle to handle the complexity and variability encountered in real-world driving scenarios.

By incorporating Neuro-symbolic AI with ML and MAS, this project takes a hybrid approach that combines the learning capabilities of ML models with the rational behaviour and adaptability of agent-based systems. This integration aims to overcome the limitations of purely ML or rule-based approaches by leveraging the strengths of both paradigms. The project strives to demonstrate that a balanced approach, combining learning-based models with rational decision-making agents, can yield superior results in autonomous driving.

The project builds upon the previous work of Hilal Al Shukairi, a former MSc AI student, who developed the ML-MAS framework and showcased improved driving scores ([Shukairi and Cardoso, 2023](#)).

Acknowledgements

I would like to express my deepest gratitude and appreciation to all those who have contributed to the successful completion of this project on integrating a Neuro-symbolic AI framework using Machine Learning (ML) and Multi-Agent Systems (MAS) for enhancing the driving capabilities of self-driving cars.

First and foremost, I would like to thank my project supervisor, Dr. Rafael C. Cardoso, for his guidance, support, and invaluable insights throughout the duration of this project. His expertise and dedication have been instrumental in shaping the direction and scope of our work.

I am immensely grateful to Hilal Al Shukairi, the former MSc AI student, whose previous work on the ML-MAS framework ([Shukairi and Cardoso, 2023](#)) laid the foundation for my project. His efforts and innovative ideas have significantly contributed to my understanding and implementation of Neuro-symbolic AI in autonomous driving.

I extend my thanks to the faculty and staff of University of Aberdeen, whose resources and facilities have provided me with a conducive environment for conducting my research. Their commitment to fostering academic excellence and pushing the boundaries of knowledge has been a constant source of motivation.

I would also like to acknowledge the participants who generously shared their time and expertise, providing valuable data and insights that enriched the project.

Furthermore, I would like to express gratitude to my colleagues, friends, and family members for their unwavering support and encouragement throughout this journey. Their belief in my abilities and their words of encouragement have been invaluable in keeping me motivated during challenging times.

Finally, I would like to thank the larger scientific community for their contributions to the field of autonomous driving. Their groundbreaking research and open sharing of knowledge have paved the way for advancements in the domain, inspiring us to push the boundaries of what is possible.

Once again, I extend my heartfelt gratitude to everyone who has played a part in this project. Your contributions, big or small, have made a significant impact on its success.

Contents

| | | |
|----------|--------------------------------------------------------------------|-----------|
| 1 | Introduction | 7 |
| 1.1 | CARLA and Leaderboard | 7 |
| 1.2 | Problem Statement | 8 |
| 1.3 | Goals | 9 |
| 1.4 | Outline | 9 |
| 2 | Background | 10 |
| 2.1 | CARLA Simulator | 10 |
| 2.2 | CARLA Agents and Sensors | 10 |
| 2.3 | Python API of CARLA | 11 |
| 2.4 | CARLA Leaderboard | 13 |
| 2.4.1 | Setting up CARLA Leaderboard 1.0 | 13 |
| 2.5 | Leaderboard Metrics | 14 |
| 2.6 | The ML-MAS Framework | 15 |
| 2.6.1 | Learning from All Vehicles - LAV | 16 |
| 2.6.2 | AgentSpeak Programming - Jason | 17 |
| 2.6.3 | Setting up the ML-MAS Framework with CARLA and LAV model | 17 |
| 2.6.4 | The BDI Bridge | 18 |
| 2.6.5 | The Orchestrator | 19 |
| 2.6.6 | The Utility Notebook | 20 |
| 2.7 | Related Work | 21 |
| 3 | Methodology | 22 |
| 3.1 | Initial Run | 22 |
| 3.2 | Extended Leaderboard Run-time | 25 |
| 3.3 | Analysing Infractions | 26 |
| 3.4 | The Running Red Light Infraction | 27 |
| 3.5 | Collision Infraction | 28 |
| 3.6 | Bug in the CARLA Leaderboard | 28 |
| 4 | Design & Implementation | 30 |
| 4.1 | Configuring for Windows | 30 |
| 4.2 | The NPC Agent | 31 |
| 4.3 | Detecting all infractions | 34 |

| | | |
|----------|-------------------------------------------------|-----------|
| 4.4 | The Driver log | 36 |
| 4.5 | Handling Yellow Light | 37 |
| 4.6 | Solving the Collision Infraction | 39 |
| 4.7 | Improving leaderboard evaluation time | 40 |
| 5 | Evaluation | 44 |
| 5.1 | Experiments | 44 |
| 5.2 | Leaderboard Metrics | 46 |
| 5.3 | ML-MAS metrics | 47 |
| 5.4 | Discussion | 50 |
| 6 | Conclusion | 52 |
| 6.1 | Summary | 52 |
| 6.2 | Limitations | 52 |
| 6.3 | Future Work | 54 |

Chapter 1

Introduction

Self-driving cars have the potential to transform the way we travel, offering a wide range of benefits and addressing various societal and technological challenges. By integrating cutting-edge technologies such as sensors, cameras, radar, lidar, and advanced Artificial Intelligence (AI) algorithms, self-driving cars can navigate and operate without human intervention. They have the potential to greatly enhance road safety by reducing human errors, which are responsible for the majority of traffic accidents. Additionally, they can optimise traffic flow, reduce congestion, and minimise energy consumption through efficient route planning and coordination. This project aims to demonstrate some of these features using a neuro-symbolic AI framework.

1.1 CARLA and Leaderboard

This project leverages the CARLA simulator ([Dosovitskiy et al., 2017](#)) and its leaderboard ([Leaderboard, 2019](#)) as a fundamental component for evaluating the performance of self-driving cars in the context of self-driving vehicles. CARLA (Car Learning to Act) is an open-source simulator specifically designed for autonomous driving research. It provides a realistic virtual environment that allows researchers and developers to test and validate their algorithms, models, and frameworks in a controlled and reproducible manner.

The CARLA simulator offers a wide range of features and functionalities that closely resemble real-world driving scenarios. It includes detailed urban environments, diverse weather conditions, and dynamic traffic patterns, enabling the evaluation of self-driving algorithms in complex and challenging conditions. By utilising CARLA, researchers can accelerate the development and testing of autonomous driving systems while minimising risks and costs associated with real-world experimentation.

In addition to the simulator itself, this project utilises CARLA's leaderboard, which serves as a standardised benchmarking platform for evaluating the performance of self-driving cars. The leaderboard provides a comprehensive evaluation framework that assesses various aspects of autonomous driving, including navigation, traffic rules compliance, efficiency, and safety. It enables researchers to compare their algorithms against state-of-the-art approaches and track their progress over time.

One of the primary metrics provided by the leaderboard is the driving score. The driving score is a composite measure that takes into account multiple factors, such as the distance covered, average speed, collision frequency, and the number of traffic rule violations. It provides an overall assessment of the agent's driving competence, balancing the need for efficiency with

safety and adherence to traffic regulations. A higher driving score indicates better performance and demonstrates the AI's ability to navigate the environment effectively.

The infraction penalty metrics provided by CARLA's leaderboard play a crucial role in assessing the safety and rule compliance of self-driving agents within the simulator. These metrics capture instances where the agent deviates from expected behaviour or violates traffic regulations, highlighting areas that require improvement in the agent's decision-making and control mechanisms.

The integration of CARLA's simulator and leaderboard in this project provides a robust evaluation framework for assessing the performance of self-driving vehicles. The leaderboard metrics collectively offer a comprehensive understanding of the AI's capabilities.

1.2 Problem Statement

Current solutions to autonomous driving predominantly rely on Machine Learning (ML) approaches. ML-based autonomous solutions, while promising, have certain drawbacks that need to be addressed (Chen and Krähenbühl, 2022). One significant disadvantage is the lack of interpretability, as ML models often operate as black boxes, making it challenging to understand the reasoning behind their decisions (Dworak et al., 2019). Limited generalisation is another concern, as ML models may struggle to adapt to unseen or novel scenarios, potentially compromising their performance in real-world driving conditions (Amodei et al., 2016). Adversarial attacks pose a threat, as ML models can be vulnerable to intentional manipulations of input data, leading to incorrect decisions (Szegedy et al., 2014). Moreover, ML-based approaches heavily rely on high-quality training data, which can be resource-intensive to obtain and may not cover all possible driving scenarios (at Work, 2023). Ethical challenges arise when determining how autonomous systems should make decisions in complex situations, raising concerns about prioritising safety and addressing societal values. Addressing these challenges necessitates research and the integration of complementary approaches, such as rule-based systems, to ensure transparency, robustness, and ethical decision-making in ML-based autonomous solutions.

A neuro-symbolic approach offers a promising solution to address the drawbacks of ML-based autonomous solutions. By integrating sub-symbolic ML techniques with symbolic reasoning, neuro-symbolic AI frameworks aim to combine the strengths of both paradigms, resulting in more robust and explainable autonomous systems (Shukairi and Cardoso, 2023). The incorporation of symbolic knowledge representations enables explicit reasoning, allowing stakeholders to understand and trust the decision-making process of the autonomous system (Wang et al., 2022b). Moreover, neuro-symbolic approaches tackle the challenge of limited generalisation by leveraging symbolic reasoning to handle novel or uncertain scenarios that ML models may struggle with (Shukairi and Cardoso, 2023). The explicit rules and domain knowledge integrated into the system enhance its ability to adapt and make rational decisions in diverse driving situations. Additionally, neuro-symbolic frameworks provide resilience against adversarial attacks by incorporating symbolic reasoning to enforce rule-based constraints and safety checks, making it more difficult for malicious actors to manipulate the system (Amodei et al., 2016).

Furthermore, neuro-symbolic approaches mitigate the data dependency issue by combining the learning capabilities of ML models with symbolic reasoning's ability to incorporate expert

knowledge and predefined rules. This hybrid approach leverages the power of data-driven learning while benefiting from the explicit constraints provided by symbolic reasoning. Additionally, neuro-symbolic frameworks address ethical challenges by incorporating ethical considerations into the symbolic reasoning component. This allows the system to make decisions aligned with predefined ethical principles, promoting fairness, addressing biases, and handling ethical dilemmas (Shukairi and Cardoso, 2023).

1.3 Goals

The goal of this project was to improve the performance of the Jason plan by incorporating diverse driving scenarios. This was done by evaluating and selecting appropriate plans for integration, and by modifying the bridge code to enable seamless communication between Python (CARLA API) and Java/Jason. Additionally, the project also aimed to improve the debugging and analysis of routes using logs.

- Incorporate diverse driving scenarios: This was done by identifying a variety of driving scenarios that would challenge the Jason plans, and then evaluating and selecting appropriate plans for integration. The scenarios included different traffic conditions and road collisions.
- Modify the bridge code: The bridge code is responsible for communicating between Python (CARLA API) and Java/Jason. The bridge code was modified to enable seamless communication between the two platforms, which allowed for the integration of the diverse driving scenarios.
- Improve debugging and analysis of routes using logs: The logs were used to debug the routes and to analyze the routes that were taken. The logs were also used to identify areas where the ML agent failed and where the Jason could be improved.

1.4 Outline

The remainder of this report is structured into five additional chapters. Chapter 2 provides a detailed explanation of various aspects required to understand the project. In Chapter 3, the methodology applied at the start of the project is detailed, including initial test scores, and a range of methods employed to detect and enhance ML-MAS. Chapter 4 explores the innovative solutions devised during the project's progression, highlighting different approaches taken to tackle the challenges faced in attempting to achieve improved driving performance. In chapter 5, the results of the final run are presented and compared against past runs, assessing the effectiveness of the integrated framework and the impact of the new Jason plans. Finally, the report concludes in Chapter 6 with a summary of the findings, highlighting key achievements, limitations, and potential future directions for further optimisation and refinement of the neuro-symbolic AI framework.

Chapter 2

Background

This chapter provides a detailed explanation of various aspects required to understand the project. It includes an overview of the ML-MAS framework, the CARLA simulator, and the Jason language.

2.1 CARLA Simulator

CARLA (Car Learning to Act) simulator (Dosovitskiy et al., 2017) is an open-source autonomous driving simulation platform developed by the Computer Vision Center (CVC) and the Barcelona Super-computing Center (BSC). It serves as a powerful tool for researchers and developers in the field of autonomous driving to test and validate their algorithms in a realistic virtual environment. CARLA offers a high-fidelity urban driving experience, complete with dynamic traffic scenarios, weather conditions, and complex road networks. Figure 2.1 shows a still image of the CARLA simulation environment.



Figure 2.1: CARLA Simulator (<https://carla.org>, 04-Aug-2023)

2.2 CARLA Agents and Sensors

CARLA allows users to create and control various actors. “Actors” refer to dynamic entities that populate the virtual environment, contributing to the realism and complexity of scenarios. These actors can represent various objects that you would encounter in real-world traffic, such as vehicles, pedestrians, traffic signs, traffic lights, and more. CARLA provides a rich set of predefined actor types that can be instantiated within the simulation.

In CARLA, the ego-vehicle is the vehicle/actor that is being controlled by the user or by an algorithm. The ego-vehicle is an important concept in CARLA because it is the focus of the simulation. All of the sensors and actuators that are attached to the ego-vehicle are used to collect

data about the environment, and the decisions that are made about how to control the ego-vehicle are based on this data. The actors that CARLA provides are:

- **Vehicle:** Actors representing various types of vehicles, from passenger cars to trucks and buses. These vehicles can be controlled manually or autonomously to simulate traffic interactions.
- **Pedestrians:** Pedestrian actors represent people walking on foot. They can move around the environment, interact with vehicles, and follow predefined paths or behaviours.
- **Traffic Signs and Traffic Lights:** These actors mimic the behaviour of real-world traffic control elements. They affect the behaviour of vehicles by regulating speed limits, right-of-way rules, and traffic flow.
- **Sensors:** Although not physical objects, sensors are essential actors in CARLA. They can be attached to vehicles and other actors to simulate various sensors like cameras, LiDARs, and radars. These sensors capture data that is vital for training and testing autonomous driving algorithms.
- **Miscellaneous:** CARLA also provides other static actors like barricades, barriers, and even wind turbines. These help in creating diverse and realistic environments for testing various scenarios.

These actors interact within the simulation environment, providing a rich testing ground for autonomous driving algorithms. Sensors like cameras, LiDARs, radars, and GPS modules can be attached to the vehicles, enabling the collection of sensor data that closely resembles real-world driving conditions. Listing 2.1 shows an example code for a spawning a vehicle actor.

```
1 client = carla.Client('localhost', 2000)
2 client.set_timeout(2.0)
3
4 world = client.get_world()
5
6 blueprint_library = world.get_blueprint_library()
7 bp = random.choice(blueprint_library.filter('vehicle'))
8
9 transform = random.choice(world.get_map().get_spawn_points
    ())
10 vehicle = world.spawn_actor(bp, transform)
11 print('created %s' % vehicle.type_id)
```

Listing 2.1: Python code for spawning a vehicle actor.

2.3 Python API of CARLA

CARLA provides a Python API that allows developers to control and interact with the simulation environment. This API empowers users to manipulate vehicles, pedestrians, sensors, and other

simulation parameters. It also facilitates the creation of custom scenarios, the testing of algorithms, and the integration of machine learning models for autonomous decision-making. Listing 2.2 shows an example code that sets up the client to communicate with the CARLA server running in your local machine, and Listing 2.3 shows how a world can be generated or loaded to the connected CARLA.

```
1 client = carla.Client('localhost', 2000)
2 client.set_timeout(2.0)
```

Listing 2.2: Python code to connect CARLA

```
1 world = client.load_world('Town07')
```

Listing 2.3: Python code to load a world

Once the world is generated, we can use that object to spawn actors, change the weather, etc. The default CARLA installation comes with pre-coded examples for different API functionalities. It can be found inside PythonAPI and then examples. Listing 2.4 shows how to list and spawn all vehicles in the generated world using python API.

```
1 spectator = world.get_spectator()
2 vehicle_blueprints = world.get_blueprint_library().filter(
    'vehicle')
3
4 location = random.choice(world.get_map().get_spawn_points
    ()).location
5
6 for blueprint in vehicle_blueprints:
7     transform = carla.Transform(location, carla.Rotation(
        yaw=-45.0))
8     vehicle = world.spawn_actor(blueprint, transform)
9
10    try:
11
12        print(vehicle.type_id)
13
14        angle = 0
15        while angle < 356:
16            timestamp = world.wait_for_tick().timestamp
17            angle += timestamp.delta_seconds * 60.0
18            spectator.set_transform(get_transform(vehicle.
                get_location(), angle - 90))
19
20    finally:
21        vehicle.destroy()
```

Listing 2.4: Python code to get the list of vehicles in a world

2.4 CARLA Leaderboard

CARLA Leaderboard (Leaderboard, 2019) is an extension of the CARLA Simulator that serves as a benchmarking platform for evaluating the performance of different autonomous driving algorithms. It features predefined scenarios that challenge various aspects of autonomous driving, such as urban driving, lane following, and interaction with pedestrians and traffic. Developers can submit their algorithms to the leaderboard to compare their solutions against others and track their progress over time. Figure 2.2 shows a still image of the CARLA Leaderboard challenge.

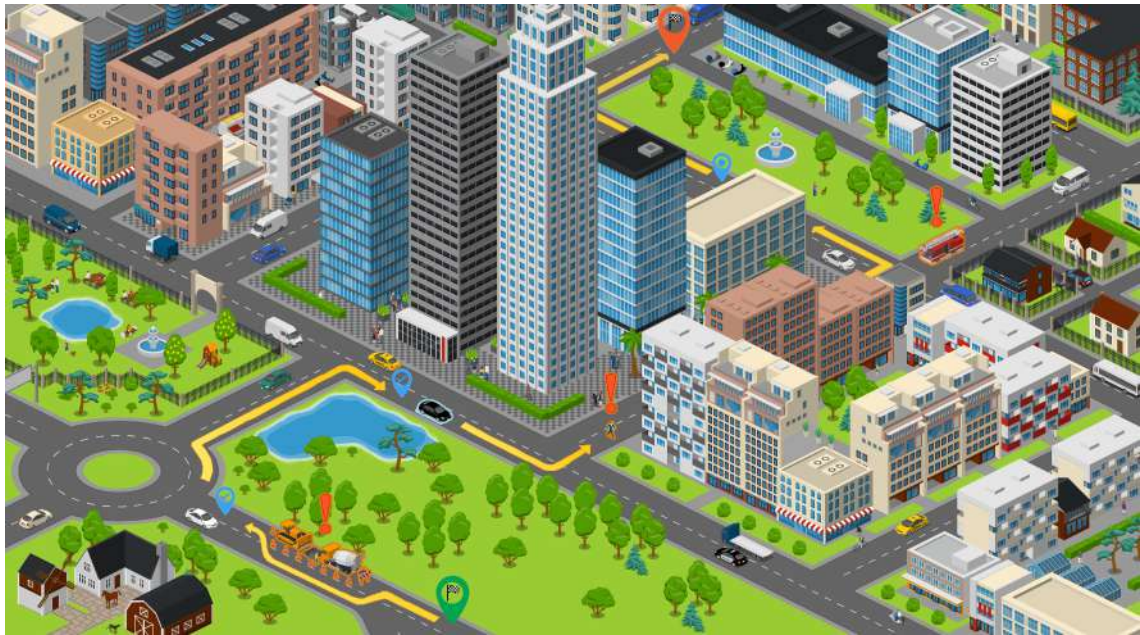


Figure 2.2: The CARLA Leaderboard challenge <https://leaderboard.carla.org>, 04-Aug-2023)

2.4.1 Setting up CARLA Leaderboard 1.0

Setting up the CARLA Leaderboard 1.0 involves creating an evaluation pipeline for autonomous driving algorithms, which are then benchmarked against predefined scenarios in the CARLA simulator. This process allows developers to assess their algorithms' performance and compare them with other solutions. CARLA Leaderboard 1.0 is a foundational version of this benchmarking system, and there is also a newer version, Leaderboard 2.0, which offers new maps, additional features and improvements.

The following steps can be taken to set up CARLA Leaderboard:

- **Installation:** Begin by installing the CARLA simulator and the necessary dependencies, following the official instructions. Make sure to have the required version of CARLA that is compatible with Leaderboard 1.0.
- **Clone Leaderboard Repository:** Clone the CARLA Leaderboard repository from its GitHub repository. This repository contains scripts, tools, and predefined scenarios for running

evaluations.

- **Scenario Setup:** Define the evaluation scenarios you want to test your algorithms against. These scenarios include different challenges like urban driving, lane following, pedestrian interactions, etc. You can customise these scenarios if needed.
- **Algorithm Integration:** Integrate your autonomous driving algorithm with the CARLA Leaderboard framework. This is where we will use our ML-MAS framework.
- **Configuration:** Configure the evaluation parameters, such as the number of simulation runs, weather conditions, and other environmental factors, in the configuration files provided by the Leaderboard.
- **Running Evaluations:** Run the evaluation pipeline using the provided scripts. The pipeline will launch the CARLA simulator, load the scenarios, and execute your algorithm within those scenarios. The performance metrics of your algorithm will be recorded.
- **Benchmarking:** After the evaluations are complete, the CARLA Leaderboard will compute various metrics based on the algorithm's performance in different scenarios. These metrics may include collision rate, infractions, distance travelled, completion time, and more.
- **Results:** The Leaderboard will provide a report detailing your algorithm's performance across the benchmarked scenarios. This allows you to identify strengths, weaknesses, and areas for improvement.

Hilal Al Shukairi, the previous MSc AI student that worked on a similar project, has created scripts to automate most of the above processes.

2.5 Leaderboard Metrics

The CARLA Leaderboard assesses submitted algorithms based on a set of metrics that reflect the algorithm's effectiveness in handling various driving scenarios. These metrics include collision rate, infractions, distance travelled, time taken, and more. By analysing these metrics, developers can gain insights into how well their algorithms perform under different conditions and identify areas for improvement. This standardised evaluation framework helps advance the development of safer and more capable autonomous driving systems. Figure 2.3 shows an example of the list of infractions the Leaderboard metrics include.

Each of these infractions has a penalty coefficient that will be applied every time it happens. Some of the major infractions and their penalties are:

- Collisions with pedestrians — 0.50.
- Collisions with other vehicles — 0.60.
- Collisions with static elements — 0.65.
- Running a red light — 0.70.
- Running a stop sign — 0.80.
- Scenario timeout — 0.70.


```

"infractions": {
  "Collisions with layout": [],
  "Collisions with pedestrians": [],
  "Collisions with vehicles": [],
  "Red lights infractions": [
    "Agent ran a red light 293 at (x=341.25, y=299.1, z=0.104)"
  ],
  "Stop sign infractions": [],
  "Off-road infractions": [],
  "Min speed infractions": [],
  "Yield to emergency vehicle infractions": [],
  "Scenario timeouts": [],
  "Route deviations": [
    "Agent deviated from the route at (x=95.92, y=165.673, z=0.138)"
  ],
  "Agent blocked": [],
  "Route timeouts": []
}

```

Figure 2.3: Example of a route where the agent both run a red light and deviated from the route.
(<https://leaderboard.carla.org>, 04-Aug-2023)

2.6 The ML-MAS Framework

The idea of combining deliberative reasoning with learning has been studied extensively in human decision-making and cognitive science. Daniel Kahneman, a psychologist, described this in his book "Thinking, Fast and Slow." (Kahneman, 2011) He argued that human decisions are made through a collaboration of two systems: System 1 and System 2.

System 1 is used for fast, intuitive, and unconscious decisions. System 2 is used for more complex situations that require logical and rational thinking. Kahneman estimates that about 95% of our thinking is done using System 1.

When applied to ML-MAS, System 1 can be seen as the ML component, which makes unconscious decisions based on experience. System 2 can be seen as the symbolic AI component, which makes cognitive decisions.

The ML-MAS framework (Shukairi and Cardoso, 2023) involves the combined decision-making power of pre-trained ML models and BDI(Belief-Desire-Intention) agents(Rao and Georgeff, 2000). BDI agents are designed to be rational agents that are able to make decisions that are consistent with their beliefs, desires, and intentions. ML-MAS has been designed for the domain of self-driving vehicles. It combines the strengths of both paradigms, resulting in more robust and explainable autonomous systems. Figure 2.4 shows an overview of ML-MAS framework. In this section, we explain what has been done in the past using this framework, which we extend in later chapters of this report.

Evaluation and testing requires a realistic environment as it allows the self-driving system to encounter a wide range of real-world challenges and scenarios. This includes dealing with complex road layouts, unpredictable traffic patterns, adverse weather conditions, and unexpected obstacles. Testing in such an environment ensures that the system can handle the complexities and uncertainties it will face in real-world driving scenarios. The CARLA simulation environment (Dosovitskiy et al., 2017) provides a realistic 3D simulation of urban driving with a wide variety of sensors including RGB Cameras and LiDAR. Moreover, it has Python API to interact with the simulation, making it easy to integrate with ML-MAS. Combining with the environment, CARLA's leaderboard feature can be used to create various routes and scenarios to test

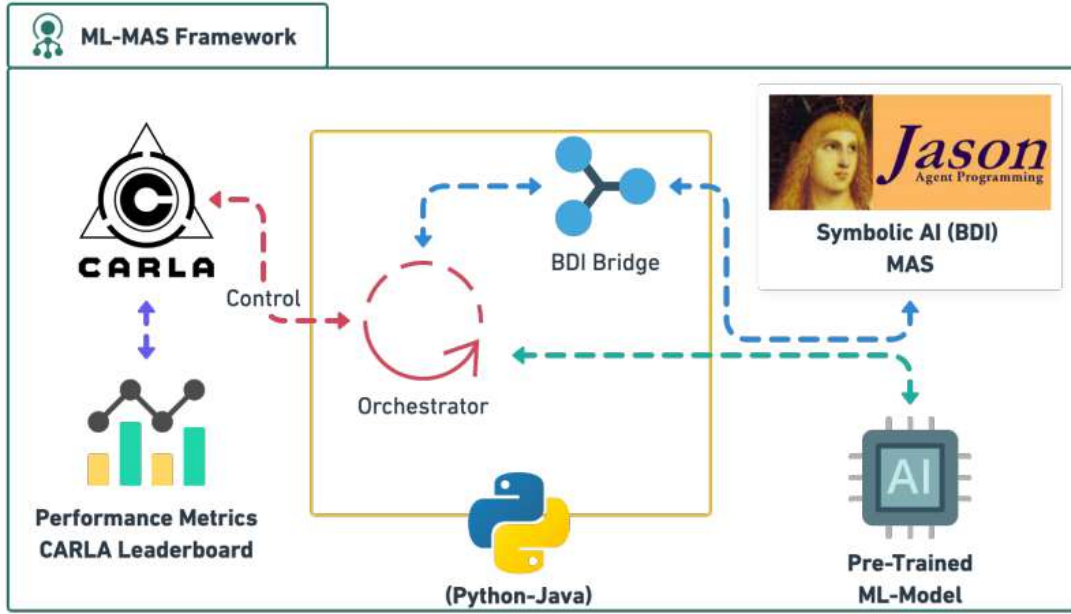


Figure 2.4: ML-MAS framework overview (Obtained from (Shukairi and Cardoso, 2023))

our approach. It also offers dedicated route evaluation metrics such as the driving scores and the infraction penalties.

2.6.1 Learning from All Vehicles - LAV

The Learning from All Vehicles (LAV) model (Chen and Krähenbühl, 2022) is a deep learning model that is used to train autonomous vehicles. It was developed by Dian Chen and Philipp Krähenbühl at the University of Texas at Austin. Figure 2.5 shows how the LAV model works.

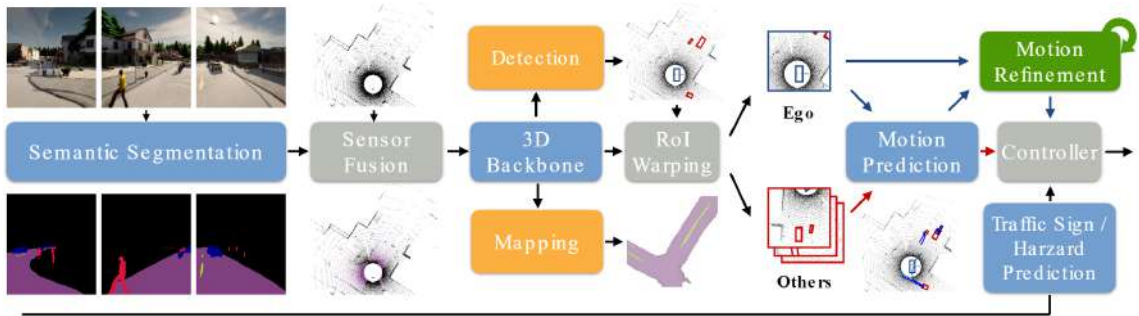


Figure 2.5: LAV model (<https://dotchen.github.io/LAV/>, 04-Aug-2023)

The LAV model is unique in that it can learn from the behaviour of all vehicles in a scene, not just the ego-vehicle. This is done by first learning a viewpoint-invariant representation of the scene using auxiliary supervision from 3D detection and segmentation tasks. This representation is then used to predict the future behaviour of all vehicles in the scene.

The LAV model has been shown to be very effective in training autonomous vehicles. It has been shown to outperform all prior methods on the public CARLA Leaderboard (Chen and Krähenbühl, 2022). The LAV model has also been shown to be able to handle more complex driving scenarios than previous models.

2.6.2 AgentSpeak Programming - Jason

AgentSpeak (Bordini et al., 2007), (Cardoso and Ferrando, 2021) is an agent-oriented programming language. It is based on logic programming and the Belief-Desire-Intention (BDI), a software model architecture for autonomous agents.

Jason (Bordini et al., 2007) is an open-source interpreter for an extended version of AgentSpeak. It enables users to build complex multi-agent systems that are capable of operating in environments previously considered too unpredictable for computers to handle.

Jason is based on the BDI agent architecture, which provides a formal framework for representing the knowledge, goals, and plans of agents. This makes it well-suited for applications where agents need to be able to reason about their environment and make rational decisions.

```

1  +!drive_to_intersection(Path,I) : at(Path) &
    get_traffic_light_status(I, "Red")
2      <- !navigate_to(I);
3      brake(1.0).

```

Listing 2.5: A sample Jason code

Listing 2.5 shows an example for a Jason plan for an agent in a scenario involving driving to an intersection. Lets break down the basics of the syntax and the different components of the plan.

- **Triggering Condition:** The triggering condition is denoted by `+!drive_to_intersection(Path, I) : at(Path) & get_traffic_light_status(I, "Red")`. This condition specifies when the plan should be triggered. In this case, the plan will be triggered when the following conditions are met:
 - The agent is at the Path (denoted by `at(Path)`).
 - The traffic light status in intersection I is Red.
- **Body of the Plan:** The body of the plan, is the set of actions that the agent will perform when the triggering conditions are met. (Anything after '`<-`'). In this case, the `!navigate_to(I)` plan will be invoked and `brake` will have a 1.0 value.

2.6.3 Setting up the ML-MAS Framework with CARLA and LAV model

Hilal Al Shukairi created scripts to automate the process of installing and evaluating the ML-MAS framework with CARLA. He used LAV model as the ML part and Jason as the BDI agent. Each evaluation took a huge amount of time to run. Therefore the “longest6” benchmark containing 36 routes were selected for the evaluation instead of the full 70+ routes in the official Leaderboard submission. Listing 2.6 shows Hilal Al Shukairi’s script code to obtain the weights of the LAV model.

```

1  echo "1. Downloading the LAV model weights"
2  mkdir tmp
3  cd tmp
4  wget https://github.com/alshukairi/public/raw/main/
    LAV_Weight/LAV_weights.zip.000

```

```

5 wget https://github.com/alshukairi/public/raw/main/
   LAV_Weight/LAV_weights.zip.001
6 wget https://github.com/alshukairi/public/raw/main/
   LAV_Weight/LAV_weights.zip.002
7 cat LAV_weights.zip.* > ../LAV_weights.zip

```

Listing 2.6: Hilals script to download the LAV model weights

2.6.4 The BDI Bridge

Hilal Al Shukairi developed a BDI bridge that allows seamless communication between Jason and CARLA. For each step, the information perceived from the environment is converted to Jason beliefs. These beliefs trigger Jason plans and return an action which is converted back to a CARLA vehicle control object.

The architecture of the BDI bridge is shown in Figure 2.6. There are mainly 3 threads running to make sure the communication is efficient and accurate: a thread to send the messages, a thread to receive the messages, and a thread to handle the messages. There are buffers set to make sure that the threads communicate independently and no thread conflicts happen throughout the evaluation.

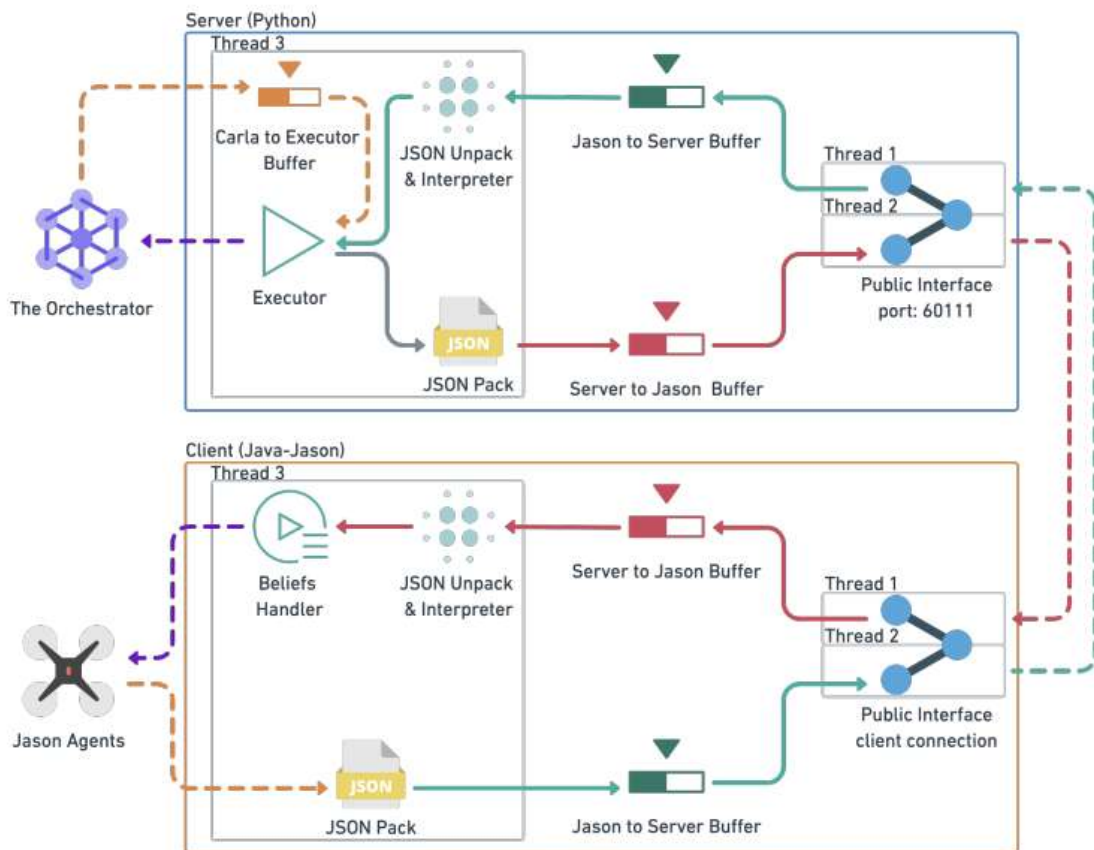


Figure 2.6: Architecture of BDI bridge (Obtained from (Shukairi and Cardoso, 2023).)

The communication in the bridge follows a JSON format. Both the client and server utilise a JSON unpacker and interpreter to identify the type of each message and determine the appropriate

actions to take for each message. Listing 2.7 shows a control (action) message in JSON format as an example of a message being sent from the agent to the orchestrator.

```

1 {
2     "type": {"id":1, "name": "control"},
3     "data": {
4         "throttle": 0.0,
5         "steer": 0.0,
6         "brake": 0.0,
7         "hard_brake": false
8     }
9 }
```

Listing 2.7: Example communication message in the bridge

The data attribute contains the values that represent a CARLA VehicleControl object. It contains the following:

- Throttle (0.0 to 1.0) : The throttle of the car. A value of 0.0 means that the car is not accelerating. A positive value means that the car is accelerating.
- Steer (-1.0 to 1.0): The steering angle of the car. A value of 0.0 means that the car is driving straight. A positive value means that the car is turning left, and a negative value means that the car is turning right.
- Brake (0.0 to 1.0) : The brake of the car. A value of 0.0 means that the car is not braking. A positive value means that the car is braking.
- hand_brake (True/False): The handbrake of the car. A value of True means that the handbrake is engaged, and a value of False means that the handbrake is disengaged.

2.6.5 The Orchestrator

The orchestrator is the main controller for the ego-vehicle. It has a set of conditions based on what the ego-vehicle perceives from the environment that decides whether the driver should be Jason or the pre-trained ML model. A script file is set to easily configure the variables used by the orchestrator such as the ML model to be used, its configuration files, the routes to be evaluated, the port and root for CARLA, etc.

The orchestrator class inherits the AutonomousAgent class of CARLA which allows it to create and control agents and the sensors attached to it. The CARLA Python API has various pre-coded examples that can be used as a reference for understanding the AutonomousAgent object. The orchestrator sets the initial required variables and invokes the BDI bridge. Sensors are set to the spawned ego-vehicle and the Leaderboard runs through the given set of routes and scenarios returning the performance metrics that we need. Hilal Al Shukairi used the LAV pre-trained model (Chen and Krähenbühl, 2022) which gave him a driving score of ~ 30 in the longest6 map for leaderboard (Shukairi and Cardoso, 2023).

The code also creates a file called “jason_metrics.csv”, which contains information about the performance of the Jason plan in each route. This file includes information such as the total

number of frames, the number of front collisions, the number of back collisions, and more. This information is used for statistical analysis specific to the ML-MAS framework at the end of the benchmark.

2.6.6 The Utility Notebook

The Utility notebook is a one-stop shop for all analysis after running the evaluation. The notebook contains Python scripts to identify collisions and agent blocking in each route, analyse and compare infractions, and much more.

One such example that the Utility notebook does is shown in Listing 2.8.

```
1 TIME = 30
2 DISTANCE = 10
3 try:
4
5     client = carla.Client(HOST, PORT)
6     client.set_timeout(60.0)
7
8     print(client.show_recorder_actors_blocked(
9         targettedroute, TIME, DISTANCE))
10    print(client.show_recorder_collisions(targettedroute,
11        "v", "a"))
12 finally:
13     pass
```

Listing 2.8: Example code from Utility notebook

Listing 2.8 is a Python script that uses the CARLA simulator to show the actors blocked and collisions in a specific route for a specified time and distance.

The first line of code defines two constants: TIME and DISTANCE. These constants are used to specify the time and distance that the script will use to look for blocked actors and collisions.

The next two lines of code create a client object and set its timeout to 60 seconds. The client object is used to connect to the CARLA simulator.

The next two lines of code call the client object's show_recorder_actors_blocked() and show_recorder_collisions() methods. These methods are used to get a list of blocked actors and collisions in the specified route for the specified time and distance.

The targettedroute parameter is the name of the route that the script is analysing.

Other parameters for show_recorder_collisions() methods are:

- h = Hero
- v = Vehicle
- w = Walker
- t = Traffic light

- o = Other
- a = Any

The "v" and "a" parameters denotes "vehicle" and "any". All the list of vehicles that were involved in a collision are returned.

2.7 Related Work

The paper "Hybrid Verification Technique for Decision-Making of Self-Driving Vehicles"(Al-Nuaimi et al., 2021) proposes a technique that combines two important things. Planning before car starts driving and checking how well its doing while its driving. To achieve this, they used two well-known model checkers for Multi-Agent Systems: MCMAS(Lomuscio et al., 2009) and PRISM(Kwiatkowska et al., 2011).

MCMAS helps to check the plan the car makes before driving and helps to check the stability and consistency of the BDI logic. On the other hand, PRISM is like a helper that watches the car while the car is driving. It is a probabilistic model checker that verifies the success probability of the decisions taken by the vehicle. This is a simple BDI agent, driving in a parking-lot scenario, not like a self-driving environment such as CARLA.

In Hierarchical Adaptable and Transferable Networks (HATN) (Wang et al., 2022a) is an approach that mimics the driving behaviours in a human mind. It is said to be efficient in tricky scenarios like roundabouts and intersections. This is done by breaking complex scenarios into smaller tasks improving the learning efficiency. While this works well for the specific scenarios, it cannot be applied to more general driving (e.g., CARLA routes).

There are not many works that uses a neuro-symbolic AI (Besold et al., 2017), (Sarker et al., 2021), (Zheng et al., 2020). Some of the works that are done so far in combining ML with BDI agents include "A Neurosymbolic Hybrid Approach for Landmark Recognition and Robot Localization"(Coraggio and De Gregorio, 2007), which uses the landmarks from a map to determine the location of an agent in that map using a weightless neural network combined with a BDI agent. The image sensor analyses and feeds the pre-processed data to the agent for calculating the agent location in the map.

Another similar work is "An Intelligent Active Video Surveillance System Based on the Integration of Virtual Neural Sensors and BDI Agents"(GREGORIO, 2008). This involves a more traditional BDI agent. The BDI agent is combined with ANN's to conduct video surveillance in two domains, railway tunnels and outdoor storage areas.

CARLA Leaderboard Sensors Track provides a wide range of solutions, methods and the scores they achieved over the years. Team Interfuser(Leaderboard, 2019) has created an unique approach to the problem. Their first submission in Leaderboard was a safety-enhanced autonomous driving framework, named Interpretable Sensor Fusion Transformer(InterFuser) (Shao et al., 2022), that fully process and fuse information from multi-modal multi-view sensors and achieves comprehensive scene understanding and adversarial event detection. Their second approach was a framework called ReasonNet(Shao et al., 2023) that extensively exploits both temporal and global information of the driving scene improving overall perception performance and benefit the detection of adverse events, especially the anticipation of potential danger from occluded objects.

Chapter 3

Methodology

In this chapter, I describe the methods and techniques that I used to develop and evaluate the extended ML-MAS framework for autonomous driving. I explain how I combined machine learning (ML) and multi-agent systems (MAS) to achieve more robust and intelligent decision-making in complex and dynamic driving scenarios. I also discuss different infractions that happened during the initial run. Additionally, I present the experimental design and data analysis methods that I employed to assess the performance of my framework and compare it with other approaches.

3.1 Initial Run

Upon taking over the project to extend ML-MAS, my initial task was to set up the ML-MAS existing code and re-evaluate the driving score achieved by the ML-MAS framework with longest6 benchmark. The results are given in Table 3.1.

Table 3.1: Comparison of CARLA Leaderboard metrics

| Metrics | ML-MAS | Just LAV | ML-MAS Extended |
|-----------------------------|--------|----------|-----------------|
| Avg. driving score | 52.472 | 38.967 | 48.673 |
| Avg. route completion | 96.379 | 77.173 | 89.071 |
| Avg. infraction penalty | 0.556 | 0.538 | 0.575 |
| Collisions with pedestrians | 0.034 | 0.049 | 0.04 |
| Collisions with vehicles | 0.044 | 0.242 | 0.058 |
| Collisions with layout | 0.042 | 0.028 | 0.072 |
| Red lights infractions | 0.212 | 0.201 | 0.199 |
| Stop sign infractions | 0.201 | 0.150 | 0.197 |
| Off-road infractions | 0.141 | 0.081 | 0.131 |
| Route deviations | 0.045 | 0.045 | 0.045 |
| Route timeouts | 0.000 | 0.017 | 0.007 |
| Agent blocked | 0.000 | 0.344 | 0.094 |

Comparing ML-MAS score of 52 with the extended ML-MAS initial score of 48 raised concerns regarding the performance discrepancy. However, after conducting a thorough analysis of multiple test runs, it became apparent that driving scores could vary dynamically within a certain range, but significant deviations from the main score were unlikely. Figure 3.1 shows two such runs having different results.

ML-MAS introduced an orchestrator as part of the project, which serves as a central component to coordinate the ML agent and the Jason agent in autonomous driving. The orchestrator acts

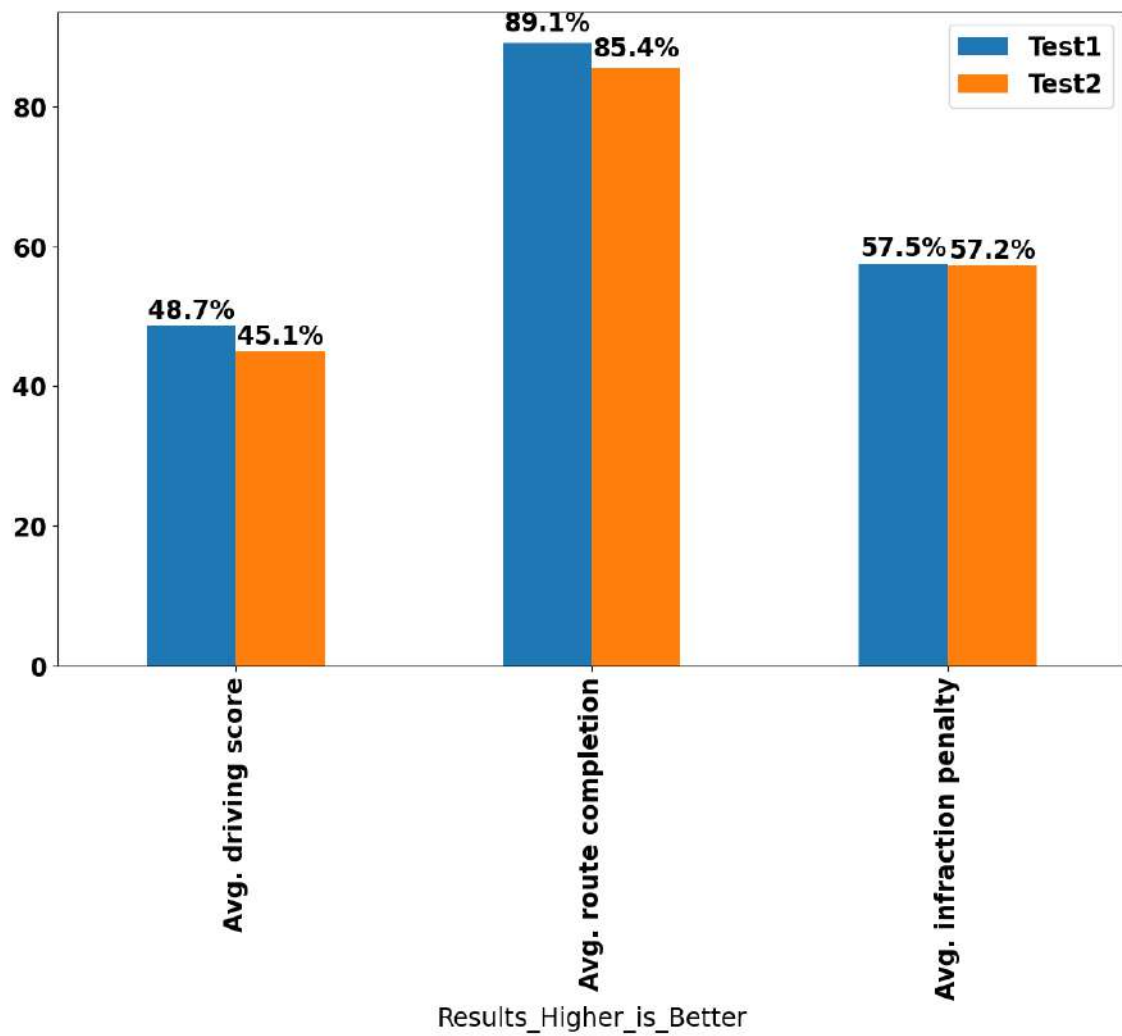


Figure 3.1: Two test runs on the ML-MAS Framework. Test1 is ML-MAS and Test2 is the extended ML-MAS

as a decision-making entity, monitoring the system's state and determining which agent, either the ML agent or the Jason agent, should take control of the vehicle based on a set of predefined conditions. The conditions that invoked Jason agent are shown in Figure 3.2.

In addition to the orchestrator, ML-MAS also had an utility code as part of the project, which facilitated the debugging and analysis of recorded routes, particularly focusing on collisions and agent blocking infractions.

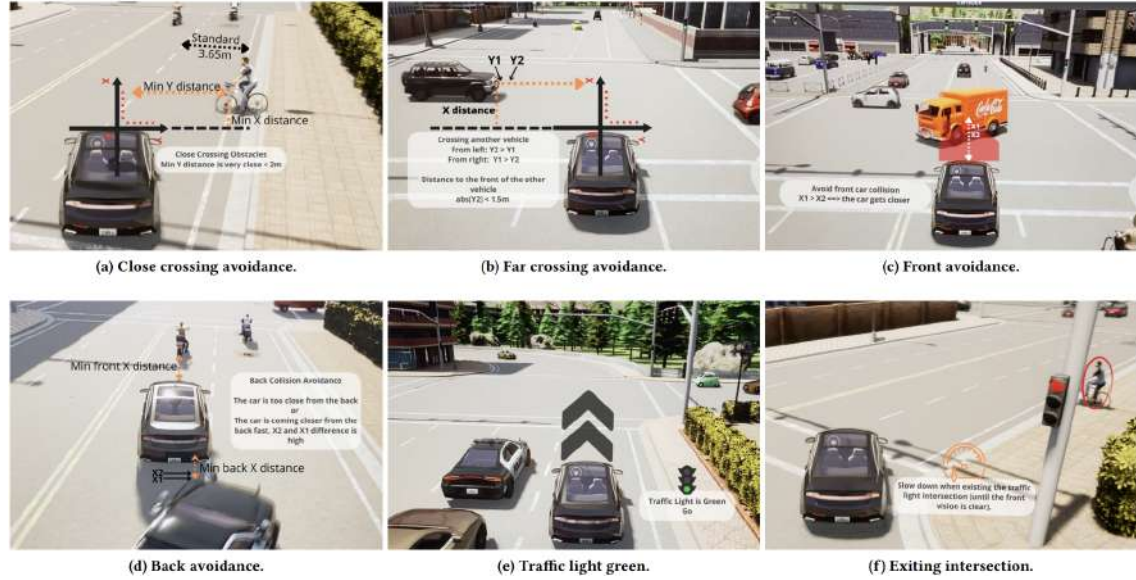


Figure 3.2: Visual representations of ML-MAS Jason plans

Figure 3.3 shows the infraction distribution across ML-MAS and extended ML-MAS results.

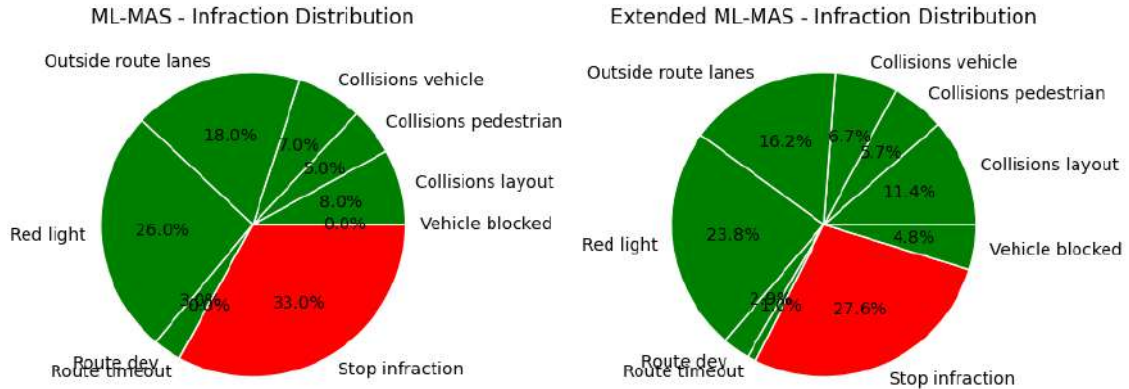


Figure 3.3: Infraction Distribution - Initial Run

The first analysis of the infractions revealed that the most common infractions were red light infractions, collisions, and stop sign infractions. These infractions occurred most frequently because the agent did not properly detect or respond to these situations.

For example, the agent sometimes ran red lights because it did not detect the light or because it did not recognise that the light was red. The agent also sometimes collided with other vehicles because it did not detect the other vehicles or because it did not take evasive action to avoid a collision.

In addition to these common infractions, the analysis also revealed some less common infractions. For example, the agent occasionally collided with pedestrians. This may have been because the agent did not detect the pedestrians or because it did not recognise them as pedestrians.

The agent also occasionally tried to overtake other vehicles when it was not safe to do so. This led to a collision with one vehicle on a route where there was tight traffic. The agent may have attempted to overtake the vehicle because it did not detect the other vehicles or because it did not recognise the risk of a collision.

The analysis also revealed that the agent's turning behaviour was not good. In some turns, the agent lost control of its lane and turned into another lane or even went to the middle divider. This may have been because the agent did not detect the other lanes or because it did not have a good understanding of how to turn safely.

On highways, the agent successfully understood lane changing and changed lane to avoid blocking. However, this lane changing did not include checking for coming vehicles in that particular lane. This led to collisions with other vehicles.

Finally, the analysis revealed that the agent occasionally got confused and went around loops, taking the routes that it had already taken. This caused route deviation infractions. This may have been because the agent did not have a good understanding of the map or because it was not able to plan a safe route.

The analysis of the infractions revealed that the agent had a number of areas where it could be improved. These improvements could be made by improving the agent's detection of other vehicles and pedestrians, improving its understanding of how to turn safely, and improving its ability to plan safe routes.

3.2 Extended Leaderboard Run-time

One of the biggest challenges I faced during this project was the extensive runtime required for Leaderboard evaluations. Each route took approximately 3000 to 4000 seconds to evaluate, which translates to about an hour. This was a significant problem, especially considering that there are 36 routes in the benchmark.

I investigated the issue further and found that the LAV agent alone was responsible for about 1000 seconds of the runtime. The code for the Jason-CARLA bridge also contributed to the inefficiency.

I knew from the outset that improving the Jason plans was important, as the LAV agent sometimes made poor decisions. However, eliminating the LAV agent would have made the evaluation faster. This would have required replacing the machine learning component with something that could operate seamlessly and make intelligent decisions in situations where the Jason plans were inactive.

I explored a few different options for replacing the LAV agent in order to speed up evaluation and testing of new Jason plans. I considered using another pre-trained model, but I decided against this because I did not want to focus on the ML weights before fixing the Jason plans. I also considered using a reinforcement learning algorithm, but I decided against this because I did not have enough time to train the algorithm.

While the elimination of the LAV agent seemed ideal as it makes the evaluation faster, this

necessitated the substitution of the machine learning component with an alternative that not only operated seamlessly but also functioned judiciously in scenarios where Jason plans were inactive. The solution found was to use the agent that comes with CARLA Leaderboard and extend it to work with the orchestrator and Jason (see Chapter 4.2 for details).

3.3 Analysing Infractions

The utility notebook is a valuable tool for analysing recordings from the Leaderboard. It can identify instances of collisions or infractions where vehicles block each other, and it even tells you the exact time these incidents occur during the game. This allows you to review what went wrong by watching those parts of the recordings.

However, there are a few limitations to the utility notebook. First, it does not detect all types of infractions. For example, it does not detect infractions like running red lights or driving off-road. Second, the accuracy of the game-time data provided for collision and agent blocked infractions is inconsistent. This can make it difficult to conduct effective route analysis.

I tried to modify the recorder class directly to address these limitations. However, this proved to be a difficult task. The recorder class is coded in C++ and heavily relies on functions linked to CARLA's DirectX functions.

Spotting other types of infractions requires me to manually watch the entire simulation. This became a constant time sink, especially when dealing with common issues like running red lights and collisions. I had to patiently watch the whole simulation, hoping to catch these specific problems. This process could be really tiresome and there was always a chance of missing important information. Listing 3.1 shows an example of recorder functions by CARLA.

```
1  try:
2
3      client = carla.Client(HOST, PORT)
4      client.set_timeout(60.0)
5
6      print(client.show_recorder_collisions(targettedroute,
7                                             "v", "a"))
8
9      print(client.show_recorder_actors_blocked(
10             targettedroute, TIME, DISTANCE))
11
12 finally:
13     pass
```

Listing 3.1: Carla recorder functions for collisions and agent block

Another hurdle I faced was the inconsistency in the accuracy of game-time data provided for collision and agent blocked infractions. I recognised the need for a solution that could rectify these discrepancies, thereby enabling me to conduct more effective route analysis. The new changes to improve this part of ML-MAS are described in Chapter 4.3.

Deeper Insights into Infractions

The recorder class from CARLA is a valuable tool for recording the routes in the Leaderboard. However, it does not provide all of the information that is needed to understand the root cause of infractions.

For example, the recorder class does not identify the agent that was in control when the infraction occurred. Additionally, it does not provide details about the vehicle's speed, the control state at the time of the infraction, or the conditions that led the driver to assume control. Obtaining such granular information was imperative for a thorough analysis of what transpired during these critical moments.

This lack of information made it difficult to understand what happened during an infraction. For example, if a vehicle runs a red light, it is not clear whether the infraction was caused by a mistake by the ML agent or by the Jason agent who took control of the vehicle.

Attempting to modify the CARLA recorder class turned out to be a dead-end and hence I could not add the extra details I needed to the existing recorder log files. An additional log file has to be created that logs all of these information and then can be used after evaluation for deeper analysis.

Figure 3.4 shows an example of “Agent collided against object with type=static.pole and id=0 at (x=363.825, y=324.231, z=0.085)”.



Figure 3.4: Collision to a static pole

3.4 The Running Red Light Infraction

During the evaluation of the autonomous vehicle, it was observed that the agent had difficulty detecting and responding to red traffic lights. The agent was able to detect red lights, but it often did not slow down or stop as expected. This was particularly noticeable in scenarios where the light was about to turn red.

To investigate this issue, a comparison was made with the behaviour of non-player agents

(NPCs) in the simulated environment. NPCs are computer-controlled agents with included way-points and information. In the case of traffic lights, NPCs slow down or stop when they see a yellow light.

The comparison with NPCs revealed that the ML agent was not specifically trained to recognise and react to yellow lights. This could be a limitation of the LAV model.

As a result, the agent did not have the necessary knowledge or understanding of how to respond to yellow lights. This led to instances where the agent crossed the intersection as the light was about to turn red.

One possible solution to this would be to let Jason handle the yellow light control, which is further discussed in Chapter 4.5.

3.5 Collision Infraction

The agent had a Jason plan for collision control, including cross-front collisions, far-front collisions, and back collisions. However, in route 0, the agent randomly collided with a pedestrian crossing the road. This indicates that sometimes the Jason plan is not working as intended. There are several possible explanations for this.

- The Jason plan may not be taking into account the specific circumstances of route 0. For example, the pedestrian may be crossing the road in a more unexpected manner than the Jason plan is expecting.
- The Jason plan may not be being executed correctly. This could be due to a bug in the code or a problem with the way the plan is being triggered.
- The Jason plan may not be comprehensive enough. It is possible that there are other types of collisions that the plan is not taking into account.

By debugging the existing Jason plans in such a way and making any necessary changes, we can improve the driving score of this route.

3.6 Bug in the CARLA Leaderboard

The CARLA Leaderboard had a bug where it occasionally stopped the agent at certain points even when there were no obstacles. This mostly happened when there was an upcoming red light which was not near and an object/vehicle passed by, the agent stopped. The NPC was also used to test it and it even reproduced the same result which made me realise that there is a bug in Leaderboard. This stopping continues for a long time.

This bug did not impact our final result, but it did increase the time for evaluating the routes. This is because the agent would stop at the red light, even though it was not yet close, and would wait for the light to turn green. This would cause the evaluation to take longer, as the agent would have to wait for the light to change before it could continue on its route.

One potential solution would be to modify the code for the Leaderboard to prevent the agent from stopping at red lights that are not close. This could be done by checking the distance to the red light before the agent decides to stop. If the distance is too great, the agent would not stop, and would continue on its route.

Another potential solution to this bug would be to add plans for red light control as a Jason plan. This would allow the agent to understand when it is safe to proceed and would prevent it from stopping unnecessarily.

Chapter 4

Design & Implementation

This chapter describes the designs and solutions I implemented to improve my driving score on the CARLA Leaderboard. I focused on solving the top infractions that were causing ML-MAS score to be low, such as colliding with vehicles and pedestrians, and running red lights. I also faced a number of challenges while solving these problems, such as determining the exact distance at which the ego-vehicle should stop for a red traffic light and debugging the Jason agent to ensure that the new plans were working correctly. However, I was able to solve the infractions and improve the driving score on the CARLA Leaderboard. This chapter provides a detailed overview of the process to achieve such results.

4.1 Configuring for Windows

Hilal Al Shukairi authored scripts for automation that were Unix-based, as Unix systems provided optimal performance for executing CARLA and the Leaderboard functionalities. However, due to my utilisation of a Windows-based operating system, I needed to implement certain modifications to ensure compatibility and seamless operation.

The initial phase involved the adaptation of the setup file created by Hilal Al Shukairi. This file facilitated the downloading of the LAV model weights along with CARLA version 0.9.10.1, (Dosovitskiy et al., 2017) inclusive of its required additional maps. This task was executed with relative simplicity. Nonetheless, a noteworthy alteration pertained to the choice of CARLA's Windows release, in contrast to the initially utilised Linux version.

After making those changes to adapt CARLA to my Windows-based system, I encountered some issues. The Windows version of CARLA had some files that were not functioning properly with the Leaderboard. Most of these problems were due to incorrect file paths. Once I fixed these file path issues, CARLA started running smoothly and without any problems.

Subsequently, I moved on to Hilal Al Shukairi's script for configuring and evaluating the Leaderboard. This phase proved to be quite intricate, given the multitude of global environmental variables that necessitated adjustments in alignment with the functionalities of the Windows operating system.

Incorporated within this evaluation script was an additional configuration file. This file addressed the configuration of the LAV model, establishment of the Jason-CARLA bridge, configuration of scenario routes, and specification of result and record storage pathways. Furthermore, I found it necessary to implement certain modifications to the CARLA Leaderboard itself, aligning it with the alterations made to the environment variables. This process demanded careful attention

due to its intricacy.

Upon completion of these modifications and adaptations, the setup was complete. Subsequently, the first test runs were conducted on the extensive “longest6” benchmark (Leaderboard, 2019). The meticulous adjustments made to the scripts, configurations, and environment variables, as well as the seamless integration of CARLA and the Leaderboard functionalities within a Windows environment, paved the way for the commencement of operational testing. This marked a significant milestone in the project’s progression, signifying the successful alignment of various components and processes.

4.2 The NPC Agent

Leaderboard auto-agents encompass example files containing code demonstrations for crafting agents utilising global way-points and other inherent functions of the Leaderboard. By inheriting the main AutonomousAgent class, a child class could capitalise on the extensive array of Leaderboard functions. The derived class should contain 3 main functions.

The setup function is the first function that is called when running a CARLA agent on the Leaderboard. It initialises all the variables that are required for a successful run, such as the track, the sensors, and the map. Listing 4.1 shows an example of the setup function defining Track.SENSORS.

```
1 def setup(self, path_to_conf_file):
2     """
3     Setup the agent parameters
4     """
5     self.track = Track.SENSORS
```

Listing 4.1: Example of a setup function in a child class of AutonomousAgent

The track variable specifies the type of track that the agent will be running on. There are two tracks available on the Leaderboard: the SENSORS track and the MAP track. The SENSORS track gives the agent access to a variety of sensors, such as cameras, LiDAR, and RADAR. The MAP track gives the agent access to a HD map of the environment. (Leaderboard, 2019)

The sensors variable specifies the sensors that the agent will be using. The default value for this variable is all sensors. However, you can also specify a subset of sensors if you want to improve the performance of your agent.

The map variable specifies the map that the agent will be using. The default value for this variable is none. However, you can specify a map if you want to improve the performance of your agent.

The setup function is a critical function, and it is important to ensure that it is properly initialised before running your agent on the Leaderboard.

Then comes the sensors function which defines what sensors is or are to be included in the ego-vehicle. (Leaderboard, 2019) There are various types of sensors that are available in the SENSORS Track. Some of them are shown in Figure 4.1.

Listing 4.2 shows an example on how to create a camera.RGB sensor in sensors function. The function returns a dictionary with keys:







| | | | | | |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |  |  |  |
| GNSS | IMU | LIDAR | RADAR | RGB camera | Speedometer |
| <i>sensor.other.gnss</i> | <i>sensor.other.imu</i> | <i>sensor.lidar.ray_cast</i> | <i>sensor.other.radar</i> | <i>sensor.camera.rgb</i> | <i>sensor.other.speedometer</i> |
| 0-1 units | 0-1 units | 0-1 units | 0-2 units | 0-4 units | 0-1 units |
| GPS sensor returning geo location data. | 6-axis Inertial Measurement Unit. | Velodyne 64 LIDAR. | Long-range RADAR (up to 100 meters). | Regular camera that captures images. | Pseudosensor that provides an approximation of your linear velocity. |

Figure 4.1: Sensors available for Track.SENSORS (Leaderboard, 2019)

- **type** : The sensor type that is required.
- **x**: The x-coordinate of the sensor in the ego-vehicle.
- **y**: The y-coordinate of the sensor in the ego-vehicle.
- **z**: The z-coordinate of the sensor in the ego-vehicle.
- **roll**: The roll angle of the sensor in the ego-vehicle.
- **pitch**: The pitch angle of the sensor in the ego-vehicle.
- **yaw**: The yaw angle of the sensor in the ego-vehicle.
- **width**: The width of the sensor's field of view.
- **height**: The height of the sensor's field of view.
- **fov**: The horizontal field of view of the sensor in degrees.
- **id**: The ID of the sensor.

```

1
2     def sensors(self):
3
4         sensors = [
5             {'type': 'sensor.camera.rgb', 'x': 0.7, 'y':
6               -0.4, 'z': 1.60, 'roll': 0.0, 'pitch': 0.0,
7               'yaw': 0.0,
8               'width': 300, 'height': 200, 'fov': 100, 'id'
9               : 'Left'}],
10
11         return sensors

```

Listing 4.2: Example of a sensors function in a child class of AutonomousAgent

Finally, the third important function is the `run_step` function. (Leaderboard, 2019) It is called every frame, and it is responsible for taking in the sensor data, making a decision, and sending control commands to the ego-vehicle. Listing 4.3 shows an example of a `run_step` function that takes `input_data` and returns a `carla.VehicleControl`.

```

1
2     def run_step(self, input_data, timestamp):
3         """
4         Execute one step of navigation.
5         """
6         control = carla.VehicleControl()
7         control.steer = 0.0
8         control.throttle = 0.0
9         control.brake = 0.0
10        control.hand_brake = False
11
12        return control

```

Listing 4.3: Example of a run-step function in a child class of AutonomousAgent

The return is a `carla.VehicleControl` object.

The `NpcAgent` class, a representative illustration provided by the Leaderboard, encapsulated a similar approach. The NPC agent was interesting due to its choice to abstain from using sensors, opting instead to exclusively rely on the information and way-points provided by the Leaderboard. Results for evaluating the NPC alone for the longest6 map in the leaderboard is shown in Table 4.1.

Table 4.1: CARLA Leaderboard metrics for NPC Agent

| Avg. driving score | Avg. route completion | Avg. infraction penalty | Col. with pedestrians | Col. with vehicles | Col. with layout | Red lights infractions | Stop sign infractions | Off-road infractions | Route deviations | Route time-outs | Agent blocked |
|--------------------|-----------------------|-------------------------|-----------------------|--------------------|------------------|------------------------|-----------------------|----------------------|------------------|-----------------|---------------|
| 14.947 | 52.106 | 0.384 | 0.068 | 0.570 | 0.000 | 1.399 | 0.108 | 0.000 | 0.697 | 0.000 | 0.255 |

The driving score achieved is really low due to the frequent inability of the agent to successfully complete the designated routes. This could be because of several reasons including the dynamic characteristics of the routes themselves, as well as vehicle-related dynamics such as speed considerations and lack of use of sensors.

However, the `NpcAgent` executed much faster than using pre-trained ML modes, therefore, the idea was to combine the `NpcAgent` with the Jason framework for testing new Jason plans. Notably, a single successful evaluation encompassing all 36 routes using the NPC agent consumed approximately 3 hours. Capitalising on the efficiency of this rapid assessment, the decision was made to integrate it with the Jason framework. This endeavour necessitated a minor adaptation to the orchestrator, a modification undertaken to facilitate the seamless amalgamation of these components. Thus, the `NpcAgent` would continue to operate as anticipated. However, when specific triggers within the Jason framework are activated, control would seamlessly transition to Jason,

ensuring a harmonious collaboration between the NPC agent and Jason's decision-making processes.

4.3 Detecting all infractions

The default CARLA recorder class does not have functions to identify infractions other than collisions and agent blocking. Therefore, a solution was required. First, I had to understand how the CARLA recorder replay function works. Through multiple attempts, I learned that the CARLA recorder replay function simply returns a string with information like the route name and route duration, as well as replaying that log to the locally connected CARLA.

The goal is to identify these infraction times from the coordinates provided by the Leaderboard metrics. Then, while running the route replay, use the coordinates to detect the time when a particular infraction took place. Because the replay function returns a string immediately after running, it does not wait until the route is fully played. This is critical because I now need to create a loop to keep my algorithm running until the route ends, and there is no way to determine when the route actually ends.

Infractions other than `outside_route_lanes` and `route_timeout` have coordinates that can be obtained from the Leaderboard metrics. A function called `get_infraction_coordinates` is created that takes the infractions from the given metrics file, converts them to CARLA Location objects, and returns a list of dictionaries with each route infraction and its CARLA Location object.

A loop is created and each infraction in the list is processed. In each iteration, the agent's location is obtained and the distance is calculated from the infraction point to the agent's location. When the agent is near the infraction location, both the current game time and the infraction name are saved. The final list is then saved to a JSON file (Listing 4.4), which contains all infractions and their times. This successfully converts infraction coordinates to game time.

```

1  {
2      "0": [
3          {
4              "time": 16.767668548971415,
5              "inf": "Agent ran a red light 8654 at (x
                      =332.04, y=316.35, z=0.103)"
6          },
7          {
8              "time": 80.618248347193,
9              "inf": "Agent collided against object with
                      type=walker.pedestrian.0007 and id=8824 at
                      (x=396.105, y=155.094, z=0.035)"
10         },
11         {
12             "time": 259.91655611619353,
13             "inf": "Agent ran a red light 8642 at (x
                      =102.72, y=52.85, z=0.103)"
14         }

```

```

15     ]
16 }

```

Listing 4.4: The final list with infraction time and details

The loop terminates when all infractions have been detected. This is done by popping the detected infraction from the main infraction list each time it is detected. When all infractions have been detected, the loop terminates and the next route replay log file is processed. This is efficient because if an infraction occurs at the beginning of a route, there is no point in running through the entire route.

In rare cases if the agent may not be able to calculate the distance, an additional condition is checked to see if the game time is greater than the game duration. The game duration is obtained from the Leaderboard metrics file (Listing 4.5).

```

1  {
2      "index": 0,
3      "infractions": {
4          "collisions_layout": [],
5          "collisions_pedestrian": [],
6          "collisions_vehicle": [],
7          "outside_route_lanes": [],
8          "red_light": [
9              "Agent ran a red light 19645 at (x
              =151.31, y=45.3, z=0.103)"
10         ],
11         "route_dev": [],
12         "route_timeout": [],
13         "stop_infraction": [],
14         "vehicle_blocked": []
15     },
16     "meta": {
17         "duration_game": 376.2500056065619,
18         "duration_system": 2872.230085849762,
19         "route_length": 1130.24134093902
20     },
21     "route_id": "RouteScenario_0",
22     "scores": {
23         "score_composed": 70.0,
24         "score_penalty": 0.7,
25         "score_route": 100.0
26     },
27     "status": "Completed"
28 }

```

Listing 4.5: An example of the route details from leaderboard metrics file

4.4 The Driver log

Identifying infractions is not enough to debug and understand what went wrong. More information is needed, such as who was controlling the vehicle at the time of the infraction and what control was returned by that agent. To log this information, the orchestrator file was modified.

The run-step function of the orchestrator class is the function that is called whenever a control is required. This function returns the vehicle control. A driver.csv file was created with three columns:

- Time: the specific game time that this step happened.
- Driver: either Jason or ML.
- Control: the control returned by a driver.

This was enough to have an initial idea of what was happening. But the time was not a good key to index a particular driver as it may not always be accurate. To solve this, a column 'frame' was also added. This made the indexing easier and accurate. More columns were also added to have an advanced level of understanding, such as:

- Condition: the triggering condition that caused to decide what Jason plan needs to be invoked.
- Speed: the speed of the vehicle for that step.

Figure 4.2 shows a sample of first 5 frames of a simulation.

| frame | time | driver | control | condition | speed |
|-------|----------|--------|--------------------------------------------------------------------------------------------------------------------------------------|-------------------------|-------|
| 1 | 3.821664 | Jason | -1 | front_objects_detection | 0 |
| 1 | 3.821664 | ML | VehicleControl(throttle=0.000000, steer=0.000000, brake=0.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0) | | 0 |
| 2 | 3.871664 | Jason | -1 | front_objects_detection | 0 |
| 2 | 3.871664 | ML | VehicleControl(throttle=0.000000, steer=-0.010668, brake=1.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0) | | 0 |
| 3 | 3.921664 | Jason | -1 | front_objects_detection | 0 |
| 3 | 3.921664 | ML | VehicleControl(throttle=0.000000, steer=-0.016246, brake=1.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0) | | 0 |
| 4 | 3.971664 | Jason | -1 | front_objects_detection | 0 |
| 4 | 3.971664 | ML | VehicleControl(throttle=0.000000, steer=-0.014002, brake=1.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0) | | 0 |
| 5 | 4.021664 | Jason | -1 | front_objects_detection | 0 |
| 5 | 4.021664 | ML | VehicleControl(throttle=0.000000, steer=-0.011449, brake=1.000000, hand_brake=False, reverse=False, manual_gear_shift=False, gear=0) | | 0 |

Figure 4.2: Driver data for the first 5 frames

In Figure 4.2, you can see the controls for Jason driver is -1. The leaderboard detected a front upcoming obstacle (front_objects_detection), which is set as the triggering condition for Jason. This means that Jason will be invoked when this condition is met. Jason is responsible for returning a control that the Orchestrator can receive. However, not all conditions need to be handled. For example, the leaderboard detects even upcoming vehicles in opposite lanes as obstacles, or poles and railings as obstacles. There are other factors, such as the distance between the ego-vehicle and the obstacle, or the speed with which the ego-vehicle is going towards the obstacle, that determine whether Jason should take an action or not. If those extra conditions are not met, then Jason simply returns no_action as shown in Listing 4.6.

```

1 //send back no action, to allow the ML model control.
2 +!start(F): info(F,Sp) & Sp <= 0 & stopCount(N)
3         <- no_action;
4         +-block(0,60);

```

```

5      .print("=== [NO ACTION] ===: ", F);
6      -+stopCount(N+1) .
7
8  +!start(F) <- no_action;
9      -+block(0,60);
10     -+stopCount(0) .

```

Listing 4.6: Jason plan for no_action

The no_action plan is a plan that is invoked when there are no Jason plans that meet their invoking conditions. This no_action plan is transformed to a -1 value in the bridge. This -1 value is then passed to the Orchestrator, which tells the Orchestrator to continue with the ML control. This is also the reason why there are 2 controls for one frame.

For example, in Figure 4.3, the leaderboard detected a front upcoming obstacle (front_object_detection). This condition is met, so Jason is invoked. However, Jason determines that the obstacle is far enough away or that the ego-vehicle is not going fast enough to pose a threat, so Jason returns no_action. The bridge then transforms this to a -1 value, which is passed to the Orchestrator. The Orchestrator then continues with the ML control, which in this case is to accelerate.

| frame | time | driver | control | condition | speed |
|-------|----------|--------|-------------------------------------------------------------------------------------------------------------------------------|-------------------------|-------|
| 1 | 3.821664 | Jason | -1 | front_objects_detection | 0 |
| 1 | 3.821664 | ML | VehicleControl(throttle=1.000000,steer=0.000000,brake=0.000000,hand_brake=False,reverse=False>manual_gear_shift=False,gear=0) | | 0 |

Figure 4.3: Driver data for the first 5 frames

One more thing that can be observed from Figure 4.2 is that the ML driver has no value for the condition column. This is because it is not possible to determine what caused the ML to return a control. The input_data that is received by the run_step function contains information from the sensors, such as RGB values or LiDAR values. It is not possible to break down this data to determine what caused the ML to take a particular control. Debugging this to understand what caused ML to take that control would be like changing the pre-trained ML agent functions. Therefore, I opted to not do it and leave the condition for ML driver blank for now.

4.5 Handling Yellow Light

The red light infraction was the most common and had a huge penalty when compared to other infractions. To address this problem, several solutions were attempted.

Initially, a new plan was devised within the Jason agent to identify and apply braking when a yellow light was detected. However, this approach proved ineffective as the agent only detected the yellow light for a single step, resulting in a momentary application of brakes followed by continued movement.

An alternative solution involved implementing a small reverse action instead of a sudden stop upon detecting a yellow light. Although this approach worked in avoiding abrupt stops, it introduced the potential risk of collisions with NPCs located behind the agent in different scenarios.

The breakthrough came when investigating the “Approaching” stage of the traffic light detection process.

The traffic light input to the Jason agent by the orchestrator contains the following parameters:

```
1 traffic_light(F,A,R,X,Y,D,InBox)
```

Listing 4.7: traffic light input to the Jason

- Frame(F): The current frame of CARLA. With this information, we can access other beliefs like speed that the ego-vehicle had or the ML control applied at that frame.
- Type (A,L): This indicates whether the traffic light is “Approaching” or “Left”. Approaching traffic type is invoked when the distance between the traffic light and the ego-vehicle is greater than 6 game distance. Left traffic type is invoked when the distance between the traffic light and the ego-vehicle is less than 6 game distance.
- State (R,Y,G): The state of the traffic light. Whether it is Red (R), Yellow (Y) or Green (G).
- X: The X coordinate of the traffic light.
- Y: The Y coordinate of the traffic light.
- InBox(0,1) : This returns 1 if the traffic light is inbox (in an intersection) and 0 if it is not inbox.

Through careful debugging and analysis of Listing 4.7, it was discovered that the “A” status, which indicated the agent’s proximity to the traffic light, remained active until the light was approximately 8 game distances away. However, further examination of recorded data revealed that the optimal distance for detecting yellow lights was approximately 6 game distances. Consequently, the code was modified accordingly, resulting in improved detection of the traffic light types. Now, a Jason plan has to be created to return the necessary control.

```
1 //A plan to stop the car when the traffic light is yellow.
2 +!start(F): traffic_light(F,"A","Y",_,_,_,_) & ml_control(F,_,St,_,_,_)
3           & info(F,S)
4           <- control(0, 0.0, St, 1.0, false, false, 5); // Stop
5           .print("===Traffic Light is Yellow: Stop ===: ", F).
```

Listing 4.8: The yellow light plan from Jason

As shown in Listing 4.8, there are 3 conditions to be met.

- There exist a traffic light belief with a certain frame F, the type is "Approaching" and the traffic light state is "Yellow".
- There exist a machine learning control belief for the given frame F with a Steer value St.
- There exist an info belief for the given frame F with a speed value S.

When these conditions are met, Jason returns a brake control to stop the vehicle from crossing a yellow light.

By analysing the behaviour, comparing it with NPCs, and repeatedly testing different solutions, including modifying the distance thresholds for detection, the agent’s performance in identifying and appropriately responding to yellow lights was significantly improved.

4.6 Solving the Collision Infraction

The ego-vehicle was having collision infractions, and upon further investigation via the use of the driver log that I developed, I found that the Leaderboard was detecting close objects, but the Jason plan was not being invoked because the conditions for the plan were not being met. With the help of the driver log and some debug print statements, I was able to identify which Jason plan was responsible for the scenario. It had 3 conditions:

- $\text{MinY} < 2.0$: MinY is the output from LiDAR showing the minimum y distance to the obstacle.
- $X < 4.5$: X is the position of the ego-vehicle.
- $\text{Sp} > 0.5$: Sp indicates Speed of the ego-vehicle.

Specifically, the condition $\text{MinY} < 2.0$ was failing, because the value of MinY was actually 2.066.

I considered two options to address this issue. One option was to increase the value of MinY to 2.1 or 2.2, but this would have caused the agent to drastically slow down multiple times, as it would now be detecting poles and steel barriers. The other option was to investigate why MinY was 2.066. I found that the values of MinY are calculated in the `ml_mas_agent0.py` file, and then sent to the Jason plan in the `jason_carla_bridge.py` file. I considered modifying the code to take the floor value of MinY, but this would have been a poor approach, as it would have modified the direct output from the LiDAR.

I then considered the possibility that the LiDAR was not working properly for the ML agent, because of the difference in speed between the ML agent and the NPC agent. I found that the NPC agent was going at around 4.783 just before the collision, while the ML agent was going at 7.087.

I could not edit the speed of the LAV agent. Therefore, I created a Jason plan that would reduce the speed of the vehicle if there was an object within 2.0 to 2.5 game distance by applying a brake of 0.4. This solution worked (Listing 4.9), and the collision infractions were significantly reduced.

```

1 // slow down if an object is detected between 2.5 and 2.0 lidar range
2 +!start(F): f(F,X,_,_, MinY) & MinY < 2.5 & MinY > 2.0 & X < 4.5
3           & info(F,Sp) & Sp > 5.5 & block(Nf, M) & Nf < M &
4           ml_control(F,_,St,_,_,_)
           <- control(6, 0.0, St, 0.4, false, false, 2). // slow down

```

Listing 4.9: plan to slow down if an object is detected between 2.5 and 2.0

Listing 4.9 shows the plan for slowing down when an obstacle is detected between 2.0 and 2.5 LiDAR range. The `f` belief contained the values for a forward collision. Using this belief, we can get the MinY values and identify if the LiDAR range is within 2.0 to 2.5.

However, there were still some issues with the solution. For example, if the vehicle had just crossed a traffic light, it would still slow down, even though there was no collision risk. Additionally, the vehicle would sometimes slow down if it was detecting poles or side barriers.

To address these issues, I added another plan that would check if the vehicle had just crossed a traffic light. If it had, the plan would not slow down the vehicle.

One extra step was to identify if you detect a front obstacle and there is a traffic light approaching within 8 game distance, then do not slow down. This reduced the possibility of accidental red-light infractions (Listing 4.10).

```

1  +!start(F): f(F,X,_,_, MinY) & MinY < 2.5 & X < 4.5
2      & info(F,Sp) & Sp > 0.5 & block(Nf, M) & Nf < M &
      ml_control(F,_,St,_,_,_) & traffic_light(F,_,_,_,_,D,_)
      & D < 8.0
3      <- no_action. // do nothing, let the ML model drive to prevent
      slowing down and causing the traffic light red infraction.

```

Listing 4.10: plan to return nothing if a traffic light is found within 8 units

These additional plans improved the remaining issues with the collision infractions, and the agent was able to drive safely by reducing the number of collision infractions.

4.7 Improving leaderboard evaluation time

The CARLA Leaderboard has a bug where the ego-vehicle occasionally stops without any reason. This was first noticed during the evaluation of route 5. The route 5 started with the ego-vehicle moving towards a red traffic light. Suddenly, the ML driver began to send brake controls. I thought this might be a temporary issue that would go away within a few frames. However, it continued for approximately 700 frames for that single route. For 700 frames, which is nearly 35 seconds of game time, the ego-vehicle applied the brakes.

This was strange because the only obstacle detection that is done is by our Jason agent. However, the ML driver was applying the brakes. This made no sense. To confirm, I tried running just the LAV agent for the Leaderboard and it returned the same result. For one last time, I tried with just the NPC agent, which has way-points coded. This is when I confirmed my suspicions: the NPC agent also applied the brakes for many consecutive frames (Figure 4.4).



Figure 4.4: The sudden stopping of the ego-vehicle

A quick look through the driver file and the recorder files revealed that this unknown weird thing happens when a red traffic light is coming ahead and an upcoming vehicle is coming from its opposite lane. The agent starts moving again when the traffic light turns green. It is important to note that the stopping is not because of the red traffic light, as the traffic light is still far away.

This was not a problem for our final results, but it did increase the time it takes to evaluate a route.

To solve the problem of the ego-vehicle stopping for no reason, I first tried testing the traffic-light distance at which the ego-vehicle is stopping. This distance was approximately 12-15 game distance. An 8 game distance within the traffic light is considered to be close enough to stop. However, applying controls directly in the Leaderboard or the orchestrator is not recommended. It should come from either ML agent or from Jason agent.

Therefore, the only way to solve the problem efficiently was to add the condition to Jason as a new plan. Currently, the traffic light red condition is handled by ML and both yellow and green are handled by Jason. Changing that to Jason would involve changing the triggering conditions for Jason. I needed to create two plans. One for traffic light red where the distance is less than 8 game distance, then stop. The second is if there is an upcoming traffic light and the distance to that is greater than 8 but less than 15, then continue moving.

```

1  +!start(F): traffic_light(F,"A","R",_._,D,_ ) & D > 8.0 & D < 15.0 &
      ml_control(F,_ ,St,_._,_ )
2      & info(F,S)
3      <- control(4, 0.3, St, 0.0, false, false, (S)); // Move
      forward
4      .print("===Traffic Light is not Red: Move ===: ", F).
5
6  +!start(F): traffic_light(F,_ ,"R",_._,D,_ ) & D < 8.0 & ml_control(F,_ ,
      St,_._,_ )
7      & info(F,S)
8      <- control(0, 0.0, St, 1.0, false, false, 5); // Stop
9      .print("===Traffic Light is Red: Stop ===: ", F).

```

Listing 4.11: New plans to control traffic light red

The first plan in Listing 4.11 checks if the distance to the traffic light is between 15 and 8 game distance. If it is, the car will move forward. The second plan checks if the distance to the traffic light is less than 8 game distance. If it is, the car will apply the brakes.

This worked perfectly for the first few routes that were tested. However, routes 20 to 36 contain highways and intersections where the traffic light is in a pole (inbox traffic light - Figure 4.5). Detecting the inbox parameter was important, which I did not consider at first. The initial Jason plans did not check for the inbox condition and kept moving when the distance was not yet 8. This caused collision with NPCs and road blockages.

One of the major challenges was, the inbox condition only gets invoked when the ego-vehicle is inside the traffic light inbox area. And the distance had to be again debugged. Upon running multiple evaluations and debugging the Leaderboard, I found out that the inbox traffic light distance should be 23.5 instead of 8. So I added a new plan where if the ego-vehicle is detecting an upcoming inbox red traffic light, and the distance is less than 23.5, then apply brake. The force moving was not required over here as the ego-vehicle did not show any signs of stopping before the traffic light.

```

1  +!start(F): traffic_light(F,"A","R",_._,D,0) & D > 8.0 & D < 15.0 &
      ml_control(F,_ ,St,_._,_ )
2      & info(F,S)

```



Figure 4.5: Traffic light inbox

```

3         <- control(4, 0.3, St, 0.0, false, false, (S)); // Move
          forward
4         .print("===Traffic Light is not Red(nb): Move ===: ", F).
5
6 +!start(F): traffic_light(F,_, "R",_,_,D,0) & D < 8.0 & ml_control(F,_,
          St,_,_,_)
7         & info(F,S)
8         <- control(0, 0.0, St, 1.0, false, false, 5); // Stop
9         .print("===Traffic Light is Red(nb): Stop ===: ", F).
10
11 +!start(F): traffic_light(F,"A","R",_,_,D,1) & D < 23.5 & ml_control(F,
          _,St,_,_,_)
12         & info(F,S)
13         <- control(0, 0.0, St, 1.0, false, false, 5); // Stop
14         .print("===Traffic Light is Red(box): Stop ===: ", F).

```

Listing 4.12: Plan modified to control traffic light red

The first two Jason plans in Listing 4.11 are modified to now include the `inbox` parameter as well, as show in Listing 4.12. The third plan in Listing 4.12 is to stop if there exist a traffic light inbox and the distance is less than 23.5.

Another problem that occurred was that the ego-vehicle stopped after moving forward inside the traffic light inbox area. On checking the driver file, I understood that one of the Jason plans to move from an inbox area was not always working.

```

1 +!start(F): traffic_light(F,"L",_,_,_,_,1)
2         & not sF(.,_,_,_,_)
3         & info(F,Sp) & Sp < 0.1 & ml_control(F,_,St,_,_,_)
4         <- control(5, 0.8, St, 0.0, false, false, 1). // Go forward

```

Listing 4.13: ML-MAS plan to move the car when the car is in the traffic light box

The plan shown in Listing 4.13 from ML-MAS to move when an ego-vehicle is in a traffic light inbox, did not work as the traffic light that the ego-vehicle was in focus has not changed its type from “Approaching” to “Leave”. This had to be changed to the plan shown in Listing 4.14

```

1 +!start(F): traffic_light(F,_, "G",_,_,_,1)

```

```
2      & info(F,Sp) & ml_control(F,_,St,_,_,_)  
3      <- control(5, 0.8, St, 0.0, false, false, 1). // Go forward
```

Listing 4.14: Updated plan to move the car when the car is in the traffic light box

Listing 4.14 plan works if there exist a traffic light in-Box that is Green, then move forward with a throttle of 0.8.

Chapter 5

Evaluation

This chapter presents the results of the evaluation of the ML-MAS extensions, in particular the traffic light control and collision control systems for the autonomous vehicle. The evaluation was conducted on the first 11 routes of longest6 benchmark.

5.1 Experiments

For the experiments, I selected the first 11 routes from the longest6 map for two main reasons:

1. These routes contain mostly small roads and turns. This allows me to understand how well the traffic light control and collision control infractions are performing.
2. The most common infraction in the longest6 benchmark is the stop sign infraction. The LAV model does not detect stop signs, which is a major issue. I could not find a way to obtain the stop sign information directly from CARLA. As a result, I could not fix the stop sign infraction problem. I decided to avoid testing the system on routes that contain stop signs. The first 11 routes do not contain any stop signs, so I am confident that the results of the evaluation are accurate.

I did 3 tests for the ML-MAS Extended and selected the best out of those. The results of the experiments are given in Table 5.1.

Table 5.1: Comparison of CARLA Leaderboard metrics for the first 11 routes in the longest6 benchmark (results are an average percentage of the total routes)

| Metrics | ML-MAS Extended | ML-MAS |
|-----------------------------|-----------------|--------|
| Avg. driving score | 72.826 | 37.692 |
| Avg. route completion | 97.391 | 85.306 |
| Avg. infraction penalty | 0.742 | 0.472 |
| Collisions with pedestrians | 0.000 | 0.020 |
| Collisions with vehicles | 0.010 | 0.030 |
| Collisions with layout | 0.000 | 0.015 |
| Red lights infractions | 0.151 | 0.249 |
| Stop sign infractions | 0.000 | 0.000 |
| Off-road infractions | 0.000 | 0.122 |
| Route deviations | 0.000 | 0.000 |
| Route timeouts | 0.016 | 0.015 |
| Agent blocked | 0.000 | 0.127 |

We can observe from these results that the modified version of the framework was able to successfully complete all 11 routes with minimum infractions. This is a significant improvement over the previous evaluation, where the framework managed to fail completing the routes due to route timeout or ego-vehicle getting blocked.

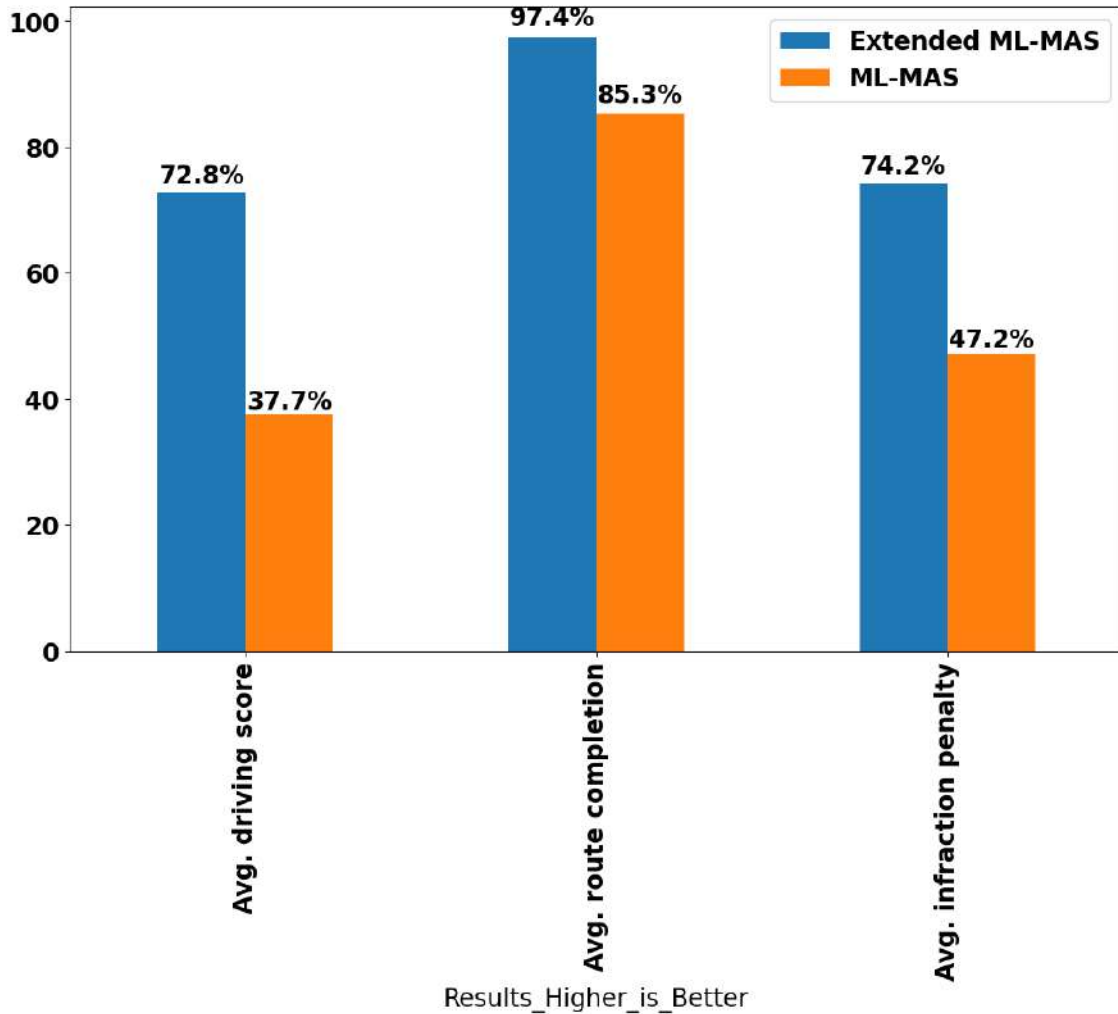


Figure 5.1: The driving score, route completion and infraction penalty for both runs

In Figure 5.1, the increased driving score of 72.826 in the ML-MAS Extended version compared to 37.692 in the ML-MAS shows that the car is able to follow the road and avoid collisions more effectively. This is due to the improvements made to the traffic light control and collision avoidance algorithms. The traffic light control algorithm now better takes into account the distance to the traffic light. Improvements to the traffic light type were also made, and the collision avoidance algorithm now better controls the speed of the car as well.

The increased route completion rate of 97.391 compared to 85.306 shows that the car is able to successfully complete more routes without any infractions in ML-MAS Extended. This could be because of less infractions and therefore the vehicle is less likely to get blocked or run out of time.

The reduced infraction penalty of 0.742 compared to 0.472 shows that the car is committing fewer infractions in ML-MAS extended. This is likely due to the reductions in the number of red

light and collision infractions.

5.2 Leaderboard Metrics

The leaderboard metrics for each route shows the following:

- infractions : the list of infractions with their coordinates.
- meta : the meta data for each route like route duration and route length etc.
- route_id : the id for each route.
- scores : The main scores like driving score, infraction penalty and route completion score
- status : The status for each route, whether its completed or failed.

Figure 5.2 shows the infraction count per route. The highest infractions are in route10 with an infraction count of 5.

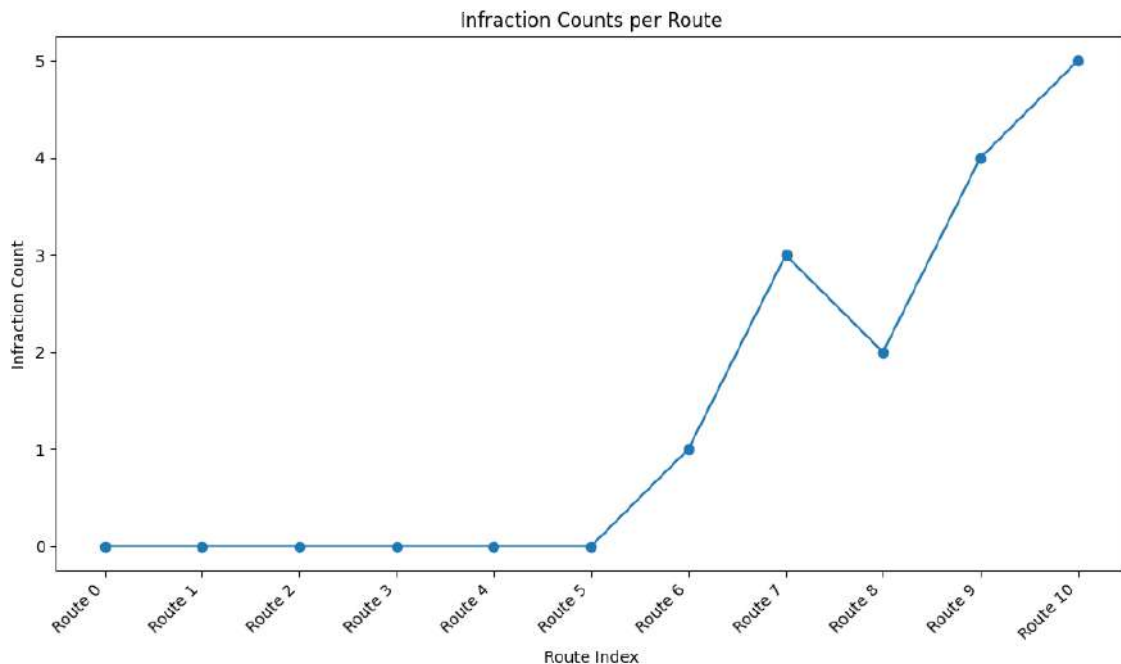


Figure 5.2: Infraction count per route

Routes 0 to 5 did not have any infractions and achieved a 100% driving score. This was because I tested and analysed these routes extensively while developing the Jason plans.

Routes 6 to 10 had red light infractions. This could be because of external factors, such as traffic congestion, or because the Jason plans were not designed properly. One idea to improve the Jason plans is to include a speed parameter in the traffic light green plan. This would slow down the ego-vehicle if it is moving too fast, giving it more time to stop at red lights.

Routes 9 and 10 also had route timeout infractions. This was likely due to the high number of infractions happening in those routes. Route 10 also had a going outside the designated route infraction, which may have contributed to the route timeout.

Route 10 had a collision infraction. This was not caused by a failure of the Jason plan. The ego-vehicle was trying to overtake a vehicle in front of it, but collided with it during the process. This could be prevented by implementing better lane control plans.

5.3 ML-MAS metrics

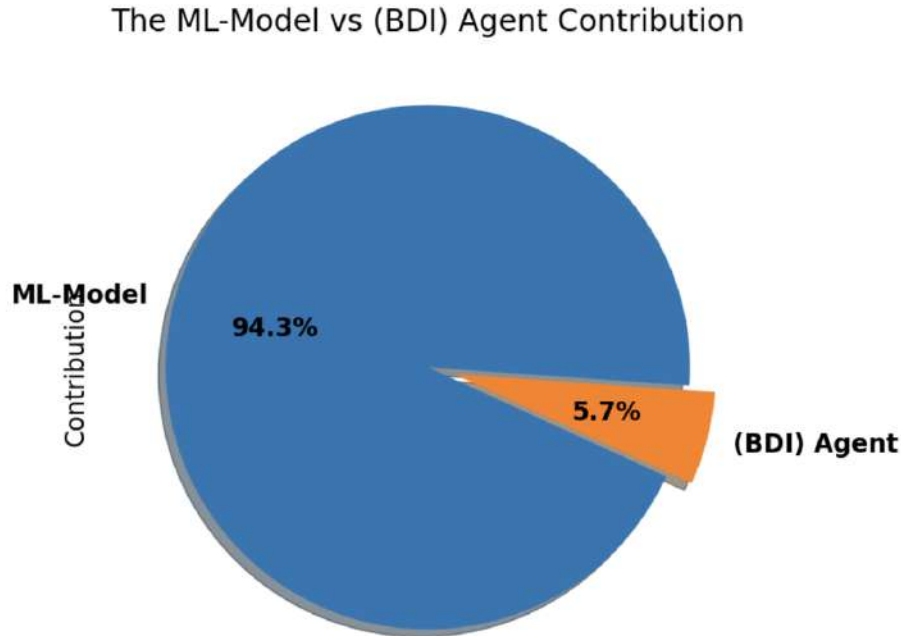


Figure 5.3: The ML-Model vs (BDI) Agent contribution

The contribution of both ML and Jason can be calculated using the driver log. The driver log contains information about each route, including who was the driver, the control that was issued, and the triggering condition for each frame. By calculating the sum of the drivers for each route and averaging them, we can get the results shown in Figure 5.3.

Figure 5.3 shows the contribution of the ML model versus the BDI agent. The Jason controlled the ego-vehicle 5.7% of the time as compared to the LAV model which did 94.3%. This shows that even with a small amount of symbolic reasoning, we can see significant improvements in the overall performance of the framework.

These are achieved by analysing and creating Jason plans that are necessary at scenarios where the ML model fails to act. There are a set of conditions where the control is transferred to Jason. Some of these Jason triggering conditions that were involved in the evaluation are as follows. Note that these conditions are created and detected by the orchestrator.

- front objects detection : When an obstacle is detected near the ego-vehicle front side.
- traffic light detection green : When the traffic light the ego-vehicle is focused on, is a green light.
- back objects detection : When an obstacle is detected near the ego-vehicle back side.

- traffic light detection red : When the traffic light the ego-vehicle is focused on, is a red light.
- traffic light detection yellow : When the traffic light the ego-vehicle is focused on, is a yellow light.
- right objects detection : When an obstacle is detected near the ego-vehicle right side.
- Left objects detection : When an obstacle is detected near the ego-vehicle left side.

Figure 5.4 shows the percentage of each triggering conditions in all the routes.

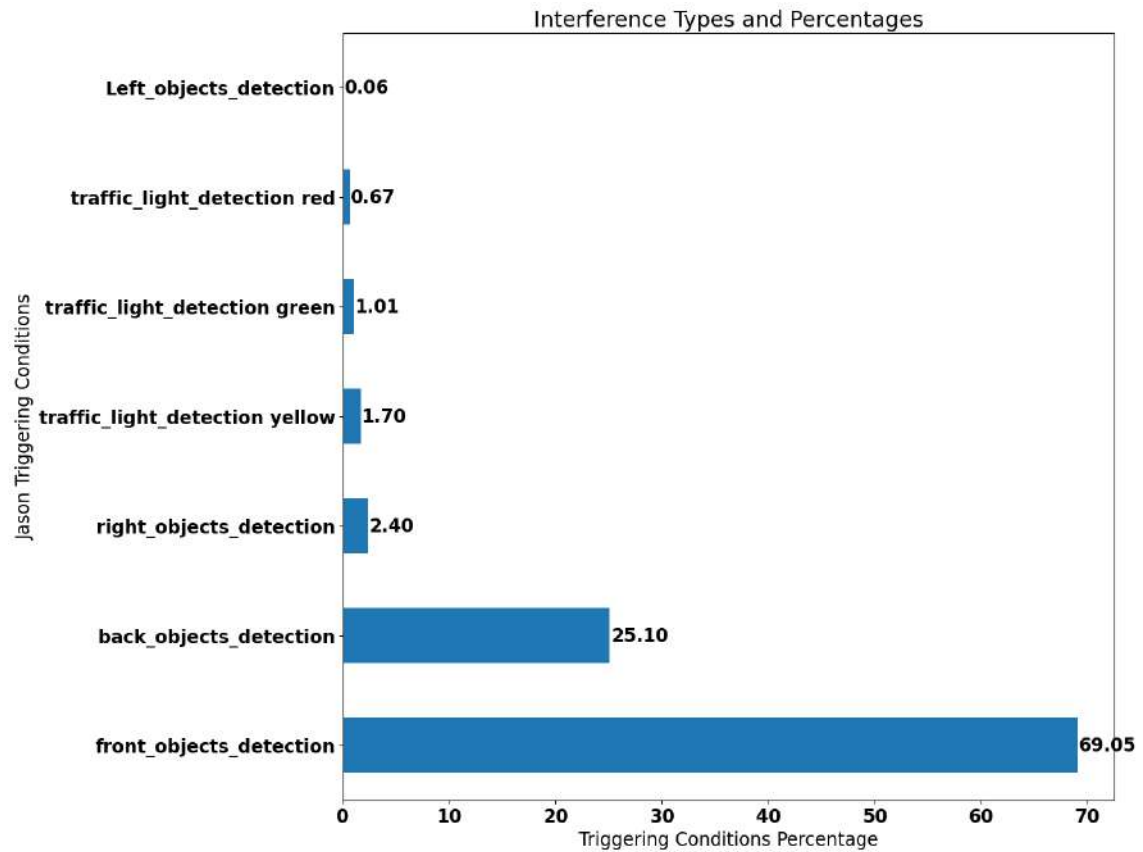


Figure 5.4: The Jason triggering conditions in all routes

The Jason plans controlled the ego-vehicle for the maximum amount of time when there was a front object detected by the orchestrator. This makes sense because the initial routes for the longest6 benchmarks are small roads containing lots of traffic and sudden crossing of pedestrians/bikes.

Figure 5.5 shows the distribution of infractions between the extended ML-MAS framework and the ML-MAS framework. The most common infraction in both frameworks is running a red light. However, the percentage of red light infractions is higher in the extended ML-MAS framework than in the ML-MAS framework. This may seem counter-intuitive, but it is because the overall number of infractions in the extended ML-MAS framework is much lower than in the ML-MAS framework. Figure 5.6 shows that the total number of infractions in the extended ML-MAS framework is 15, while the ML-MAS framework had 29 infractions. This difference in the number of infractions is why the percentage of red light infractions appear to be higher in the extended ML-MAS framework.

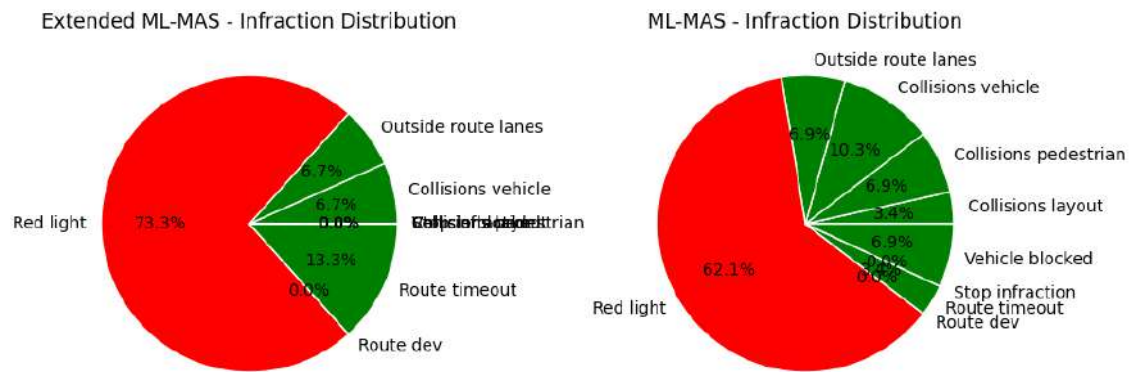


Figure 5.5: Overall Infraction Distribution : Extended ML-MAS vs ML-MAS

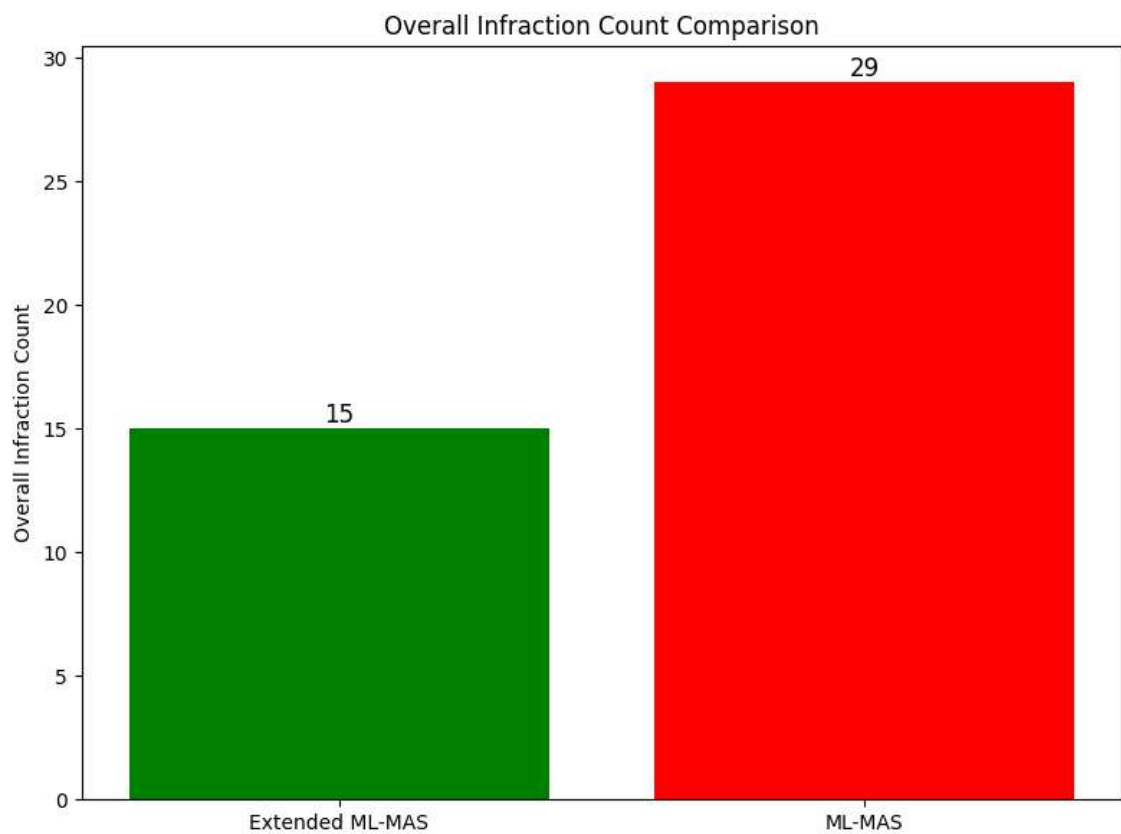


Figure 5.6: Overall Infraction Count : Extended ML-MAS vs ML-MAS

Figure 5.7 shows the number of infractions for each type of infraction in the extended ML-MAS framework and the ML-MAS framework. The extended ML-MAS framework was able to reduce the number of infractions for all types of infractions except route timeout. This is a significant improvement, as it means that the extended ML-MAS framework is able to drive more safely.

The increase in route timeout infractions could be a potential limitation of the extended ML-MAS framework. This is because it means that the car is taking longer to complete the route. This could be due to the ego-vehicle being more cautious on avoiding infractions, or it could be due to other factors, such as traffic congestion.

It is important to investigate the cause of the increase in route timeout infractions in order to determine if it is a significant limitation of the extended ML-MAS framework. If it is a significant limitation, then it may be necessary to make changes to the framework in order to reduce the number of route timeout infractions.

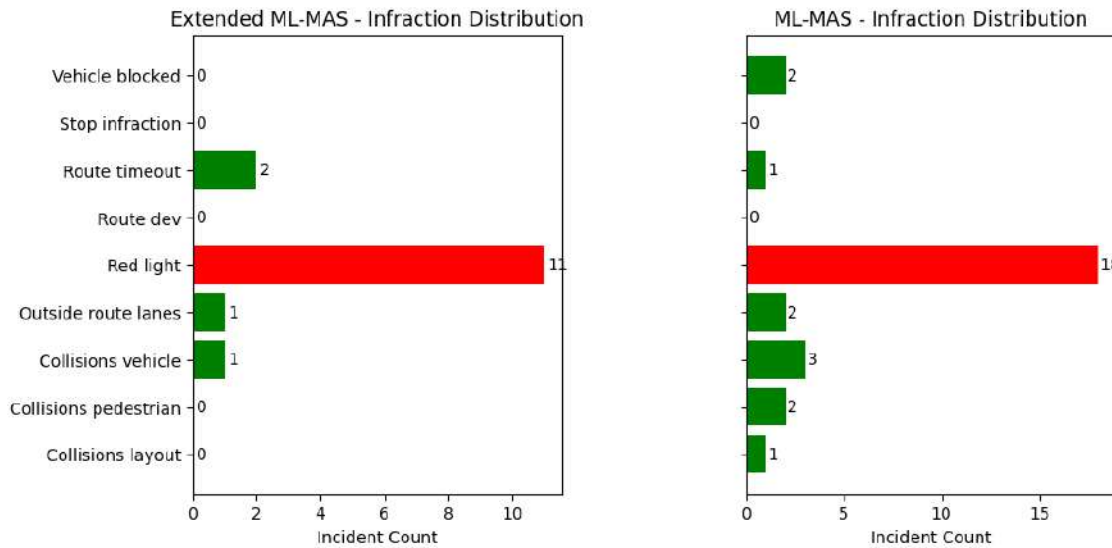


Figure 5.7: Infraction Distribution : Extended ML-MAS vs ML-MAS

5.4 Discussion

Overall, the red light infraction and the collision infraction plans seemed to work perfectly. However, there are a few factors that could be included to improve the handling of red lights, such as the speed of the ego-vehicle.

For example, if the ego-vehicle is moving too fast towards a green light, it may not be able to stop in time if the light turns yellow. In this case, the plan could be modified to slow down the ego-vehicle before it reaches the intersection.

In case for the collision infraction in route 10, the infraction caused because of the vehicle trying to overtake the vehicle in front due to traffic, but instead collided while turning, causing the collision infraction. I believe this could be caused due to the ego-vehicle assuming that the vehicle in front is an obstacle. This could be improved by implementing a better obstacle detection plane or including the lane detection as well.

In route 10, the ego-vehicle collided with the vehicle in front while trying to overtake it.

This was likely caused by the ego-vehicle assuming that the vehicle in front was an obstacle. There are a few things that could be done to improve the obstacle detection in this case. First, the obstacle detection plane could be improved by using better obstacle detection plan to get a good understanding of the environment around the ego-vehicle. Second, lane detection could be included in the obstacle detection process to help the ego-vehicle to keep in the lane that its required to and not take unnecessary overtaking controls.

By implementing one or both of these changes, the collision infraction in the framework could be improved. This would help to prevent collisions, such as the one that occurred in route 10.

The driver log file was a critical addition to the framework. It allowed me to analyse each frame individually and design plans accordingly. For example, the red light infraction plan was designed to address the issue of the ego-vehicle running red lights because it was not able to detect yellow lights in time. The plan included detection of yellow lights and with the modification of the orchestrator "Approaching" status for the traffic lights, the infraction was reduced.

The collision infraction plan was also designed from the driver file. It allowed me to address the issue of the ego-vehicle colliding with pedestrians and bikes crossing the road. The plan included a slow-down process if it detected an upcoming obstacle, giving it more time to stop if the obstacle is in front of the vehicle.

By using the driver log file, I was able to design better Jason plans that prevented infractions. This made the framework more safe and reliable.

In addition to the red light infraction plan and the collision infraction plan, I also designed other Jason plans based on the driver log file. These plans helped to prevent other types of infractions, such as traffic light in box and to handle the unwanted breaking bug of CARLA leaderboard.

The driver log file was a valuable tool that helped me to improve the framework. It allowed me to analyse and understand what was happening in each frame, which helped me to design better Jason plans.

There were some plans that did not work as expected. For example, one plan to address the red light infraction was to identify if there was a yellow light and then do a small reverse action. However, this plan failed because it led to potential back collisions.

Another plan that did not work was to address the stop infraction. This is the most common infraction across the framework, so I tried to get input from CARLA to create a plan. However, this plan did not work because the leaderboard way-points are just a combination of coordinates and there was no way to identify if a stop sign is approaching. As a result, the stop sign infraction still continues.

Chapter 6

Conclusion

This chapter talks about the limitations and future scope that could improve the extended ML-MAS framework.

6.1 Summary

In my project, I extended the ML-MAS framework by integrating new Jason plans that handle more complex and diverse driving scenarios, such as yellow lights, collisions, and agent blocking. I modified the BDI bridge code to enable seamless communication between Python (CARLA API) and Java/Jason, and to detect and report all infractions that occurred during the evaluation. I configured the CARLA simulator for Windows and created a new NPC agent that can interact with the ego-vehicle and other traffic participants.

I improved the debugging and analysis of routes using logs, which recorded information such as the number of frames, collisions, infractions, and agent switches for each route. I evaluated my ML-MAS framework using the longest6 benchmark, which consists of different challenges and weather conditions. I compared my results with the previous ML-MAS framework and found that my framework achieved higher driving scores, route completion rates, and infraction penalties than the previous ML-MAS framework.

I discussed the strengths and weaknesses of my framework, highlighting the advantages of combining ML and MAS for autonomous driving. Overall, my project demonstrated the potential of neuro-symbolic AI for improving the driving performance of self-driving cars in complex and dynamic environments.

6.2 Limitations

One of the major limitations of this framework is the red light infractions. There are still red light infractions happening in the final run for route 6 to 11. Analysing the log files, I believe that the red light infractions in Figure 5.7 are likely caused by the speed with which the ego-vehicle is moving towards the green light. The LAV model moves the vehicle with high speeds, which is ideal for completing the route faster in less traffic. However, when a sudden obstacle or a need to stop appears, this can backfire and cause delayed response. This is what happened in the cases of red light infractions. The ego-vehicle was moving too fast to stop for the green light, and as a result, it ran the red light. The same was repeated for collisions but that was handled properly.

```
1 | // A plan to move the car when the traffic light turns green.
```

```

2  +!start(F): traffic_light(F,"A","G",_,_,_,_) & ml_control(F,_,
    St,_,_,_)
3      & not sF(.,.,.,.,_) & info(F,S) & S < 0.5
4      <- control(4, 0.8, St, 0.0, false, false, 3); // Go
    forward
5      .print("===Traffic Light is Green: Go ===: ", F).

```

Listing 6.1: Jason plan for green light

Listing 6.1 shows the Jason plan that currently handles the red light infraction. This plan is triggered when the following conditions are met:

- The `traffic_light` predicate has the parameters ("A", "G") which means there is an approaching green traffic light.
- The `ml_control` predicate provides information about the ml control.
- The 'not sF' predicate is to check if there are no side-front obstacles detected.
- The `info` predicate provides information about the speed with which the ego-vehicle is moving.
- and that speed should be less than 0.5

And if these conditions are met, there is a 0.8 throttle with the given steering control returned.

The value of Speed (S variable in the `info` predicate) is less than 0.5. That means if there is a green light and the agent has just started to move, continue moving forward. This plan handles those scenarios where the ego-vehicle has to start moving because it stopped for a red/yellow light. This works fine, but what about those scenarios where the ego-vehicle is moving towards a traffic light that is already green. The ML part takes control of those scenarios. There are 2 possibilities, If the agent is moving towards a green light with speed that is controllable by the Jason for the yellow/red stop plan to act correctly and If the agent is moving towards a green light with speed that is too high for the Jason to invoke yellow/red stop plan causing red light infractions. The later should be controlled by Jason by adding a new plan like that's shown in Listing 6.2.

```

1  // A plan to move the car when the traffic light turns green.
2  +!start(F): traffic_light(F,"A","G",_,_,_,_) & ml_control(F,Tt,
    St,_,_,_)
3      & not sF(.,.,.,.,_) & info(F,S) & S > 5.5
4      <- control(4, Tt, St, 0.4, false, false, 3); // Slow
    down for green
5      .print("===Traffic Light is Green: Slow down ===: ",
    F).

```

Listing 6.2: New extra plan for green light

The new updated plan applies a brake of 0.4 if it detects a speed greater than 5.5. This should be able to control the ego-vehicles speed as it approaches a traffic light that is green.

The collision in route 10 was not caused by over-speeding, but by poor lane control. The ego-vehicle has a very poor understanding of what lane it should be in, and the only thing that is keeping it in its lane is the ML part of the framework.

However, this has limitations, such as the route timeout infraction. This infraction occurs when the agent moves into the opposite lane, causing traffic congestion and road blocks. This takes more time to complete the routes than is required.

Another limitation of the framework is the evaluation time. It takes around 3 to 4 days to evaluate the whole longest6 benchmark with the extended ML-MAS framework. This makes debugging after analysing all the routes very time-consuming. There are a bunch of time delay functions across the bridge and the orchestrator, but changing them causes more problems as they are necessary for the orchestrator to get proper response from the bridge.

6.3 Future Work

Looking at all the routes of the longest6 benchmark, the highest infraction that was noted was the stop sign infraction. (See Figure 3.3). I tried to get the information about stop signs from the CARLA leaderboard, but I could not find a way to convert the leaderboard way-points to a form where we can identify the stop signs. Since the LAV model also fails to detect stop signs, this results in very poor driving scores overall.

A solution to this problem could be to understand and convert the leaderboard way-points for each route to a list with the stop signs and their locations, just like how traffic lights are detected in the orchestrator. This would allow the ego-vehicle to avoid stop sign infractions by slowing down or stopping before it reaches a stop sign.

Understanding the leaderboard way-points would also enable me to create a lane control plan for the vehicle. This would help to reduce unnecessary over-takings, wrong lane driving due to improper turnings, agent blocking, and route timeouts.

Here are some ideas on how a lane control plan could be implemented in Jason:

- The ego-vehicle could be instructed to stay in the center of its lane. This would help to prevent it from drifting out of lane and causing collisions. This could also reduce ego-vehicle slowing down as its driving away from the side railings.
- The ego-vehicle could be instructed to change lanes when necessary, such as when turning or overtaking another vehicle. This would help to improve traffic flow and prevent lane blocking. This is done by the LAV agent for now, but it would be interesting to see how Jason controls it.
- The ego-vehicle could be instructed to slow down or stop when approaching a lane change, to ensure that it does not cause a collision. Collision during lane change is a constant issue in ML-MAS as the LAV model do not check any upcoming vehicles in the changing lane.

The ML-MAS framework had some issues with collisions with pedestrians in highway routes. This was likely due to the speed of the ego-vehicle. The ego-vehicle was moving in high speeds in highways, and sudden crossing of pedestrians created a problem.

The extended ML-MAS framework has plans that handle collisions more accurately. This should be tested to see if it can avoid colliding with pedestrians in highways.

Route deviations were also an important infraction in the ML-MAS framework. This is because it fails the entire route evaluation. One possible solution to this problem is to add lane control. If the ML is deviating from the path it is supposed to follow, and has deviated for a certain distance, Jason could take control and drive back to its destined path. This should solve the route deviation issue.

There were many such issues noted during the initial run of the extended ML-MAS framework. However, I was not able to solve all of them because the time to debug each route is high. One possible solution to this problem is to make the simulation times faster in future versions of the framework.

Bibliography

- Al-Nuaimi, M., Wibowo, S., Qu, H., Aitken, J. M., and Veres, S. M. (2021). Hybrid verification technique for decision-making of self-driving vehicles. *J. Sens. Actuator Networks*, 10:42.
- Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete problems in ai safety.
- at Work, I. (2023). How data is being used to train autonomous vehicles to navigate roadways.
- Besold, T. R., d’Avila Garcez, A., Bader, S., Bowman, H., Domingos, P., Hitzler, P., Kuehnberger, K.-U., Lamb, L. C., Lowd, D., Lima, P. M. V., de Penning, L., Pinkas, G., Poon, H., and Zaverucha, G. (2017). Neural-symbolic learning and reasoning: A survey and interpretation.
- Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.
- Cardoso, R. C. and Ferrando, A. (2021). A review of agent-based programming for multi-agent systems. *Computers*, 10(2).
- Chen, D. and Krähenbühl, P. (2022). Learning from all vehicles. In *CVPR*.
- Coraggio, P. and De Gregorio, M. (2007). A neurosymbolic hybrid approach for landmark recognition and robot localization. In Mele, F., Ramella, G., Santillo, S., and Ventriglia, F., editors, *Advances in Brain, Vision, and Artificial Intelligence*, pages 566–575, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16.
- Dworak, D., Ciepiela, F., Derbisz, J., Izzat, I., Komorkiewicz, M., and Wójcik, M. (2019). Performance of lidar object detection deep learning architectures based on artificially generated point cloud data from carla simulator. In *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 600–605. IEEE.
- GREGORIO, M. D. (2008). An intelligent active video surveillance system based on the integration of virtual neural sensors and bdi agents. *IEICE Transactions on Information and Systems*, E91.D(7):1914–1921.
- Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux, New York.
- Kwiatkowska, M., Norman, G., and Parker, D. (2011). Prism 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Leaderboard, C. A. D. (2019). Carla autonomous driving leaderboard.
- Lomuscio, A., Qu, H., and Raimondi, F. (2009). Mcmas: A model checker for the verification of multi-agent systems. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 682–688, Berlin, Heidelberg. Springer Berlin Heidelberg.

- Perry, J. and Sevil, H. E. (2022). A multi-agent adaptation of the rule-based wumpus world game. *International Journal of Artificial Intelligence and Soft Computing*, 7(4):299–312.
- Rao, A. and Georgeff, M. (2000). Bdi agents: From theory to practice.
- Sarker, M. K., Zhou, L., Eberhart, A., and Hitzler, P. (2021). Neuro-symbolic artificial intelligence: Current trends.
- Shao, H., Wang, L., Chen, R., Li, H., and Liu, Y. (2022). Safety-enhanced autonomous driving using interpretable sensor fusion transformer.
- Shao, H., Wang, L., Chen, R., Waslander, S. L., Li, H., and Liu, Y. (2023). Reasonnet: End-to-end driving with temporal and global reasoning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13723–13733.
- Shukairi, H. A. and Cardoso, R. C. (2023). Ml-mas: a hybrid ai framework for self-driving vehicles. In *Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '23*.
- Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks.
- Wang, L., Hu, Y., Sun, L., Zhan, W., Tomizuka, M., and Liu, C. (2022a). Transferable and adaptable driving behavior prediction.
- Wang, W., Yang, Y., and Wu, F. (2022b). Towards data-and knowledge-driven artificial intelligence: A survey on neuro-symbolic computing.
- Zheng, N., Du, S., Wang, J., Zhang, H., Cui, W., Kang, Z., Yang, T., Lou, B., Chi, Y., Long, H., Ma, M., Yuan, Q., Zhang, S., Zhang, D., Ye, F., and Xin, J. (2020). Predicting covid-19 in china using hybrid ai model. *IEEE Transactions on Cybernetics*, 50(7):2891–2904.