MOTORVEHICLE
UNIVERSITY OF
EMILIA-ROMAGNA

UNIVERSITÀ
DI PARMA

Electronic Engineering for Intelligent Vehicle
Autonomous Driving Engineering

Reinforcement Learning applied to ADAS testing

Thesis Advisor:
Prof. Nicola Mimmo

Graduand:
Valerio Tiri

Company Supervisor:
Ing. Pasquale Sessa

Hosted by:
Maserati SpA

Academic Year 2023/2024

Reinforcement Learning applied to ADAS testing

the culmination of a long journey

Abstract

Among the various stages required by an automotive company to bring a brand-new vehicle to completion, testing plays a pivotal role. In this study is proposed a novel approach to enhance the procedural testing methods used in the development of certain new Maserati vehicles by integrating an agent into the testing process. This agent is, in fact, a neural network trained through Reinforcement Learning. In this thesis, are then explored the motivations that led to select this innovative approach, as well as the underlying mechanisms of the technology. The main goal was to optimize and automatize the testing process, providing higher efficiency and accuracy in assessing vehicle performance, and reliability, while keeping pace with the cutting-edge technological advancements of the Maserati brand.

Contents

List of Figures

List of Tables

Acronyms

ADAS Advanced Driver Assistance Systems. 2–4, 8, 20, 23, 25, 67, 75, 76

DQN Deep Q-Network. 40, 49–51, 53, 63

ECU Electronic Control Units. 2, 4, 7, 23, 25, 27, 28, 30, 75, 76

HDC Hill Descent Control. 25, 27, 67

HiL Hardware in the Loop. 7, 9, 11, 20, 21, 25, 28, 37, 64, 76

LSTM Long Short Term Memory. 40–42

NN Neural Network. 10, 11, 18, 19, 79–81

PPO Proximal Policy Optimization. 40, 49, 53, 63

RL Reinforcement Learning. 11–14, 16, 19, 20, 35, 37–39, 43, 45, 55

SBT System Behaviour Test. 8, 9

VTD Virtual Test Driver. 9, 21, 28–30, 32, 33, 35, 37, 44, 67, 69–73, 75, 77

Chapter 1

Introduction

This chapter outlines the environment in which this thesis was developed, highlighting the current state of the art in automotive testing. Then an overview on the advance-ments in deep learning and, more precisely, in Reinforcement Learning is given. A detailed examination of the latest trends and methodologies in vehicle testing will be provided, alongside a review of how Reinforcement Learning can been applied in similar contexts. This foundational knowledge is essential to understand the motivation behind the innovative approach presented in this work.

1.1 Motivations

This section highlights the growing importance of electronics in modern vehicles, with a particular focus on the historical and economic factors that have led to the development of this thesis. Additionally, it explores the critical role of testing in automotive systems and emphasizes how the integration between components is essential for ensuring seamless functionality in complex vehicle systems.

1.1.1 Electronics in automotive field

In recent years, the automotive industry has experienced significant changes with the growing integration of electronic systems. These innovations are designed to address market demands and improve vehicle performance across the board. Consumers are now looking for enhanced driving comfort and advanced assistance features, while governments continue to tighten regulations on vehicle emissions. The advancements

in Hybrid Electric Vehicles (HEVs) and Full Electric Vehicles (EVs) have further accelerated this trend.

Advanced Driver Assistance Systems (ADAS) have been developed to mitigate the driving stress and for reducing driver fatigue, by intervening in specific situations where the driving comfort could be further improved and reduce human error on the road.

Naturally, Maserati, a brand known for producing high-end vehicles in premium market segments, such as the Maserati Ghibli (segment E) and the Maserati MC20 (segment S), integrates the latest technologies into its cars, including cutting-edge advancements in the field of ADAS. These systems reflect the brand's commitment to safety and innovation, ensuring that Maserati vehicles offer both luxury and state-of-the-art driver assistance features. To fully grasp the level of complexity achieved in the automotive industry, it is essential to consider the range of ADAS integrated into Maserati vehicles. The following is a list of ADAS features, as highlighted on the official Maserati website.

- Adaptive Cruise Control with Stop and Go
- Lane Keeping Assist
- Surround View Camera
- Highway Assist System
- Active Blind Spot Assist
- Traffic Sign Recognition
- Forward Collision Warning Plus
- Hill Descent Control

It is also evident that each of these ADAS requires significant computational power to function properly and *meet the real-time constraints* imposed by its specific application. For instance, is evident how Traffic Sign Recognition may have more relaxed real-time requirements compared to the Forward Collision Warning system, due to the differing nature of their functions. As a result, the number of electronic components, particularly Electronic Control Units (ECU), needed to handle this computational load tends to increase significantly. In modern high-end vehicles, it is not uncommon to find up to 150 ECUs [1] and more than 200 sensors [2] working together to ensure optimal performance and safety.

1.1.2 Integration and Suppliers

Contrary to common perceptions, the modern automotive industry has significantly diverged from its early foundations. While technological advancements are often highlighted, the transformations extend deeply into management practices and industrial strategies. In the pioneering era of the automotive sector, Henry Ford established a vertically integrated supply chain encompassing raw materials, manufacturing, and assembly. This approach aimed to reduce costs, enhance efficiency, and ensure stringent quality control by maintaining full control over the entire production process.[1]

Following World War II, the automotive industry underwent substantial changes driven by technological progress and a strategic emphasis on diversification. Companies began to diversify their operations and concentrate on their core competencies, leading to a decline in vertical integration as manufacturers increasingly outsourced components and systems to specialized suppliers. This trend was further accelerated in the new century by globalization, which expanded supply chains across international borders.

In recent years, the emergence of electric vehicles (EVs) and autonomous driving technologies has spurred a renewed interest in vertical integration. Automotive companies are investing heavily in the development of batteries, electric drivetrains, and ADAS. By vertically integrating these critical technologies, manufacturers can exert greater control over key components and ensure their seamless compatibility with vehicle designs. Big players such as Toyota, Tesla, and BYD are pursuing comprehensive vertical integration strategies to retain control over their processes and preserve proprietary knowledge.

Nonetheless, the trend of horizontal integration—the counterpart to vertical integration—continues to prevail, particularly among European manufacturers, including Maserati, which operates under the Stellantis Group. Horizontal integration involves the consolidation of similar or complementary businesses to enhance market presence and leverage synergies, and it remains a dominant strategy in the contemporary automotive landscape.

The reasons outlined above briefly explain why Maserati relies on external suppliers. During the development of a new vehicle, the manufacturer first designs the vehicle with a clear vision of the desired final product. For each production segment, suppliers are carefully selected to ensure that the performance and costs align with the project's requirements. Specifically, in the electronic domain—central to this

[1] Vertical integration is a strategy that companies use to streamline their operations. It involves taking ownership of various stages of its production process. Companies achieve vertical integration through mergers or acquisitions or establishing suppliers, manufacturers, distributors, or retail locations rather than outsourcing them. Vertical integration often requires significant initial capital investment.[3]

work—Maserati selects the necessary electronic components to meet the targeted features, performance standards, and safety regulations. Based on these specifications, suppliers are chosen to provide electronic components **and ADAS systems**. These suppliers are specialized in producing sensors, control modules, radars, and other hardware; while also delivering the necessary software to ensure the proper functioning of these systems. The selection of the right supplier is a critical task for managers, as poor choices can result in underperforming systems or failure to comply with stringent safety regulations. Maserati is proud to collaborate with renowned companies in this field, such as Bosch, Brembo, and Dallara, ensuring that the vehicles are equipped with top-tier systems and components.

Given these considerations, it is clear that a significant part of Maserati's work involves integrating various subsystems provided by multiple external suppliers to ensure that everything functions as expected. The integration process is crucial and can be divided into three main areas:

Hardware Integration This involves the strategic placement of physical components, such as sensors and other electronic devices, both around and inside the vehicle. Ensuring that these components are positioned optimally is essential for accurate data collection and system performance.

Software Integration The data gathered by the sensors is processed by multiple ECUs, which run complex algorithms to interpret inputs and make decisions. Software from various suppliers must be carefully integrated into the vehicle's overall software architecture to ensure seamless operation.

Communication Systems Components communicate through in-vehicle networks, such as the CAN bus or Ethernet. Ensuring smooth interaction and efficient data sharing between these systems is critical to the overall functionality of ADAS.

The primary challenge in this integration process lies in achieving smooth communication and compatibility between the different subsystems. Timing and synchronization are particularly crucial, as ADAS requires real-time responses. Ensuring that data from all sensors is processed and acted upon without delay represents a significant technical challenge.

1.1.3 Testing purposes

Ensuring that the systems provided by suppliers, once installed and integrated into the vehicle, work correctly in all expected scenarios is the primary goal of **testing**.

In this context, a test object is broadly defined as any "work product to be tested", which could be:

- one unit
- a compilation of several software parts
- a complete software program
- a control unit
- a network of several control units
- a whole vehicle
- any other object to be tested

When conducting a specific test, we refer to it as a *test case*. A test case always includes at least two essential pieces of information: first, the test data, which specifies how the test object should be stimulated; and second, the expected outcomes, which define the computations or states the test object should exhibit during stimulation. Additionally, a test case can be supplemented with other relevant details, to provide context and ensure proper testing conditions [4]. Once the individual components have been validated, the Continuous Integration (CI) process is initiated. Depending on the complexity of the overall software, the CI pipeline may require anywhere from a few to several hundred intermediate stages for the integration tests. Typically, these tests are developed using a bottom-up approach, where a few units are integrated and tested together first. The resulting composite is then combined with other previously tested composites or units at the next intermediate stage, and the process is repeated. This iterative chain continues until the entire system has been fully integrated and tested. Although the high number of integration tests may initially seem labor-intensive, this approach offers a significant advantage since it allows for faster and more accurate detection of errors by ensuring that problems are identified and addressed at an early stage in the development process.

More generally there are two ways in which the testing of a specific component or a group of them can be performed.

Static testing where the test object is not stimulated, but analyzed statically. An example of a static test is the review of a source code file.

Dynamic testing a test case is created and executed that stimulates a test object with the test data. The stimulation causes the test object to either perform a calculation or change its state. The reaction of the test object is recorded

in dynamic testing and compared with an expectation value. If the reaction is equal to the expectation, the test case is considered to have passed. If it is not equal, it is considered to have failed.

As we move on to the final aspects of test development theory, it is important to define the *Test Environment*. A test environment can be thought of as a training ground for the test object and its various components. The test environment should closely replicate the real-world production environment to ensure that interactions between the test object, other systems, and signals are as meaningful as possible. Before testing on the vehicle itself, four key test environments are typically defined, as also illustrated in Figure 1.1:

- Model-in-the-loop (MiL)
- Software-in-the-loop (SiL)
- Processor-in-the-loop (PiL)
- Hardware-in-the-loop (HiL)

In these environments, the term preceding "in-the-loop" refers to the type of test object being evaluated. The "in-the-loop" aspect refers to the interaction between the test object and components within a simulated production environment. Unlike open-loop tests—where the environment does not react to the test object's states or calculations—in the "in-the-loop" tests the environment responds dynamically, making these tests more realistic and valuable for assessing system behavior.

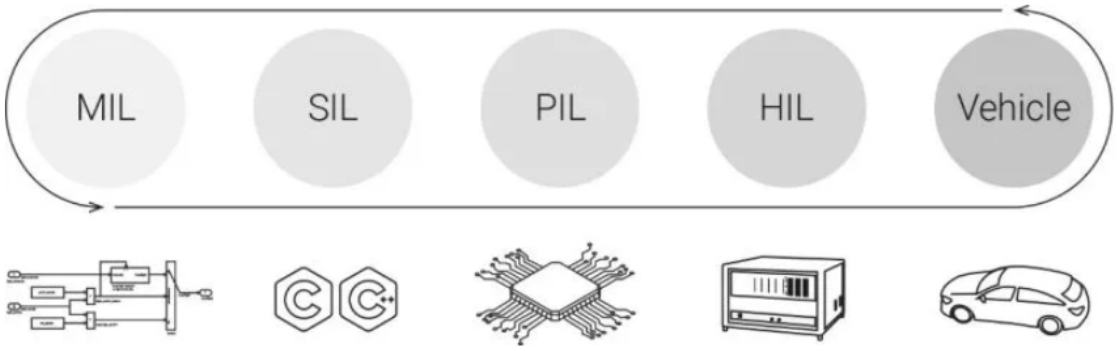

Figure 1.1: "in-the-loop" Test Environments. Source: [4]

In the automotive industry, development is often model-based. Most models are created using MATLAB/Simulink and are typically validated in the Model-in-the-Loop (MiL) environment. This early stage of testing, conducted within the development environment, allows for quick detection and correction of errors during the

initial phases of model construction. MiL is followed by Software-in-the-Loop (SiL) testing, which is applied for unit testing, integration testing, and, in some cases, software testing. At this stage, hardware is not yet involved, focusing purely on software behavior.

Processor-in-the-Loop (PiL) testing follows, aimed at detecting compiler errors and checking the compatibility of software components with hardware, particularly in cases where software interacts closely with hardware (such as drivers or actuator control). This stage ensures that software and hardware elements work harmoniously before moving on to physical testing.

The next logical step is HiL testing, where the fully developed software is tested on physical ECUs and peripherals. This stage emphasizes real-time interaction between inputs, outputs, communication buses, and other interfaces. HiL test benches can be highly sophisticated and are capable of testing entire vehicles, though these setups can be costly and complex to operate. In vehicle testing, components such as ECUs, actuators, and sensors are tested in their final target environment. Vehicles are typically subjected to a variety of environmental conditions, including cold, warm, and hot climates. Despite advances in technology, much of this testing is still performed manually, although some measurements are automatically recorded and later analyzed using specialized tools. It is within this HiL test environment that this work makes its contribution showing off a new approach in partially automatize and expand the possibility of behavioural test that could be performed on a in-developing project for a new vehicle.

1.1.4 Testing approaches

Conducting all the tests described in the previous section, especially for large-scale projects like those in the automotive industry, is both time-consuming and costly. As part of a capitalistic economic system, automotive companies are constantly seeking ways to reduce costs and increase profits. In the testing field, the most effective strategy is to automate as much of the process as possible. Automation not only accelerates testing but also minimizes human error, limiting manual intervention to only those areas where human involvement is still necessary.

Manual testing

A particular challenge arises when tests require driving, a task primarily conducted in HiL and vehicle tests. This is an inherently expensive activity for several reasons. For instance, when tests need to be carried out in an urban environment to evaluate the

integration of various ADAS features—such as Traffic Sign Recognition and Lane Keeping Assist—a specialized human test driver is required. Test drivers possess unique skills and expertise, and given that automotive companies often run multiple projects simultaneously, their availability is not always guaranteed, which can create bottlenecks in the testing process.

Moreover, performing the tests themselves often presents logistical challenges. It is not always feasible to find or replicate ideal testing conditions on the road. Many environmental factors—such as weather, road surface conditions, and the behavior of surrounding vehicles—are unpredictable and difficult to control. This unpredictability is particularly problematic when reproducibility is crucial, such as in cases where a software bug is discovered under specific conditions that are difficult to replicate, becoming a problem validate the software's updates. Additionally, human factors introduce variability into test results. A driver's emotions, fatigue, and handling skills can influence the outcome, making it difficult to obtain objective and consistent data. Lastly, it is important to consider the inherent risks of on-road testing, which always implies a risk of damaging people and properties.

First automatization

A commonly used method for automating not only integration tests but also other testing processes is *Script-Based Testing*. In this approach, scripts are created as sequences of instructions that simulate actions such as pressing pedals, turning the steering wheel, or pressing buttons. These scripts automate specific test sequences by simulating inputs and verifying outputs in a repeatable and controlled manner, marking a significant step toward automation—even in its simplest form, such as the **monkey test**. This type of test involves sending random inputs to the system to observe its behavior, helping to identify errors or unexpected responses. Due to the randomness property of this type of test it can be easily automatized but is still not the most effective tool for testing such systems.

Despite these advancements, script-based testing still requires significant effort from test engineers. The test sequences must be manually written, and interpreting the directives from validation documents can be challenging. These documents often provide only a high-level description of the testing purpose, leaving the engineer responsible for properly configuring the *Factors* (pedals, buttons, and other elements involved in the test) and defining the correct *Levels* for every chosen *Factor*. While some levels are simple to define—such as buttons with binary states (on/off)—others, like speed or target velocities, can be more complex to set.

Recognizing these limitations, Maserati developed its own method to more effectively test system functionality and integration: **System Behavior Testing** (SBT),

which is more advanced then the *monkey test* but still partially relies on *Script-Based testing*. This approach evaluates how a system reacts across a wide range of operational scenarios, which is crucial in the automotive industry for validating complex systems like automatic braking or driver assistance. Tests simulate extreme operating conditions or difficult situations to ensure that the system reacts safely and effectively.

Even with the introduction of this powerful tool, Maserati continues to explore ways to further automate the testing process, seeking to reduce manual intervention and increase testing efficiency.

VTD

Given these considerations, imagine having a tool that can autonomously perform System Behavior Testing (SBT) without requiring intervention from a test engineer—except at the end of the simulations to evaluate the overall results in terms of bug detection and correct system behavior. The advantages of such a system, referred to as a **Virtual Test Driver** (VTD), are evident. A VTD capable of executing a wide range of operations and efficiently covering all necessary test cases in an automatic, safe, repeatable, and cost-effective manner offers substantial benefits.

The system developed in this thesis aims to automate SBT by eliminating the need for engineers to manually select test patterns. The VTD is designed to integrate with the Hardware-in-the-Loop (HiL) testing environment, where its primary function is to conduct exclusively virtual tests. It serves as the core component of a more extensive system developed by Maserati to enable comprehensive virtual testing, which will be described in greater detail later in this thesis.

Virtual Drivers in history

The Virtual Driver one, is not a new concept in history, but with substantial differences with respect to the definition given in the previous paragraph. In the past, the primary goal of modeling a virtual driver was to analyze traffic flow, road accidents, and study human behavior in traffic scenarios. Over time, additional models were developed to emulate human driving behavior more closely. After several years of research, it has been established that driver behavior can be subdivided into three hierarchical levels [5], as depicted in Figure 1.2, where:

Strategical level is the highest level of the hierarchy, responsible for trip planning. It involves selecting the route to reach the final destination and scheduling the time needed to complete the journey. The choice of path is influenced by factors

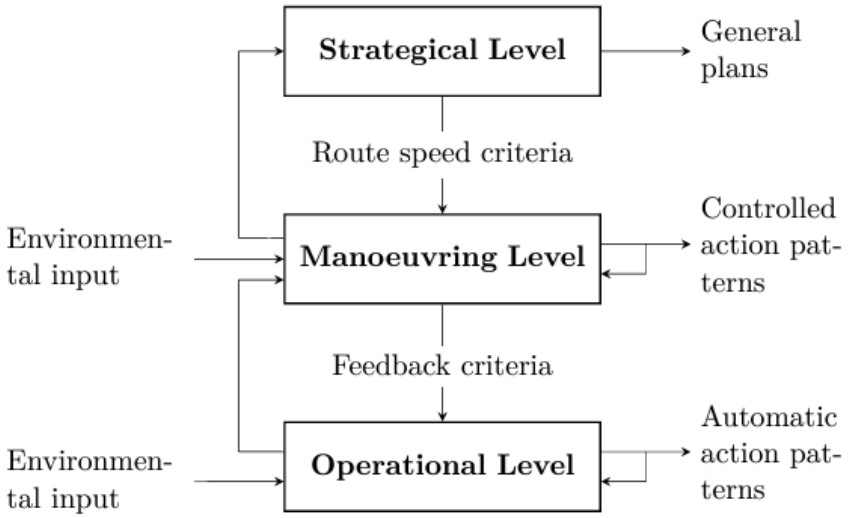

Figure 1.2: Hierarchical structure of the road user task. Source: [5]

such as road availability, obstacles, the current state of infrastructure, and traffic conditions.

Tactical level is subordinate to the strategic level and is responsible for making decisions based on environmental conditions. It considers the vehicle's relative position to infrastructure and other vehicles, adapting the driving strategy accordingly.

Operational level is the deepest level in the hierarchy, responsible for executing specific driving maneuvers. This level translates decisions made at the tactical level into concrete actions, such as steering, accelerating, and braking.

Maserati's previous efforts

Recent advancements in technology, particularly in Machine Learning and Deep Learning, have enabled the development of data-driven approaches to virtual drivers that mimic human behavior more accurately [6], [7]. NNs can be used in every level of the hierarchical three, from the trip planning depending to the traffic conditions till be used to approximate the typical driver's throttle-and-brake actuation depending on the situations, this aims creating virtual drivers that behave similarly to human drivers in various scenarios and conditions. It is worth noting that, once developed, these general-purpose drivers can perform maneuvers with a complexity limited only

by the *Model Capacity* [1] of the NN employed and the *quality* of the data used for training, which always relies on the mantra *garbage in, garbage out*.

In previous efforts, Maserati also attempted to develop data-driven virtual drivers for use in the HiL testing environment, by integrating the model in the same manner as discussed in the previous section. While the results were promising, limitations remained in the applicability of this method for automatic testing purposes. These virtual drivers provided a good approximation of average driver behavior when placed in the appropriate scenario, enhancing the HiL testing environment by introducing human-like behaviors, both in operational and strategical levels [10], that automated testers could not replicate.

However, they still fell short in fully automating the testing process and covering all possible test cases through multiple iterations of the virtual driver, still leaving out a lot of work to engineers in designing test patterns.

One significant challenge was the difficulty in extracting relevant data sequences from the vast hours of recorded on-road testing conducted by human drivers acting as testers. Identifying the exact testing sequences needed for training the NN proved nearly impossible, meaning the data-driven network could not be exclusively trained on pure testing scenarios. This represented a challenging problem that Maserati's team was unable to fully resolve. For this reason the behaviour of such systems was to act like a common driver rather than a *Test* Driver.

1.2 Neural Networks and Reinforcement Learning

This section focuses on providing fundamental knowledge in the field of Deep Learning, with particular emphasis on the Reinforcement Learning approach. It highlights both the strengths and weaknesses of this method, while also outlining the key motivations behind its selection for this thesis. This knowledge will be useful later, when introducing the contributions of this work.

1.2.1 Introduction to RL

Reinforcement Learning (RL) is a major branch of artificial intelligence (AI) and machine learning that focuses on learning the proper control laws or policies through

[1] The Model Capacity of a Neural Network may be informally defined as the "power" of a model, its ability to approximate accurately complex hidden functions.

interaction with a complex environment, learning by experience. Thus, RL is situated at the growing intersection of control theory and machine learning, and it is among the most promising fields of research towards generalized artificial intelligence and autonomy[8]. Unlike other forms of machine learning, such as supervised learning, where a model is trained on labeled datasets, in Reinforcement Learning, an agent learns to make optimal decisions by exploring an environment and receiving feedback in the form of rewards or penalties.

The main goal of RL is to learn an effective control strategy or set of actions through positive or negative reinforcement. The agent takes actions and receives rewards (or penalties) as feedback from the environment, which gradually guides it to modify its behavior. This learning approach is based on trial and error: the agent experiments with different actions and, by observing the consequences of these actions on the environment, learns to choose those that lead to greater rewards. In this way, RL is fundamentally biologically inspired, mimicking how animals learn to interact with their environment through positive and negative reward feedback from trial-and-error experience. In addition to the immediate reward, the agent must also consider future rewards, as its objective is to maximize the cumulative reward by the end of the simulation. This may involve taking actions that do not provide the highest immediate reward but, when combined in a strategic sequence, can lead to the maximum overall reward. To achieve this, the agent requires a form of planning and optimization, as it must balance short-term actions with long-term benefits.

Figure 1.3 provides a schematic overview of the RL framework, in which the key elements can be identified in:

1. Agent: The entity that learns and makes decisions, aiming to maximize long-term rewards.

2. Environment: The context or world in which the agent operates, providing feedback in the form of states and rewards after each action.

3. Actions (a): The possible decisions or moves the agent can take, which affect the state of the environment.

4. States(s): Representations of the current situation, describing the world as the agent perceives it, and changing based on the agent's actions.

5. Reward(r): A numerical value given to the agent as feedback for its actions, with the goal being to maximize the cumulative reward over time.

What sets RL apart from other machine learning paradigms is that the agent does not know in advance which actions will lead to the highest rewards. Instead, through repeated interaction with the environment, it gradually builds a strategy (or policy) that enables it to select the most effective actions in any given situation.

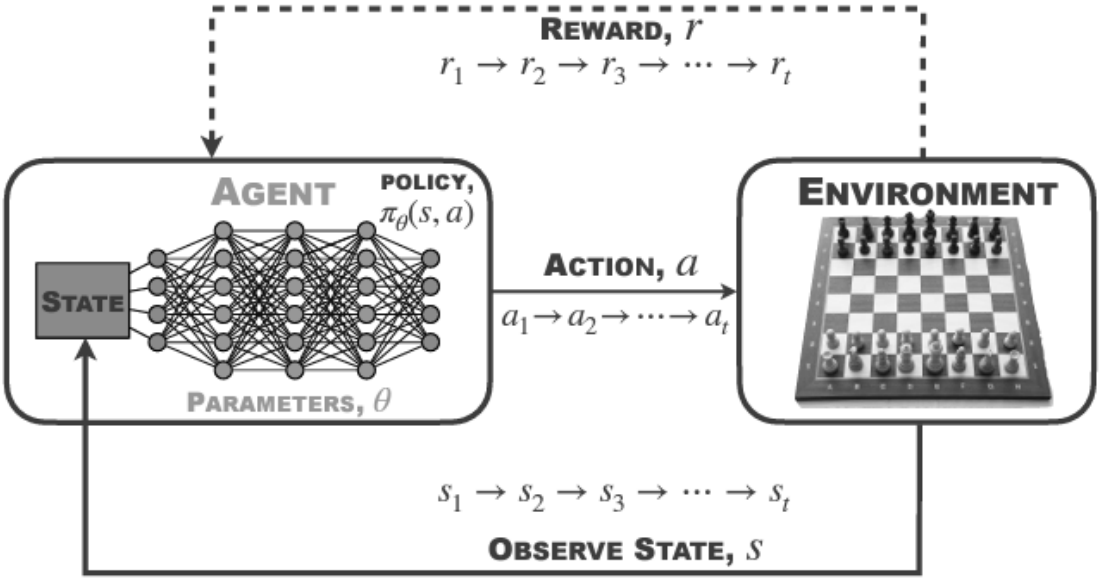

Figure 1.3: Schematic of Reinforcement Learning framework. Source: [8]

In the scenario depicted in Figure 1.3 the agent is attempting to learn how to play chess. The environment consists of the chessboard, including the positions of all the pieces. The actions are represented by the various moves the agent can make with each piece, while the states reflect the configuration of the board after each move. Since the state of the game changes dynamically after every move made by either player, the number of possible states corresponds to the number of potential game moves.

For the agent to learn effectively, it must continuously update its policy $\pi_\theta(s, a)$ where it selects actions based on the current state, with the goal of maximizing its cumulative reward by the end of the game[1].

1.2.2 Main differences with Data Driven

Before delving into the mathematics of Reinforcement Learning, it is essential for the reader to fully grasp the key differences that distinguish the RL approach from the more commonly known data-driven approach. Understanding this contrast will be crucial for following the architecture developed for training, shown later in this thesis.

[1]Typically in learning chess the reward is given at the end of the game since is difficult to say for each move if it was the best one or any grade of correctness.

While many readers may be more familiar with the data-driven approach—given its widespread use and prominence in recent years, as well as its frequent coverage in university courses—RL introduces a fundamentally different paradigm that requires a distinct mindset.

The data-driven and Reinforcement Learning approaches differ fundamentally in how they handle data, learn, and interact with the environment. Data-driven approaches, such as supervised learning and traditional machine learning, rely heavily on pre-labeled datasets to train models that can generalize to new data. In this framework, algorithms are trained on a fixed set of examples, where each instance consists of input features and a corresponding output label. The primary objective is to minimize prediction error, and the model's performance depends largely on the quality and quantity of available data. Consequently, data collection, cleaning, and representation become critical components of the process.

In contrast, Reinforcement Learning operates under a completely different paradigm. Instead of relying on labeled data, an RL agent learns through direct interaction with its environment, exploring various actions and observing the resulting outcomes. The agent is not explicitly told which actions are correct; rather, it learns the optimal strategy through a process of trial and error. Feedback comes in the form of rewards or penalties, which help guide the agent towards maximizing its long-term cumulative reward. This approach is particularly well-suited for sequential decision-making problems, where actions must be chosen based on their potential long-term effects.

Another key distinction lies in how models evolve. In traditional data-driven approaches, once a model is trained, it remains static and cannot improve unless new training data is introduced. In RL, however, the agent continuously learns during its interaction with the environment, adapting its strategy over time. This makes RL particularly advantageous in dynamic and uncertain environments, where real-time decision-making is crucial, such as in autonomous systems or video games. The ability to continuously refine behavior without requiring new, labeled data highlights RL's strength in scenarios where adaptability is essential.

1.2.3 Mathematical formulation

As mentioned earlier, an RL agent senses the state of its environment and learns to take appropriate actions to achieve optimal immediate or delayed rewards. The agent transitions through a sequence of different states $s_k \in S$ by performing action $a_k \in A$ with the selected actions leading to positive or negative rewards r_k which are used for learning. Here S and A represent the sets of possible *states* and *actions*, respectively.

Reinforcement learning is typically formulated as an optimization problem, where the goal is to learn a **policy** $\pi(s, a)$, that maximizes the total future rewards, the policy represents the probability of taking action a given state s and is expressed as:

$$\pi(s, a) = Pr(a = a \,|\, s = s) \tag{1.1}$$

However, for most real-world problems, directly representing and learning this policy can be computationally prohibitive. To address this, the policy π is often represented as an approximate function parameterized by a lower-dimensional vector θ:

$$\pi(s, a) \;=\; \pi(s, a, \theta) \tag{1.2}$$

which can be also denoted as $\pi_\theta(s, a)$. Is worth noting that the function approximation is the basis behind the deep reinforcement learning, so this is the notation that is going to be met more frequently in this elaborate.

In general, the measured state of the system may be a partial measurement of a higher-dimensional environmental state that evolves according to a stochastic, nonlinear dynamical system. However, for simplicity, most introductions to RL assume that the state evolves according to a **Markov decision process (MDP)**. The key property of an MDP is the *Markov's property* which says that the probability of the system occurring in the current state is determined only by the previous state without being influenced by the actions previously taken and the states before.

An MDP consists of:

- A set of states S, a set of actions A, and a set of rewards R, along with the probability of transitioning from state s_k at time t_k to state s_{k+1} at time t_{k+1} given action a_k,

$$P(s', s, a) = Pr(s_{k+1} = s' \,|\, s_k = s, a_k = a). \tag{1.3}$$

- A reward function

$$R(s', s, a) = Pr(r_{k+1} \,|\, s_{k+1} = s', s_k = s, a_k = a). \tag{1.4}$$

This said, for an MDP, given a policy π, the transition process can be written as

$$s' = \sum_{a \in A} \pi(s, a)\, T_a s \tag{1.5}$$

where for each action a, the MDP can be seen as a simple Markov process in which $P(s', s)$ can be written in terms of a transition matrix, also known as a stochastic matrix, or a probability matrix, T_a with all columns summing to 1.

The *Bellman equation* is a fundamental recursive relationship in Reinforcement Learning that expresses the value of a state in terms of the immediate rewards and the expected value of future states. The desirability of being in a certain state, given the policy π, can be quantified by a **value function** such that:

$$V_\pi(s) = \mathbb{E}(\sum_k \gamma^k r_k \,|\, s_0 = s) \tag{1.6}$$

where $\mathbb{E}$ is the expected reward over the time steps k, subject to a discount rate γ. The meaning in having a discounted future reward is to reflect the economic principle that current rewards are more valuable than future rewards. An important property of the value function is that the value at a state s may be written recursively as

$$V(s) = \max_\pi \mathbb{E}(r_0 + \sum_{k=1}^{\infty} \gamma^k r_k \,|\, s_1 = s') \tag{1.7}$$

which implies that

$$V(s) = \max_\pi \mathbb{E}(r_0 + \gamma V(s')) \tag{1.8}$$

where $s_0 = s_k + 1$ is the next state after $s = s_k$ given action a_k, and the expectation is over actions selected from the optimal policy π. This expression is the Bellman's equation, a statement of Bellman's principle of optimality, and it is a central result that underpins modern RL. From the last equation is possible to extract the optimal policy as:

$$\pi = \operatorname{argmax}_\pi \mathbb{E}(r_0 + \gamma V(s')). \tag{1.9}$$

Learning the policy π the value function V, or both jointly is the central challenge in Reinforcement Learning. Depending on the structure of π, the size and dynamics of the state space S, and the reward landscape R, determining an optimal policy can range from a simple closed-form optimization to a highly complex, high-dimensional unstructured optimization. As a result, a large number of trials are often required to evaluate and identify the optimal policy.

In practice, training a reinforcement learning model can be computationally expensive, and may not always be the ideal strategy for problems where testing a policy is costly or potentially unsafe. In such cases, alternative approaches might be more suitable.

1.2.4 Brief categorization of RL

There are various approaches to learning an optimal policy π, which is the ultimate goal of Reinforcement Learning. One of the key distinctions in RL is between *model-*

based and *model-free* approaches.

Reinforcement Learning techniques can be categorized based on how the agent learns and uses information about the environment, as depicted in Figure 1.4:

1. Model-Free vs. Model-Based
    - Model-Free: The agent learns directly from interaction with the environment without building an explicit model of the environment. Examples: Q-Learning, SARSA.
    - Model-Based: The agent learns or builds a model of the environment to predict state transitions and rewards, and uses this model to plan its actions. Example: Dyna-Q.
2. On-Policy vs. Off-Policy
    - On-Policy: The agent learns the policy while following the same policy that it is trying to improve. An example is SARSA.
    - Off-Policy: The agent learns a policy different from the one used during interactions with the environment. An example is Q-Learning, where the agent learns the optimal policy independent of the one used for exploration.
3. Value-Based vs. Policy-Based
    - The agent learns a value function that estimates how beneficial it is to be in a particular state or take a specific action, and derives the policy from this value function. Examples include Q-Learning and Deep Q-Networks (DQN).
    - The agent directly learns a policy, which maps states to actions, without the need to construct a value function. An example is the Policy Gradient Methods.
4. Actor-Critic
    - Actor-Critic: A hybrid approach where the agent consists of two components: the actor, which learns the policy, and the critic, which learns a value function to evaluate the actions proposed by the actor. An example of this approach is the Deep Deterministic Policy Gradient (DDPG).

When a model of the environment is available, strategies such as policy iteration or value iteration—forms of dynamic programming that use the Bellman equation—can be employed to learn the optimal policy or value function. In the absence of a model, alternative strategies such as Q-learning must be used.

Reinforcement learning becomes particularly challenging in high-dimensional systems with unknown, nonlinear, stochastic dynamics and sparse or delayed rewards. Many of these techniques can be combined with function approximation methods, such as **neural networks**, to approximate the policy π or the value function V.

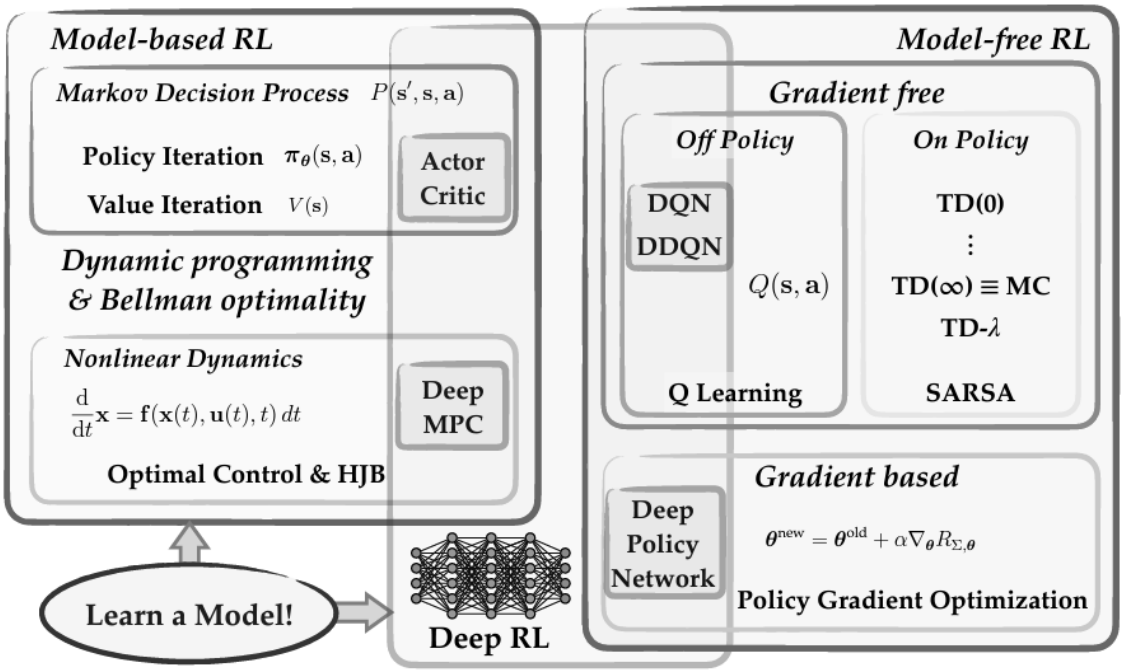

Figure 1.4: Categorization of RL techniques. Source: [8]

1.2.5 Neural Networks in RL

Classic reinforcement learning faces a significant representation problem, as many of the key functions—such as the policy π, the value function V, and the quality function Q,—can be exceedingly complex and defined over very high-dimensional state and action spaces. The complexity of these functions makes it practically impossible to represent them exactly without approximation. For instance, even a simple game like Pong, which has a black and white screen with a resolution of 336x240 pixels, contains over $10^{24'000}$ possible discrete states. This makes it infeasible to represent any of these functions exactly in a conventional manner.

Deep learning offers a powerful solution to this problem by providing tools for function approximation. Specifically, Neural Networks (NN) can serve as general

approximators for complex functions. An overview on NN is provided in Appendix A for the interested lector. In the context of reinforcement learning, the policy can now be approximated as:

$$\pi(s, a) \approx \pi(s, a, \theta) \tag{1.10}$$

where θ represents the weights of a neural network. This combination of deep learning for function representation and reinforcement learning for decision-making and control has led to significant advancements in the capabilities of reinforcement learning systems.

The way deep learning changed the way in doing reinforcement learning can be summarized by looking at these two examples in relation with the categorization shown in Subsection 1.2.4.

- In traditional Q-Learning, the Q-function (which represents the value of a state-action pair) is stored in a table. However, when the state space is large or continuous, using a table becomes impractical. Neural networks are then used to approximate the Q-function, as in Deep Q-Networks (DQN), where a neural network takes a state as input and outputs Q-values for all possible actions.
- In the Actor-Critic method, two neural networks are used together: one network, the actor, learns the policy (i.e., which action to take in each state), while the other network, the critic, learns a value function to evaluate the actions taken by the actor. Both networks are updated simultaneously to improve the agent's performance.

This synergy between deep learning and reinforcement learning opens up new possibilities for developing autonomous systems capable of operating in dynamic, high-dimensional environments, such as those found in automotive testing and other real-world applications.

1.2.6 Actual RL applications

In recent years, Reinforcement Learning has found numerous practical applications across various industries, due to its ability to tackle complex decision-making and control problems. One of the most well-known sectors for RL applications is gaming, with notable examples such as AlphaGo [11]—a neural network trained to play Go—and RL systems that excel at Atari games, often surpassing human performances [12]. These systems have demonstrated the power of RL by outperforming human champions in highly complex games like Go, chess, and shogi. RL algorithms enable these systems to learn optimal strategies through billions of game simulations, without any prior human knowledge. In recent years, commercial video games have

also integrated RL to create smarter non-playable characters (NPCs) that can adapt to player behavior in real time.

Another area where RL is making a significant impact is robotics [13]. Deep Reinforcement Learning techniques are used to train robots to perform complex tasks such as object manipulation, walking, and navigating dynamic environments. RL allows robots to adapt to new situations without the need for explicit programming for each task, making it a powerful tool for autonomous systems.

Moreover, RL is increasingly being applied in fields such as finance, healthcare, and energy management, where decision-making and optimization are critical.

1.3 Contribution

For the reasons outlined in Sections 1.1.4, this thesis adopts a completely different approach, utilizing a cutting-edge technology: neural networks trained through reinforcement learning (RL). The resulting *agent* is integrated into the HiL testing environment and is tasked with navigating the vehicle through the entire state space, progressively validating all possible test cases over multiple iterations. The idea is to integrate the developed agent within the HiL testing environment, allowing the agent generate new **reference signals** at each time step. These reference signals dynamically modify the vehicle's state, with low-level controllers ensuring that the actual vehicle state follows the reference signals (see Figure 1.5). This aims to simulate various driver actions such as accelerating to a specific speed or engaging an ADAS by pressing a button. As a result, the cost of generating test cases is significantly reduced, saving countless hours for test engineers. Moreover, the created tests are fully repeatable and reliable, ensuring consistency in validation efforts.

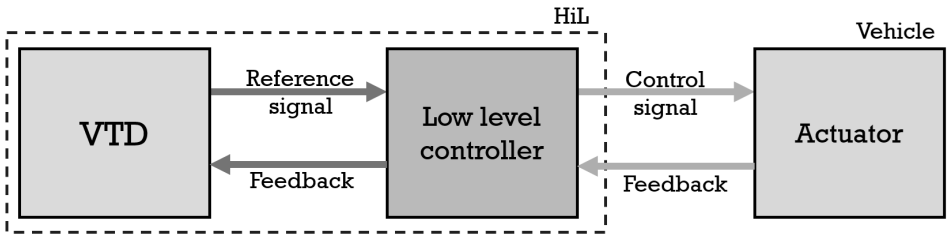

Figure 1.5: VTD as reference generator.

By repeating these validations with varying state trajectories and starting conditions, it becomes possible to sift through all potential combinations in search of

bugs or malfunctions during the interactions between the systems under test. It is important to note that the developed VTD does not directly control actuators like the brake or throttle pedal. Instead, it sets a set point that the automatic controls in the HiL environment use as a reference to adjust and follow by the use of the automatic controls already present onboard. For continuous signals such as brake pressure, the system pursues the new reference accordingly, while for discrete actions like pressing a button, the system directly interacts with the control.

This work was made possible thanks to the support of Pasquale Sessa, leader of Maserati's Test Team, and Professor Nicola Mimmo, who generously provided their time and resources. From May to September 2024, I was associated with Maserati's Testing and Validation department for the development of a system that meets the company requirements. This thesis is structured around the workflow followed during the development of the virtual test driver.

Chapter 2

Proposed Methodology

This chapter is dedicated to formalizing the problem that the developed work seeks to address. It outlines the current testing methodologies in use, highlights the aspects the proposed system aims to improve, and delves into the thought process behind the development of this thesis.

2.1 Problem formulation

2.1.1 General test flow

Every testing procedure begins when the test manager receives the document, *test paper* outlining the specific functionalities to be tested, along with specifications that provide a deeper understanding of the test objectives. The specific task is then assigned to a test engineer, who is responsible for designing an appropriate test routine composed of various test cases, aiming to achieve as complete a coverage as possible of the scenarios in which the specific functionality is expected to operate.

The test engineer must create multiple *test cases* to evaluate the component or functionality, not only in standard operational conditions but also in borderline scenarios—within the realm of "possible"—where software bugs or other malfunctions are more likely to emerge. Often, a single functionality does not rely on just one ECU, but rather a set of interconnected ECUs, as all the ADAS in a fully developed car must function in unison. As a result, the designed tests may involve a large number of variables that must interact seamlessly.

Once the test cases are designed, they are applied in the specific *test environment* (Section 1.1.3). This involves creating scripts or procedures to execute the test cases. The final step in the testing process is to verify the outputs of the ECUs and the overall system behavior by comparing them with the expected results. This is essential to ensure that there are no issues that could lead to a loss of functionality

when the vehicle is used by the end customer.

In the case of a software bug discovered during testing, it must be reported to the relevant department or supplier, along with a detailed description of the problem and the environmental conditions that led to its occurrence. The bug is then analyzed, and a new software version is released. The testing procedure is repeated to verify that the bug has been resolved and that no new issues have been introduced. The general flow of the testing process is depicted in Figure 2.1.

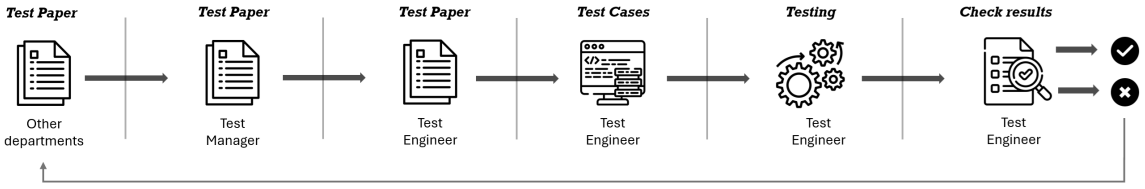

Figure 2.1: Flowchart of test development.

How it can be deducted, the time spent by test engineers in developing tests is substantial. The process requires a deep understanding of the main goals of the test, careful design and implementation, and finally, a manual check of the output signals; always remembering that the testing process, like any other, is susceptible to human error, which must always be taken into account. Additionally, test development is highly time-consuming, and engineers often have limited time to achieve the necessary threshold of test coverage—remembering the principle that *time is money*-so the number of test cases that can be designed and submitted to the system is limited and must be accurately chosen.

For this reason, the company is actively seeking a system that can reduce the workload of test engineers by automating the design of test cases and allowing for more tests to be run in the same period of time. This would enable engineers to focus more on analyzing the results of the simulations rather than spending time on designing and executing the tests themselves. Such a system would increase efficiency, reduce human error, and help ensure more comprehensive test coverage within shorter time frames.

2.1.2 Case study

This thesis aims to find a suitable, cost-effective, and scalable solution to address the company's needs outlined earlier. To achieve this, it was necessary to select a comprehensive case study that would allow the development of an approach that is fully integrated and scalable. From this point forward, when referring to the *case*

study, we mean the implementation of the ADAS - Hill Descent Control (HDC) within the Hardware-in-the-Loop (HiL) test environment.

The working principle of the HDC can be described as follows: "Once the driver has activated hill descent control and chosen a maximum speed, the vehicle draws on its traction-control and anti-lock braking systems to minimize tire slip and ease you down the slope."[14]

Testing Environment

The HiL environment used during the thesis internship consists of several interconnected components that allowed for the simulation of the entire vehicle's functionalities in a virtual setting. This approach is significantly cheaper and safer than conducting advanced tests on the road with a fully operational vehicle, particularly when multiple ECUs are involved. HiL testing enables the validation of complex systems under controlled conditions, reducing risks and costs while offering comprehensive coverage of vehicle functionalities.

The test environment consists of a test bench that houses the entire electrical system of the vehicle, along with additional tools to facilitate testing. A schematic representation of this environment is shown in Figure 2.2, which includes the following components:

- **The Vehicle Environment**: This encompasses the Electronic Control Units (ECUs), sensor suite, and actuators. Some actuators may be simulated, while others, such as the Virtual Human-Machine Interfaces or headlights, are fully implemented.

- **dSPACE SCALEXIO**: A hardware-in-the-loop simulation platform used for testing and validating complex embedded systems. This platform provides a flexible and scalable environment for simulating the behavior of mechatronic systems. It supports the integration of virtual models of vehicle components with real hardware, allowing the system's responses to be tested in real-world scenarios such as adverse driving conditions or hardware failures. It is particularly useful for testing ADAS by simulating various driving scenarios, weather conditions, or obstacles the system might encounter. It is used to refer to it simply as *dSpace*.

- **The Control Desk**: This interface allows engineers to monitor and control a wide range of variables during simulations, as well as initialize them at the start of the test. It serves as a crucial tool for real-time analysis.

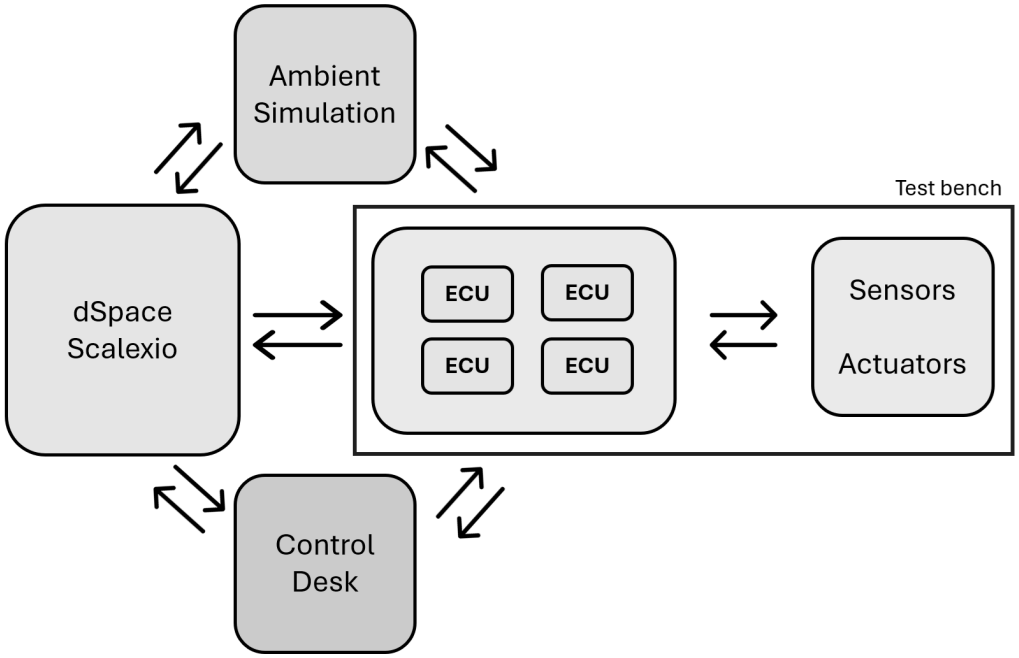

Figure 2.2: Test Environment.

- **Ambient Simulation**: This is the external visualization (displayed on a PC monitor) of the virtual environment in which the car operates. It is used by test engineers and the vehicle's sensors for various purposes. For example, ADAS systems can use this simulation to recognize speed limit signs and automatically adjust the vehicle's maximum speed. The virtual environment is highly customizable, allowing test engineers to design it according to the specific requirements of the test. Using specialized software, engineers can create a wide range of environments, from expansive parking lots to narrow mountain roads, depending on the scenario being tested. Additionally, it is possible to introduce other vehicles into the simulation or create dynamic events, such as a deer crossing the road or other unexpected obstacles, to test how the system reacts in real-world-like situations.

Almost all communications within the described test environment are conducted via Controller Area Network (CAN), Local Interconnect Network (LIN), or Ethernet, which are the most commonly used communication protocols in the automotive industry. These protocols are essential because they offer reliable and standardized communication, ensuring correct real-time data transmission. CAN is the most widely used protocol, as it is specifically designed for high-priority communications

with low latency, enabling real-time data exchange between ECUs on the same bus. LIN, while slower (20 kbps) than CAN, is simpler and more cost-effective. It adopts a master-slave architecture and is typically used for systems that do not require high data transmission speeds, such as interior lighting or door controls. Ethernet offers much higher bandwidth than both CAN and LIN, making it ideal for transmitting large volumes of data, such as from cameras, radar, or advanced ADAS systems. In many modern vehicle architectures, Ethernet is increasingly being used alongside or as a replacement for CAN in applications that demand high-speed, high-reliability communication, such as infotainment systems and autonomous vehicles.

Test Design

In recent times, most tests are conducted using a *Script-Based Testing* approach. The test engineer is responsible for designing a sequence of actions that the vehicle must go through. During the test, the vehicle passes through various states, meaning that certain variables must change dynamically. The test engineer must carefully script these state changes to ensure the vehicle is tested properly.

The complexity of this approach can quickly grow when dealing with functionalities that involve multiple *Factors* and *Levels* (Section 1.1.4). Recalling the HDC case study, the factors and levels were defined as follows, and are summarized in Table 2.1:

Longitudinal Speed The vehicle must adopt ten different speed levels to conduct a complete test.

HDC button Simulates the pressing or releasing of the HDC activation button on the vehicle's HMI[1], with two states: ON and OFF.

ACC button Simulates the pressing or releasing of the Adaptive Cruise Control (ACC) activation button on the HMI, with two states: ON and OFF.

EPB button Simulates the pressing or releasing of the Electric Parking Brake (EPB) activation button on the HMI, with two states: ON and OFF.

Speed Limiter button Simulates the pressing or releasing of the Speed Limiter activation button on the HMI, with two states: ON and OFF.

It is clear that creating each individual test case requires a significant amount of time, and there is no guarantee that it will perform as expected. Furthermore, vehicle state changes must be validated in both directions—forward and backward—for every

[1]The Human Machine Interface (HMI) can include both real buttons on the dashboard and steering wheel and simulated buttons on the vehicle's internal touch screens.

Factors	Number of *Levels*	*Levels*
Longitudinal Speed [km/h]	10	[5,10,15,20,25,30,35,40,45,50]
HDC button [-]	2	[0,1]
ACC button [-]	2	[0,1]
EPB button [-]	2	[0,1]
Speed Limiter button [-]	2	[0,1]

Table 2.1: *Factors* and *Levels* for HDC testing.

factor. Despite these challenges, Script-Based Testing is a feasible approach and is well-understood by test engineers, who, through their expertise, are able to achieve the necessary test coverage within the given time constraints.

To assist engineers in selecting the most critical combinations of factors and levels, a coverage matrix is typically created. This matrix is often a subset of the full matrix generated by all possible combinations of Factors and Levels, as manually writing test steps for every possible scenario can be time-consuming and complex. As a result, the coverage matrix usually consists of around 6×6 combinations, resulting in 36 test cases. By narrowing the focus to the most relevant test scenarios, this matrix helps streamline the testing process, ensuring more efficient use of time and resources.

2.2 Proposed Solution

The aim of this thesis is to eliminate the need for test engineers to manually create test cases through Script-Based Testing for submission to the HiL environment. To achieve this, a Neural Network-based system, referred to as the (VTD), has been developed. This system automatically generates the values corresponding to the Factors and Levels at each time step of the simulation, effectively creating test cases that dynamically evolve throughout the simulation. This process occurs without human intervention, allowing the test engineer to focus solely on monitoring the expected outputs from the ECUs and reporting any bugs that arise. This would increase the testing coverage and a better management of the given time for functional tests.

2.2.1 State space

One of the core ideas behind this project is the use of defined Factors and Levels to create a **State Space**, intended as a framework that encompasses all the possible states the vehicle can assume in order to correctly validate a specified functionality. The state space is constructed as follows: assuming a simplified and non-realistic model, with the Factors and Levels outlined in Table 2.2. In this example, there are three Factors, which determine the number of **dimensions** in the state space, and each Factor has three Levels, which define the **length** of each dimension.

As a result, the state space consists of $3 \times 3 \times 3 = 27$ distinct state positions, representing all the different conditions in which the vehicle can operate and should be tested. This simplified example may not fully illustrate the advantages of this

Factors	Number of *Levels*	*Levels*
Longitudinal Speed [km/h]	3	[10,20,30]
Lateral Position [m]	3	[-2.5,0,2,5]
Drive Mode Selector [-]	3	[0,1,2]

Table 2.2: Example of *Factors* and *Levels*.

approach in terms of vehicle state exploration. However, when applied to the case study—where the Factors and Levels are described in Table 2.1—the resulting state space expands to five dimensions, yielding a total of 160 possible states. This represents a significant improvement compared to the achievable 6×6 matrix typically produced with Script-Based Testing, allowing for much more comprehensive testing and validation of the vehicle's behavior. The concept of state space will be also treated during the practical implementation part of this work (Chapter 3), where it is going to be referred as *m2*.

2.2.2 Trajectories

Based on the assumptions outlined earlier, the developed system—the Virtual Test Driver (VTD) —should be capable of efficiently guiding the vehicle through all possible states in a meaningful manner. The system must be integrated into the existing simulation model, as illustrated in Figure 2.3 [1], so that at each time step of the simulation, the controlled variables are updated and the vehicle environment adjusts accordingly.

[1]The VTD is here indicated as 'Agent' for reasons that will be outlined shortly.

The output of the VTD is expected to be a sequence of arrays that specify the next state space location the vehicle should move to. An example of the VTD's output, for the simplified case study described before, is shown in Table 2.3.

t_0	t_1	t_2	...	t_n
$[10, 0, 0]$	$[10, -2.5, 0]$	$[10, -2.5, 2]$	...	$[30, 0, 2]$

Table 2.3: Example of VTD output sequence.

It is worth noting the two main scenarios in which the vehicle can adapt to the incoming state changing signals:

- For two-state buttons or multi-state switches, the system simply sends a CAN message to the appropriate ECU, specifying the overwrite of the relevant variable.
- Continuous variables—such as longitudinal speed, lateral position, and similar parameters— have already been discretized (*Levels*). The vehicle is then guided to the correct state by the use of automatic controllers, which output continuous signals like steering wheel angle or pedal travel percentage, receiving as input the updated state which indicates the target to reach. The output of the automatic controller is then sent as CAN message to the appropriate ECU.

Random Trajectory

The company specified the desired system characteristics in a way that ruled out the possibility of using a **pseudo-random number generator** to produce random arrays of states, which would result in the VTD generating a trajectory in the state space, such as the one shown in Figure 2.4[1]. Such an approach would produce outputs that are completely uncorrelated with each other, making it unsuitable for the intended purposes.

As can be easily guessed, producing arrays composed by random numbers, the pseudo-random number generator outputs consecutive arrays whit an always different *Euclidean distance* between them. Euclidean Distance is a key concept in this thesis and must be comprehended before moving on.

It measures the straight-line distance between two points in Euclidean space, whether two-dimensional, three-dimensional, or higher-dimensional. The Euclidean

[1]For a matter of ease in visualization is shown the general case of a $5 \times 5 \times 5$ state space.

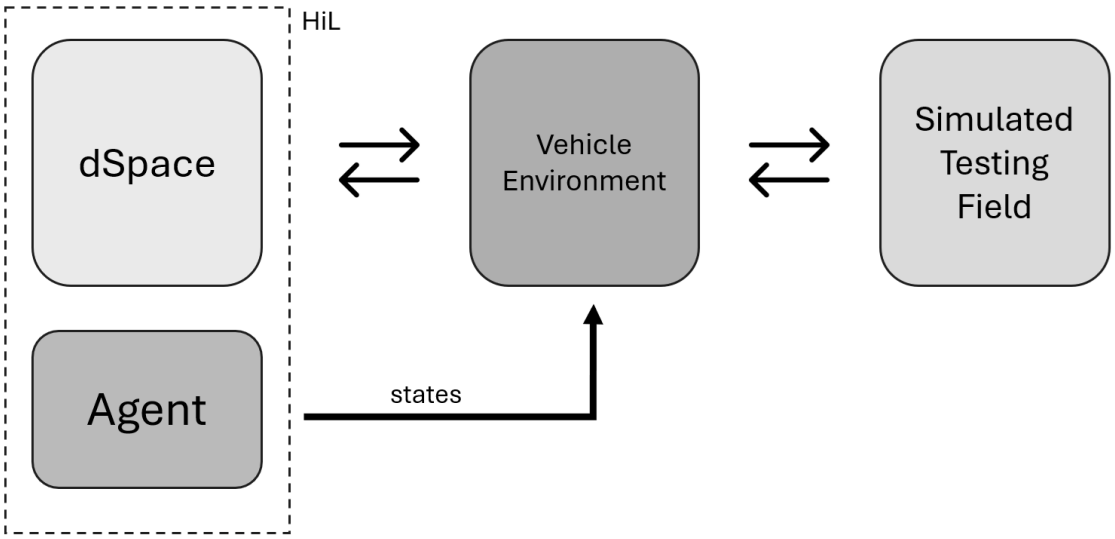

Figure 2.3: Agent Integration in HiL environment.

Distance between two points, $P_1 = (x_1, x_2, x_3, \ldots, x_N)$ and $P_2 = (y_1, y_2, y_3, \ldots, y_N)$, in an $N-dimensional$ space is calculated using the Pythagorean theorem as follows:

$$d = \sqrt{\sum_{n=1}^{N} (x_n - y_n)^2}, \tag{2.1}$$

This measure helps determine how far apart two consecutively generated states are.

Given the assumption of a uniform probability distribution for each number within the span provided to the pseudo-random number generator, the mean Euclidean distance between two consecutive generated states can indeed be approximated by considering the distance between a corner point of a matrix and its center. This provides a measure of how far a random state is expected to be from the initial state.

In the example of a $5 \times 5 \times 5$ matrix, starting from the corner point [1,1,1], is possible to compute the mean Euclidean distance by finding the distance between this corner point and the center of the matrix. The coordinates of the middle point are calculated by taking the mean of the maximum and minimum indices for each dimension:

$$meanIndex_i = \frac{max_i - min_i}{2} + min_i, \tag{2.2}$$

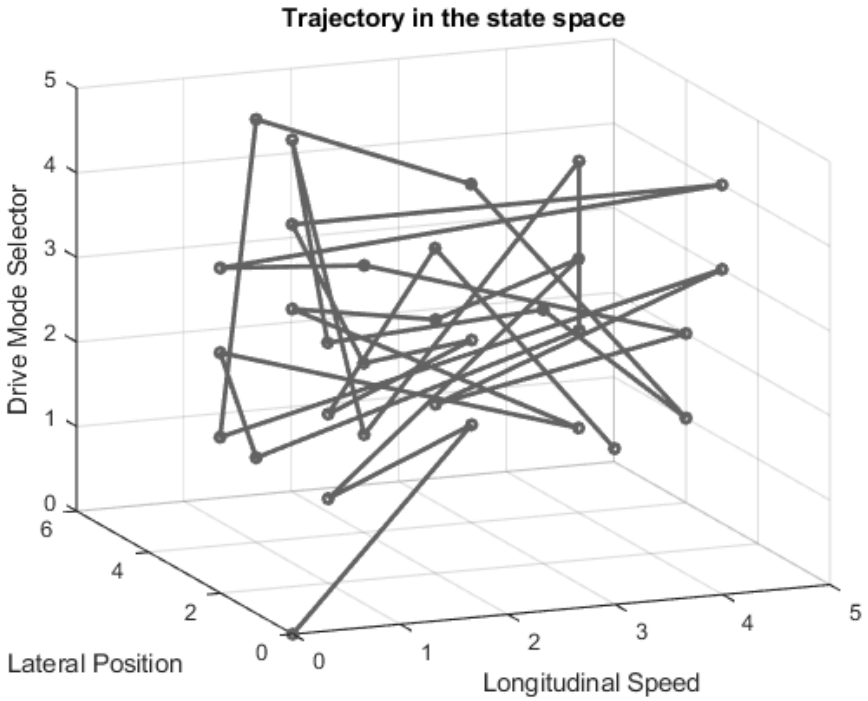

Figure 2.4: Trajectory from a random number generator.

For each dimension i in a $5 \times 5 \times 5$ matrix, this gives $meanIndex_i = 3$, resulting in the center point [3,3,3].

The Euclidean distance between the corner point [1,1,1] and the center point [3,3,3] is given by the standard Euclidean distance formula (equation 2.1):

$$d_{mean} = \sqrt{(3-1)^2 + (3-1)^2 + (3-1)^2} = \sqrt{12} \approx 3.46. \tag{2.3}$$

Thus, the mean Euclidean distance between two randomly generated consecutive states, assuming uniform distribution, is approximately $d_{mean} = 3.46$. This provides a useful benchmark for evaluating the performance of the system relative to purely random state transitions.

Although implementing such a random approach would be simple, requiring just a few lines of code in MATLAB, it would be ineffective for testing purposes. The VTD is intended to simulate a more "feasible, human-like approach", and it would be unrealistic for a driver to press multiple buttons simultaneously while steering. While testing simultaneous button presses is part of the process, it should occur as an exception, not as the rule. Additionally, a random approach could lead to suboptimal results when controlling variables like longitudinal speed. If two consecutive outputs

from the VTD guide the vehicle to states that are too far apart in the same factor, the automatic controllers may not have enough time to reach the desired state before the next one is generated, resulting in incomplete coverage of the test case.

Contrary to what can thought assumptions, reproducibility of the test patterns is not an issue. This is because MATLAB allows programmers to specify a *random seed*. By specifying the seed and the same starting point, two consecutive runs of the same program generating random state arrays will produce identical output patterns and trajectories. This reproducibility is crucial for testing, as it allows the same test to be performed on different vehicles or under different environmental conditions. From this point forward, the problem of reproducibility will not be discussed further, as it will always be guaranteed in the developed systems.

MATLAB Tip

Fix the random seed with the following commands.

```
seed = 95;
rng(seed)
```

Single and Multi Stepper

On the opposite end of the spectrum, it is possible to implement a **Single-Stepper** approach, it is also based on a pseudo-random number generator, but uses the randomicity in a different way. Starting from an initial position, the system randomly selects a *Factor* at each time step and changes its state by just one *Level* (randomly deciding whether to increase or decrease it), the outlining state is then chosen if it was not chosen before in the simulation. The resulting array, which describes the next state, will differ from the previous one by only one Factor, which shifts by just one position (as shown in Figures 2.5 and 2.6). Consequently, the Euclidean Distance between each pair of consecutive states will always be one.

While this approach seems straightforward, it quickly becomes problematic to implement in MATLAB, especially when more Factors and Levels are involved. As the number of changes per time step increases, the program's complexity grows exponentially, leading to a higher risk of errors and bugs that could compromise the test results. For this reason, the **Multi-Stepper** approach, which involves changing multiple Factors and Levels at each step, has been discarded.

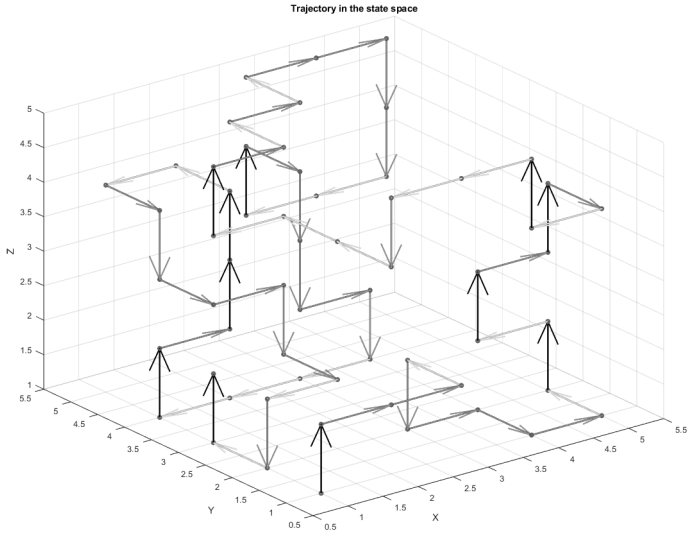

Figure 2.5: Trajectory from a Single Stepper.

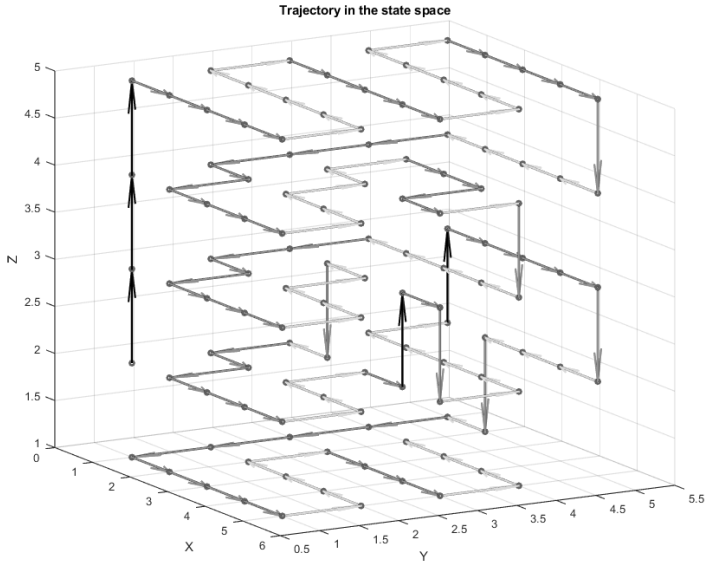

Figure 2.6: Trajectory from a Single Stepper.

2.2.3 Use of Reinforcement Learning

In light of the impossibility of using both the pseudo-random number generator, neither the single or multi-stepper, the outlined solution moves away from traditional logic and towards the use of Neural Networks. The objective was to develop a Neural Network-based Virtual Test Driver (VTD) capable of dynamically generating a new state array at each time step. The generated pattern would need to meet the following criteria:

1. A continuously changing target state at each time step.
2. A "feasible, human-like approach," where the number of state changes between consecutive time steps remains reasonable.
3. A varying Euclidean Distance between consecutive state arrays, ensuring that the distance is neither always one nor constant.

The initial idea was to train a Neural Network using a data-driven approach. However, this would have required the creation of a completely new dataset, as it would be fundamentally different from anything previously used by the Company. Building such a dataset from scratch would be time-consuming and resource-intensive.

Given these challenges, the data-driven approach seemed less effective compared to an alternative: Reinforcement Learning. Due to the lack of existing data and uncertainty about how to generate it, RL emerged as the most suitable method. As outlined in Section 1.2, with RL, the developed VTD could learn optimal behavior through trial-and-error interactions with a simulated environment. This eliminates the need for creating a dataset, making RL a more efficient and promising approach for the task.

Chapter 3

Development and Evaluation

This chapter outlines the methodologies used to develop the Virtual Test Driver, with a particular focus on the implementation of the Reinforcement Learning agents and the selection of the reward function. The chapter concludes with a detailed report on the performance of the VTD during testing.

3.1 Modeling phase

Due to company requirements, the implementation of the VTD was carried out using MATLAB and Simulink. This decision was straightforward, as the dSpace HiL system, mentioned in Section 2.2.2, works with pre-compiled Simulink code. The implemented VTD needed to be integrated alongside other Simulink components required for the proper functioning of the vehicle testing bench, making the choice of MATLAB and Simulink mandatory.

The training of the RL agent required both MATLAB and Simulink implementations, differing from the data-driven approach where only MATLAB was necessary during the training and deployment phases, the use of Simulink in data-driven applications is demanded when the neural network should be integrated with other systems. In contrast, RL development for HiL testing involves Simulink as well, as the agent must interact with an environment during training. This environment can either be entirely virtual, as in Simulink, or based on a real-world scenario, but this is not the case. From now on the case of an entirely virtual approach is considered.

In RL training, the process is divided into episodes rather than epochs, as is typical in data-driven approaches, but the underlying principle remains the same. Each episode runs for a finite number of time steps, during which the agent interacts

with the environment and receives rewards. The reward can be given after each action (usually at each time step), which is known as *dense rewarding*, or after a certain number of steps or specific events—this is called *sparse rewarding*. For example, in chess (seen in Section 1.2.1), the reward is calculated only at the end of the game, training an agent to play chess also requires, since a game can last for a mean of 80 moves, episodes which length will last in mean 40 time steps, meaning that the agent doesn't require an excessive number of steps within each episode to explore strategies and improve its performance. Instead, the challenge lies in the diversity of games and the vast state space, not the length of individual episodes[1].

At the end of each episode, the learning algorithm updates the weights and biases of the Neural Networks based on the total reward accumulated, enabling the agent to improve its behavior and learn the optimal policy through experimentation and feedback from the environment.

The following sections will delve into the process of developing the RL agent, which pipeline can be divided in two branches (Figure 3.1), one of the Simulink implementation and one for MATLAB implementation, since both are needed for training purposes.

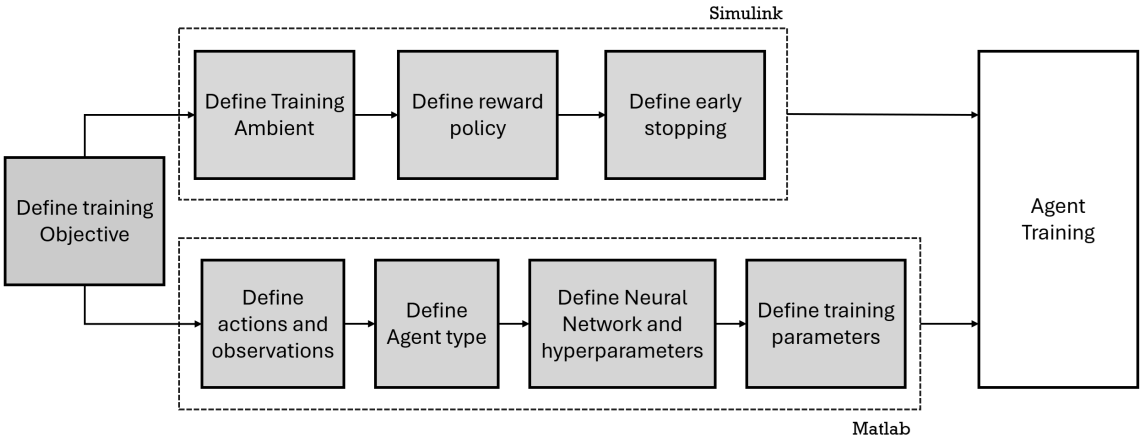

Figure 3.1: RL training pipeline

[1]Chess is surely one of the most challenging scenarios for a machine to deal with and learn to play, given the immense number of possible combinations and the fact that no two chess games are exactly alike. What is interesting to reflect on, however, is that while training a RL agent to play chess undoubtedly requires many episodes, the actual number of steps per episode is relatively low, which may seem counterintuitive.

3.1.1 MATLAB Implementation

As previously mentioned, the RL agent requires a dual implementation. The Simulink version is primarily used for in-training purposes and deployment, with minimal differences between the two. On the other hand, the MATLAB implementation focuses on defining all the metrics necessary to set up the training phase. It is important to note that the MATLAB script for deployment is entirely different from the one used for training, as will be explained later. For the training phase, there are four fundamental steps that cannot be skipped.

Actions and Observations

Every agent that moves and acts within an environment receives observations from the surroundings, which allow the agent to understand its position in the environment or how it has changed after taking an action. The actions represent the ways in which the agent actively interacts with the environment, whether by moving through it or modifying it in some way.

In the case study adopted for this thesis, the RL agent needs to learn how to navigate within the state space to trace a trajectory that satisfies specific constraints. Referring to Table 2.1, the state space is five-dimensional, represented as a $10 \times 2 \times 2 \times 2 \times 2$ matrix, with a total of 160 possible states. The state space is initially defined as a 5D matrix filled with zeros.

As the agent learns to navigate through the state space, the **observations** coming from the simulated environment take the form of a *vector with 160 elements*. This vector is obtained by flattening the state space matrix. Unfortunately, the system only allows observations to be passed as vectors, making this flattening step necessary. The impact of this requirement on the final results is still a subject of debate.

To guide the vehicle through the various states, the agent must provide the location of the next state space position, the **actions**, as a *five-element vector*. During each simulation time step, the agent outputs an array containing the next position in the state space, and the 5D matrix is updated by changing the value of the specified position from zero to one. This helps the agent keep track of the states it has already explored. The updated matrix is then flattened and reported as the next observation.

MATLAB Tip

Initialize the matrix representing the state space (m2) and use the specified functions to determine actions and observations.

```
m2 = zeros(10,2,2,2,2)
actions = rlFiniteSetSpec([5, 1])
observations = rlNumericSpec([80, 1]);
```

Define Agent and NN hyperparameters

After defining the actions and observations, the next step is to choose the type of RL agent and design the neural network architecture. In this thesis, two types of RL agents were primarily used: *Proximal Policy Optimization* (PPO) and *Deep Q-Network* (DQN). These agents differ in several characteristics, which will be explored further in Section 3.1.3.

When it comes to defining the neural networks employed by these agents, there are two possible approaches:

- Using the default network provided by the MathWorks library.
- Creating a custom network, fully configurable according to specific preferences.

The default network architecture suggested by MATLAB is generally recommended, as it has been optimized to work effectively with most projects. While using the default architecture, it is also possible to specify parameters such as the number of nodes (neurons) and the inclusion of a *Long Short-Term Memory* (LSTM) layer. This can be achieved using the following command:

MATLAB Tip

Initialize a pre-defined neural network structure and specify number of nodes and the use of an LSTM layer.

```
initopts = rlAgentInitializationOptions("NumHiddenUnit",80, ...
        "UseRNN",false);
```

In the previous code example, a default network architecture was created with 80 nodes and no use of an LSTM layer (since the flag *UseRNN* is set to *false*). For

those interested, a deeper explanation on the LSTM working principle is provided in Appendix B for the interested reader. The structure of the resulting network is summarized in Table 3.1. MATLAB automatically adjusts the number of input and output layer nodes based on the specified actions and observations, ensuring compatibility with the task at hand.

If the *UseRNN* flag is set to *true*, MATLAB will introduce a LSTM layer with the same number of nodes defined in the *NumHiddenLayers* option. The created network is different from the previous one, as shown in Table 3.2. It is important to note the placement of the LSTM layer, which may be considered debatable in comparison to common practices in literature. Some designers may disagree, preferring a different configuration, making the creation of a custom network necessary in certain cases.

Layer Number	Layer Type	Number of nodes
1	Feature Input	80 features
2	Fully Connected	80 fully connected layer
3	ReLU	ReLU
4	Fully Connected	80 fully connected layer
5	ReLU	ReLU
6	Fully Connected	1 fully connected layer

Table 3.1: Resulting default network architecture.

Layer Number	Layer Type	Number of nodes
1	Feature Input	80 features
2	Fully Connected	80 fully connected layer
3	ReLU	ReLU
4	Fully Connected	80 fully connected layer
5	ReLU	ReLU
6	LSTM	LSTM with 80 hidden units
7	Fully Connected	1 fully connected layer

Table 3.2: Resulting default network architecture with LSTM layer.

By specifying a custom network, the designer is free to explore various architectures, such as deeper networks composed of additional layers. It is also possible to adjust the number of nodes in each layer, creating pyramidal-shaped networks that either compress or decompress the incoming information for different purposes. One such purpose could be reducing the number of nodes required in an LSTM network, as LSTM layers significantly affect training time. In this thesis, the inclusion of an LSTM layer led to a training time increase ranging from three to ten times, while the performance improvements were not always proportional to this added complexity.

With a custom network, designers can also concatenate multiple input layers, creating V-shaped or Y-shaped architectures. By installing the appropriate MATLAB Toolbox, users also gain access to the Deep Network Designer, a tool that helps visually design neural networks, including specifying various architectural aspects. However, this tool is more suitable for data-driven approaches since is also possible to specify datasets and training parameters; the most effective way to specify a custom network is through the dedicated library commands, which provide powerful tools for network design.

An example of MATLAB code used to specify a custom network is provided below, and the structure of the resulting network is summarized in Table 3.3. The custom network defined shows some of the features discussed before like the pyramidal-shaped used to compress information before the introduction of the LSTM layer allowing the use of a reduced number of nodes within it.

MATLAB Tip

Define a totally custom network.

```
CustomNetwork = [
    featureInputLayer(80,"Name","input_1")
    fullyConnectedLayer(80, 'Name', 'fc1')
    reluLayer('Name', 'relu1')
    fullyConnectedLayer(64, 'Name', 'fc2')
    reluLayer('Name', 'relu2')
    fullyConnectedLayer(32, 'Name', 'fc3')
    lstmLayer(32, 'OutputMode', 'sequence', ...
        'Name', 'lstm1')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(64, 'Name', 'fc3')
    reluLayer('Name', 'relu3')
    fullyConnectedLayer(1, 'Name', 'output')
];
```

Training Parameters

In addition to configuring the neural network parameters, it is essential for an effective system to specify all the necessary training parameters. Training is a critical

Layer Number	Layer Type	Number of nodes
1	Feature Input	80 features
2	Fully Connected	80 fully connected layer
3	ReLU	ReLU
4	Fully Connected	64 fully connected layer
5	ReLU	ReLU
6	Fully Connected	32 fully connected layer
7	LSTM	LSTM with 32 hidden units
8	ReLU	ReLU
9	Fully Connected	64 fully connected layer
10	ReLU	ReLU
11	Fully Connected	1 fully connected layer

Table 3.3: Custom network architecture.

component of the developed system, and correctly setting up the training characteristics is one of the most important aspects of the RL approach. MATLAB provides specific functions for defining these parameters.

The most crucial parameters in training an RL system are the number of steps per episode and the number of episodes. If the number of steps per episode is set too low, the agent may not have enough time to complete the required maneuvers and, as a result, may learn very little. On the other hand, if the number of episodes is insufficient, the training algorithm may fail to generalize an optimal policy for the agent, leading to an agent that does not reach its full potential. The number of episodes normally increases with the complexity of the task and so the neural network required to generalize the correct behavior.

Finding the optimal training parameters is not straightforward and requires fine-tuning based on experience during development. However, a good developer should at least have a clear understanding of how many steps per episode the system should perform. MATLAB also provides functionality to specify an early stopping method during training. For example, if the mean total reward exceeds a specified threshold, it may indicate that the agent has learned enough, making further training unnecessary. Additionally, certain conditions, such as a minimum number of episodes or a minimum reward threshold, can be set to ensure that the agent is sufficiently trained before saving it.

Saving the trained agent is another crucial aspect that should not be overlooked. Incorrect specification here can lead to the loss of the entire training process, preventing the agent from being evaluated or reused later. This would be a significant waste, as each training may require a significant amount of time and the company will surely be not happy about that.

The easiest way to exclude this eventuality is to set the *SaveAgentCriteria* on *EpisodeCount* and fix the number of the episode on which save the agent and the number of episodes to train the agent (*MaxEpisodes*) with the same variable, as depicted in the following *MATLAB Tip*. This will ensure that the agent will be saved once reached the last episode of the training.

An example of code implementation for training parameter specification is provided below. In the current case study, the number of steps per episode is set to 1600, as there are 160 states, and the goal is to allow the agent to experience more than just a single pass through the minimum number of states needed to fill the matrix by ones. It has been found that an effective training process typically requires between 800 and 1200 episodes. Normally, the final agent is saved after the last episode concludes.

MATLAB Tip

Define training parameters.

```
trainopts = rlTrainingOptions("MaxEpisodes",episodes_to_train, ...
    "MaxStepsPerEpisode",1600, ...
    "StopTrainingValue", -12720, ...
    "StopTrainingCriteria","AverageReward", ...
    "UseParallel",false, ...
    "SaveAgentValue",episodes_to_train, ...
    "SaveAgentCriteria","EpisodeCount", ...
    "SaveAgentDirectory", pwd + "\RL_agents\PPO\", ...
    "Plots", "training-progress");
```

3.1.2 Simulink Implementation

The counterpart to the MATLAB implementation is the Simulink model, which is primarily used to enable the agent to interact with the developed simulated environment and receive rewards. The Simulink model for the VTD is shown in Figure 3.2. This section provides an in-depth explanation of each block used in the model, except for the *reward calculator* to which is dedicated the entire Section 3.1.4.

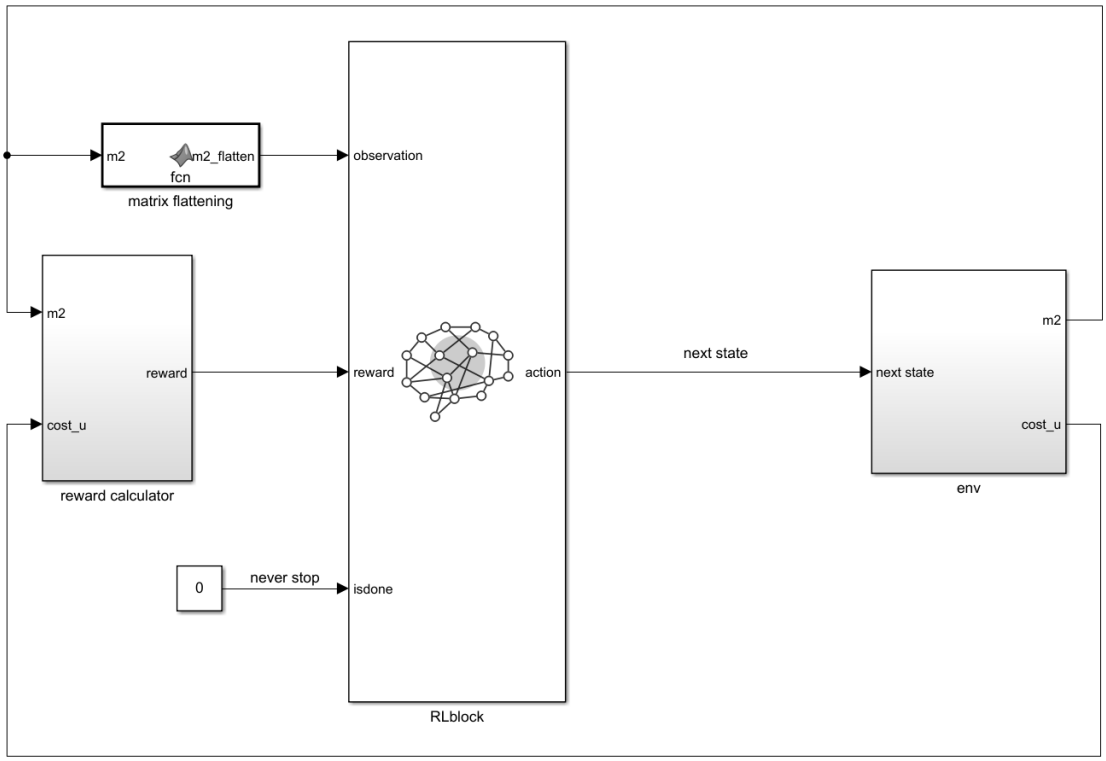

Figure 3.2: Developed Simulink model

RLblock

The core block in the Simulink model is the *RLblock*, which contains the RL agent, loaded from the MATLAB Workspace, either in its untrained state or as a pre-trained agent coming from previous training efforts. All other blocks in the model serve the RLblock, as it interacts with the simulated environment (*env* block). The RLblock receives observations from the environment, performs actions, and received the calculated reward. As shown in the Figure 3.2, there is one output port and three input ports.

- The **output** port delivers actions to the Environment block at each time step of the simulation.
- The three **input** ports are used for:
    1. **Observations**: The RLblock receives observations from the Environment block, informing the agent of how the environment's state has changed after

taking an action. In the case study of this thesis, a *matrix flattening* block is placed before the observation input. This block transforms the 5D matrix coming from the environment into a 160-element vector, as explained in Section 3.1.1.

2. **Reward**: At each time step, the agent receives the computed reward from the *reward calculator block*. There is no need for the designer to integrate the reward manually, as the RLblock automatically accumulates the total reward and makes it available in the MATLAB Workspace among the other simulation variables at the end of the training.
3. **IsDone**: This input, while not necessary for the current case study, can be useful in other scenarios. It allows the agent to terminate the current episode prematurely, speeding up the training process. Although this feature is not required in this study (as the agent always has more to learn from the environment), it is useful in situations such as training a robot to walk. Early in the training, the robot may fall frequently, and when it does, it is unlikely to learn anything further about walking in that episode. In such cases, the episode can be ended early, for example, by designing a small logic providing a high signal when the robot's core falls below a certain vertical threshold.

Environment block

The developed *Environment* block (shown in Figure 3.3) computes, at each time step, the changes in the environment based on the actions received from the *RLblock*. Before diving into the logic behind updating the environment state, it is important to highlight the blocks surrounding the *MATLAB function* block located in the center.

Starting from the left side of the figure, we can see the input of the *env* block, which is the five-element vector coming from the *RLblock*. This action is decoded and multiplexed by the next two Simulink blocks (a MATLAB function and a multiplexer) to reformat the array into a more convenient structure. Throughout the Environment block, three blocks labeled $\frac{1}{z}$ are visible. These are *Unit Delay* blocks, which serve to initialize certain values at the start of the simulation, preventing algebraic loops.

An algebraic loop occurs when a signal loop in a model contains only blocks with direct feedthrough, meaning that the output value for a given time step depends directly on the input value of the same time step. This creates a circular dependency, resulting in an algebraic equation that must be solved at each time step, increasing computational cost. The problem is most evident during the first time step when there is no previous output to use as input, making it impossible to calculate the

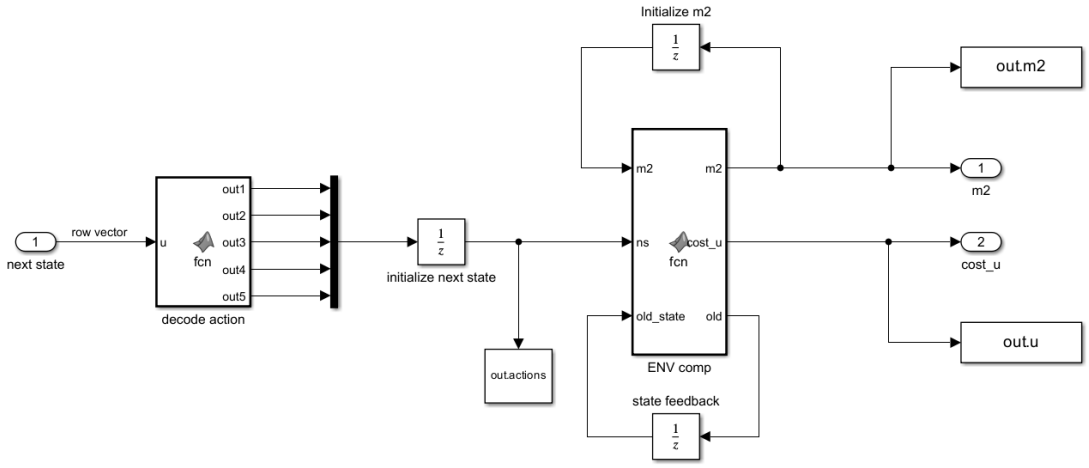

Figure 3.3: Inside of the Environment block

value directly. The Unit Delay blocks address this issue by initializing key quantities and synchronizing the information flows, ensuring that the input to the block at time t is available at time t+1.

There are three main uses for these Unit Delay blocks:

Initialize m2 Each time a new action is received, the state matrix is updated. This block provides feedback, making the updated matrix available at the next time step.

Initialize next state To prevent an algebraic loop within the *ENV comp* block, the first action must be initialized. It was discovered that using a random initialization, rather than starting at [0,0,0,0,0], helps improve the training process by starting the agent from different positions.

State feedback This block inputs the action from the previous time step into the *ENV comp* block for use in a specific computation that will be explained later.

The three blocks labeled *out.x* are *To Workspace* blocks, which send the connected information to the MATLAB workspace at the end of the simulation, allowing the user to review the data.

MATLAB Tip

Define random array for initialization.

```
random_array = [randi(10), randi(2), randi(2), ...
        randi(2), randi(2)]
```

It is important to note that both the *Initialize next state* and *State feedback* blocks are initialized with the same random array. This is because three different processes are carried out within the env comp block.

1. Updating the state matrix: The first task is to update the state matrix by setting the value of the state identified by the RLblock to 1, using the following formula. Let m2 be the state matrix and ns the vector containing the coordinates of the chosen state:

$$m2(ns(1), ns(2), ns(3), ns(4), ns(5)) = m2(ns(1), ns(2), ns(3), ns(4), ns(5)) + 1 \tag{3.1}$$

   This formula updates the state matrix, marking the states already explored and providing this information as observations to the RLblock.

2. Computing the u-cost: The second process involves calculating the u-cost, which depends on the vector generated by the RLblock at the current time step and the vector generated at the previous time step. The formula used is:

$$cost = (||ns_t - ns_{t-1}|| - 1)^2 \tag{3.2}$$

   This formula first calculates the Euclidean distance between the current state ns_t and the previous state ns_{t-1}. Then, it subtracts 1, so the cost is 0 when two consecutive states have a unitary Euclidean distance. The squaring ensures the cost is always positive and increases when the distance between consecutive states is greater than 1. The practical use of these calculations will be clarified later when discussing about the choice of the reward function.

3. State feedback: Finally, the state generated at the current time step is saved into a variable, which is passed as feedback to the *env comp* block via the *State feedback* block for use in the next iteration.

Once both the MATLAB script and the Simulink model are fully developed and error-free, is possible to start the training process of the RL agent.

3.1.3 RL Agents

This section discusses the two different Reinforcement Learning (RL) agents primarily employed in this thesis: *Proximal Policy Optimization* (PPO) and *Deep Q-Network* (DQN). The choice of these two architectures was driven by two main factors.

First, the objective was to explore examples from the two primary families of RL agents: on-policy (PPO) and off-policy (DQN).

on-policy agents learn the policy they are currently following. During training, the agent's behavior is dictated by the same policy it is trying to optimize.

off-policy agents learn from data that may have been generated by a policy different from the one currently being optimized. This allows them to reuse past experiences or data generated by other policies by the use of a replay buffer.

This allowed for a broader understanding of how each type functions in practice. Second, both agents were chosen for their compatibility with discrete actions and observations, a requirement for this particular case. Not all RL agents available in MATLAB, such as DDPG and TD3, can handle discrete action spaces, making PPO and DQN the ideal candidates for this work.

In general, RL agents contain two components: a *policy* and a *learning algorithm*. The policy is a mapping from the current environment observation to a probability distribution over possible actions. Within an agent, the policy is implemented by a function approximator with tunable parameters and a specific approximation model, such as a deep neural network. The learning algorithm continuously updates the policy parameters based on actions, observations, and rewards. The objective of the learning algorithm is to find an optimal policy that **maximizes**[1] the expected discounted cumulative long-term reward.

Agents can use approximators in two ways:

Critics For a given observation and action, a critic estimates the policy value, which is the expected discounted cumulative reward.

Actors For a given observation, an actor returns the action that maximizes the policy value.

Agents that rely solely on critics to select actions are called value-based agents. These agents use an approximator to represent either a value function (value as

[1]Note that the maximization of the computed reward is not a strict rule but is the default approach implemented in MATLAB. Special attention must be paid to whether the policy is designed to maximize or minimize the reward during training, especially when dealing with unknown systems. This decision can significantly impact the success of the developed application, as incorrect handling of the reward policy could lead to suboptimal or unintended outcomes.

a function of the observation) or a Q-value function (value as a function of both observation and action). Value-based agents are generally more effective in discrete action.

Agents that rely only on actors to select actions use a direct policy representation and are referred to as policy-based agents, which are not covered in this thesis.

Agents that use both an actor and a critic are called actor-critic agents. In these agents, the actor learns the best action to take based on feedback from the critic, rather than directly from the reward. Meanwhile, the critic learns the value function from the rewards, allowing it to evaluate and guide the actor.

DQN

The *Deep Q-Network* algorithm is an *off-policy* reinforcement learning method designed for discrete action spaces. A DQN agent trains a Q-value function to estimate the expected discounted cumulative long-term reward when following the optimal policy. DQN is a variant of Q-learning that incorporates a *target critic* and an *experience buffer*.

To estimate the value of the optimal policy, a DQN agent uses two parameterized action-value functions, each maintained by a corresponding critic:

Critic $Q(S, A; \phi)$ Given an observation S and action A, this critic stores the estimated value of the expected discounted cumulative long-term reward when following the optimal policy.

Target critic $Q_t(S, A; \phi_t)$ To improve optimization stability, the agent periodically updates the target critic parameters ϕ_t using the latest critic parameter values.

Both $Q(S, A; \phi)$ and $Q_t(S, A; \phi_t)$ are implemented using function approximator objects with the same structure and parameterization.

The DQN agents follow the training algorithm outlined below, updating their critic model at each time step. Before training begins, the critic $Q(S, A; \phi)$ is initialized with random parameter values ϕ and the target critic parameters are initialized to match: $\phi_t = \phi$. The training algorithm pipeline is:

1. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \underset{A}{\operatorname{argmax}}\, Q(S, A; \phi) \tag{3.3}$$

2. Execute action A, then observe the reward R and next observation S'.

3. Store the experience (S, A, R, S') in the experience buffer.

4. Sample a random mini-batch of M experiences (S_i, A_i, R_i, S_i') from the experience buffer.

5. For each experience in the mini-batch:

   - if S_i' is a terminal state, set the value function target y_i to R_i.
   - Otherwise, set
   $$y_i = R_i + \gamma \max_{A'} Q_t(S_i', A'; \phi_t). \tag{3.4}$$

6. Update the critic parameters by one-step minimization of the loss L across all sampled experiences,
$$L = \frac{1}{2M} \sum_{i=1}^{M} (y_i - Q_t(S_i, A_i; \phi))^2. \tag{3.5}$$

7. Update the target critic parameters depending on the target update method.

8. Update the probability threshold ϵ for selecting a random action based on the decay rate applied to ϵ.

When defining a DQN agent in MATLAB, there are numerous parameters that can be configured and tuned for effective training. The most commonly used parameters during the development of this work, along with their explanations, are summarized in Table 3.4. Many more options are available over the ones reported, with which the designer can have a full overview and take the full control over the behavior of the agent during the training phase.

PPO

Proximal Policy Optimization is an *on-policy*, policy gradient reinforcement learning method suitable for environments with either discrete or continuous action spaces. It directly estimates a stochastic policy and uses a value function critic to evaluate the policy. The algorithm alternates between sampling data through interaction with the environment and optimizing a clipped surrogate objective function using stochastic gradient descent. This clipped objective function improves training stability by limiting the size of the policy change at each step.

PPO is a simplified version of TRPO (Trust Region Policy Optimization). Specifically, PPO has fewer hyperparameters, making it easier to tune and less computationally expensive than TRPO.

During training, a PPO agent:

Option	Explanation
DiscountFactor	Discount factor applied to future rewards during training (γ).
SequenceLength	Maximum batch-training trajectory length when using a recurrent neural network.
NumStepsToLookAhead	Number of future rewards used to estimate the value of the policy, specified as a positive integer.
EspilonGreedyExploration	Options for epsilon-greedy exploration, in which is possible to determine the probability threshold to either randomly select an action or select the action that maximizes the state-action value function, the minimum value for ϵ and its decay rate in time.

Table 3.4: Primarily used DQN options.

- Estimates the probabilities of taking each action in the action space and randomly selects actions based on the probability distribution.
- Interacts with the environment for multiple steps using the current policy before using mini-batches to update the actor and critic parameters over multiple epochs.

To estimate the policy and value function, a PPO agent maintains two function approximators:

Actor $\pi(A|S;\theta)$ The actor, with parameters θ, outputs the conditional probability of taking each action A when in state S. When dealing with discrete action spaces, as in this case study, the sum of the probability of taking each action is 1.

Critic $V(S;\phi)$ The critic, with parameters ϕ, takes observation S and returns the expected discounted long-term reward.

The training algorithm employed for the PPO agent is the following:

1. Initialize the actor $\pi(A|S;\theta)$ and the critic $V(S;\phi)$ with random parameter values θ and ϕ.
2. Generate N experiences by following the current policy. The sequence of experiences consists of state-action-reward triplets:

$$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1} \dots S_{t+N-1}, A_{t+N-1}, R_{t+N} \tag{3.6}$$

3. For each episode $t = t_s, t_{s+1}, \ldots, t_{s+N-1}$ compute the return and advantage using one of the two advantage functions implemented in MATLAB: Finite Horizon [15] or Generalized Advantage Estimator (GAE) [16]. The return G_t is then computed.

4. Learn from mini-batches of experiences over multiple epochs by sampling a random mini-batch of size M from the current experiences, where each mini-batch element contains a current experience, return, and advantage values.

5. Update the critic parameters by minimizing the loss function L_{critic}:

$$L_{critic}(\phi) = \frac{1}{2M} \sum_{i=1}^{M} (G_i - V(S_i; \phi))^2. \tag{3.7}$$

6. Normalize the advantage values D_i based on recent unnormalized advantage values.

7. Update the actor parameters by minimizing the actor loss function L_{actor}.

8. Repeat steps 3 through 7 until the episode reaches a terminal state.

To encourage exploration, an entropy loss term can be added to the actor loss function. Entropy is higher when the agent is uncertain about the next action, and maximizing entropy (or minimizing negative entropy) increases agent uncertainty, promoting exploration. A larger entropy loss weight can be specified to further enhance exploration, helping the agent escape local optima.

When defining a PPO agent in MATLAB, various parameters can be configured and tuned for effective training. The most commonly used parameters during this project are summarized in Table 3.5. Many additional options are available, allowing the designer to fully control the agent's behavior during the training phase.

In conclusion, the differences between PPO and DQN agents not only shape their theoretical foundations but also significantly impact their practical implementations, particularly in environments where stability and adaptability are crucial. While DQN is easier to implement for discrete action spaces, PPO offers greater flexibility, as it handles both continuous and discrete spaces. Moreover, PPO's enhanced training stability makes it better suited for more complex, real-world applications. When facing practical implementations, the selection of the appropriate algorithm based on the specific demands of the task is crucial.

Option	Explanation
ExperienceHorizon	Number of steps used to calculate the advantage.
MiniBatchSize	Mini-batch size used for each learning epoch.
EntropyLossWeight	Entropy loss weight, which must behave between 0 and 1. A higher entropy loss weight value promotes agent exploration by applying a penalty for being too certain about which action to take. Doing so can help the agent move out of local optima..
DiscountFactor	Discount factor applied to future rewards during training.

Table 3.5: Primarily used PPO options.

3.1.4 Choice of Reward Function

This section focuses on the design of the most successful reward function developed during the thesis internship. As mentioned in previous chapters, designing the reward function is one of the most critical aspects of training a neural network using reinforcement learning. The reward function directly influences the agent's behavior, guiding the learning process by providing feedback on the actions the agent takes in its environment. A poorly designed reward function can lead the agent to learn suboptimal or even harmful strategies that do not align with the intended goals. Conversely, a well-crafted reward function encourages the agent to explore efficiently and adopt behaviors that maximize long-term success. This becomes especially important in complex environments where the agent must balance short-term rewards with long-term objectives.

In summary, the reward function serves as the foundation upon which the entire learning process is built, making its careful design crucial for achieving the desired outcomes. There are no strict rules for designing specific reward functions, as they depend heavily on the observation signals from the environment and the type of actions the agent can perform. For example, two designers training a robot to walk may develop fundamentally different reward functions if they rely on different observations. The differences become even more pronounced if the possible actions vary, such as controlling the robot through torque or relative position.

For most reinforcement learning applications, particularly in complex scenarios, the reward function consists of multiple components that collectively determine the total reward at each time step of the simulation. Returning to the example of the walking robot, two distinct reward contributions might be based on the robot's head height above the ground and its longitudinal speed. The first contribution informs

whether the robot is still upright or falling, while the second evaluates how fast the robot is moving forward.

May happen that even if a reward functions has been correctly conceptualized, fails during the training because the contributions are not balanced one to each others. At this point a useful trick to follow for developers is to pair each reward component with a scalar factor:

$$R = a \cdot r_1 + b \cdot r_2 + c \cdot r_3, \tag{3.8}$$

where a, b and c are scalar quantities that can be fine-tuned during training without altering the reward function itself. For instance, in the case of the walking robot, if the scalar factor associated with the robot's head height is too large, the RL agent will prioritize keeping the robot upright without focusing on walking. On the other hand, if the factor for longitudinal speed is too high, the agent might learn to increase speed rapidly without ensuring stability, resulting in a robot that "falls like a cut tree" by purpose.

However, even in this scenario, there is no strict rule for determining the right proportions between the reward factors. Fine-tuning these scalar values by finding the optimal proportions is an iterative process that depends on the specific dynamics of the task that is intended to balance.

Another key distinction in designing a reward function lies in its overall structure. There are two main approaches: The first is to design a reward function that delivers both positive and negative rewards. In this case, when the agent performs well, the total reward tends to remain above zero, driving the agent to achieve increasingly higher positive values. The second approach is to define an always-negative reward, where the best the agent can do is minimize the negative reward. In this scenario, the agent's goal is to bring the total reward closer to zero, which acts as an asymptote.

In MATLAB, even though the default behavior of an RL agent is to maximize the total reward (as described in Section 3.1.3), the second approach—minimizing negative rewards—yielded the best results in practical implementation. It was observed that in certain scenarios, the agent learns to avoid penalties rather than continually seeking to increase rewards, which can lead to sub-optimal behavior. By focusing on minimizing the negative reward, the agent develops strategies that are more stable and aligned with the intended goals.

Reward Function proposal

During the development of this thesis, several reward function architectures were proposed and tested in practice, with some yielding poor results. However, through

a methodical and conceptual approach, a reward function was successfully formulated to guide the agent's learning process as intended.

Recalling the primary objectives outlined in Section 2.2.2, the agent was expected to cover the given state space in a manner that:

- The agent should output consecutive states with a low Euclidean distance, but avoiding a constant 1-distance step at each transition.
- By the end of the simulation, the state space matrix should be as uniformly covered as possible.

The reward function was designed to reflect these objectives, translating them into functional components continuously evaluated during training.

Reward Function Components

The proposed reward function consists of three main contributions. The first is described by the formula:

$$r_1 = -\sum_{x \in S} \frac{1}{\epsilon + M(x(N))} \tag{3.9}$$

where S represents the state space, x is an element in the matrix, N is the simulation time step, and *M(x)* denotes the number of times the agent has visited a particular state space location. Initially, all *M(x)* values are set to zero. The small value ϵ is introduced to prevent division by zero at the start of the simulation.

This first contribution encourages the agent to explore new states by imposing an increasingly negative reward if it repeatedly visits the same state. As *M(x)* increases, the agent is motivated to take more diverse actions to counteract this growing negative reward. The behavior of this part of the function is illustrated in Figure 3.4.

The second contribution is described by the following formula:

$$r_2 = -\sum_{x,y \in S} (M(x(N)) - M(y(N)))^2 \tag{3.10}$$

This contribution calculates and accumulates the squared difference between each *M(x)* and all other states in the matrix. The goal is to assign an increasingly negative reward as the differences between the states—based on how many times they have been explored—grow larger. The goal is to encourage the agent to keep the differences between the number of times each state is explored balanced.

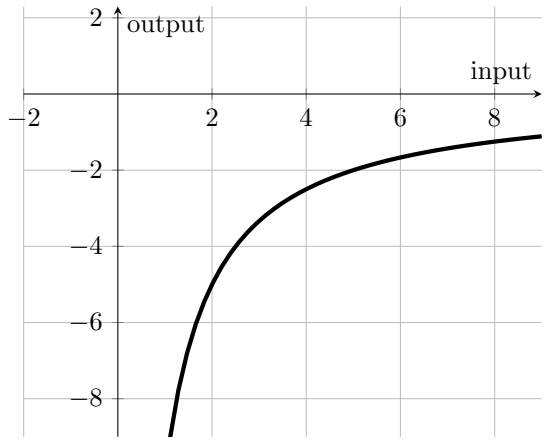

Figure 3.4: Proposed Reward Function first contribute.

When practically implemented in MATLAB, the complexity of this procedure increases exponentially with the number of possible states in the state space. This is due to the growing number of *M(x)* values and the combinations required for comparison with every other state, demanding significant computational power and time.

To address this issue, instead of comparing each *M(x)* with every other state, it is possible to restrict the comparisons to a local neighborhood. For instance, the squared difference between *M(x)* and its four nearest neighboring states (two in one direction and two in the opposite direction), for each dimension of the state space, can be calculated and accumulated, rather than comparing the entire matrix. This can be done using a relatively simple algorithm, where the size of the neighborhood window is passed as a parameter.

A simplified visualization of how the function progresses as the difference between states increases is shown in Figure 3.5.

The first two contributions aim to guide the agent in satisfying the second criterion mentioned in the previous section, ensuring that the state space matrix is covered as uniformly as possible. The first criterion, however, is achieved through the third contribution, which was previously discussed in Section 3.1.2. This contribution is defined by the formula:

$$r_3 = -(||ns_N - ns_{N-1}|| - 1)^2, \tag{3.11}$$

where ns_t and ns_{t-1} are the current state output by the *RLblock* and the output from the previous time step, respectively. This term calculates the Euclidean distance between two consecutive time steps, applying a greater penalty (negative reward)

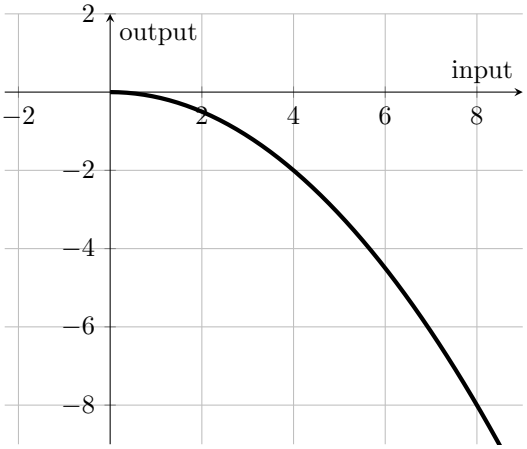

Figure 3.5: Proposed Reward Function second contribute.

when the distance between them is larger.

Bringing all three contributions together, the resulting reward function—expressed as a single equation with their associated factors—is:

$$R_t = -\sum_{x \in S} \frac{Q_1}{\epsilon + M(x(N))} - Q_2 \cdot \sum_{x,y \in S} (M(x(N)) - M(y(N)))^2 - R_1 \cdot (||ns_N - ns_{N-1}|| - 1)^2 \tag{3.12}$$

where Q_1, Q_2 and R_1 are scalar factors associated with each component. This reward function is computed at each time step during the training process, providing the agent with continuous feedback on its performance relative to the expected outcomes.

Reward Function Simulink block

The *Reward calculator* block has been practically implemented as shown in Figure 3.6. The inputs to this block are as follows:

- The matrix *m2* coming from the environment,
- The Euclidean distance between the current and the previous time step, depicted as *cost u*,
- The factors Q_1, Q_2 and R_1 coming from the MATLAB workspace,
- the window size for calculating the second reward contribution, also importing from workspace.

Initializing these variables in the MATLAB workspace and then using them in the Simulink model is advantageous, as it allows the user to adjust training parameters without modifying the Simulink model itself.

The described reward function is implemented inside the *MATLAB Function* block, and the output is passed through the subsystem, feeding into the *RLblock* input labeled reward. The lower-right part is dedicated to export variables into the workspace for after-training visualization.

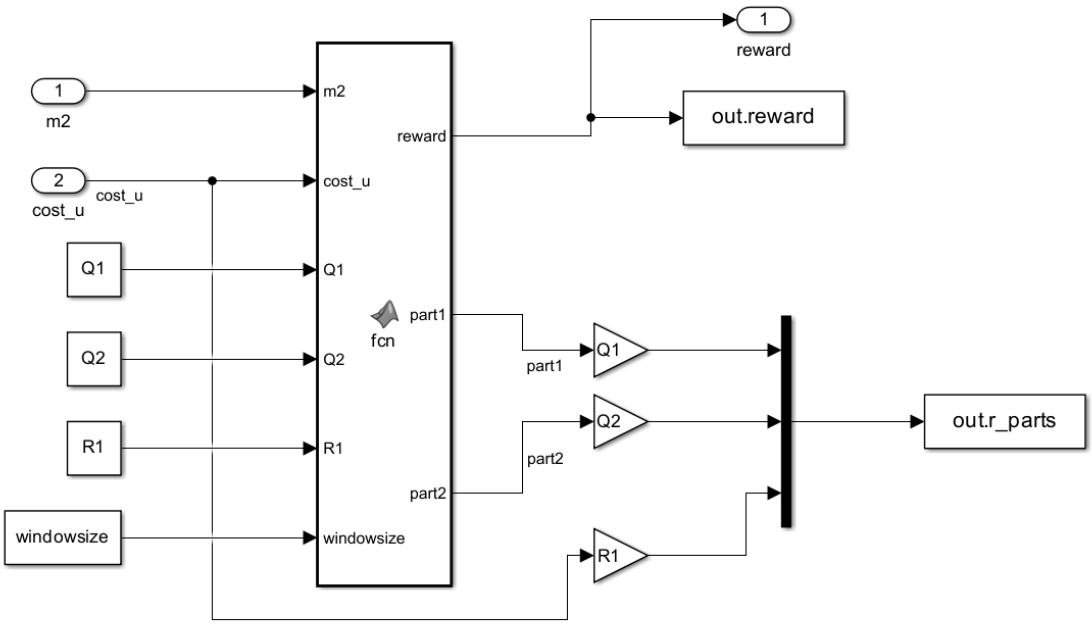

Figure 3.6: Inside of the Reward calculator block

3.2 Training phase

In this section, one of the final training attempts is analyzed, focusing on the case study covered in this thesis. Once the training environment is fully set up, the RL agent can be trained with a single line of code. Upon starting the training, MATLAB provides a graphical representation of the training progress. An example of this is shown in Figure 3.7, which displays the complete training process. Each point on

the graph represents the total reward computed by the agent in a specific episode.

It is evident that the agent improves its performance over time, as the reward for each episode moves closer to zero compared to the initial stages. In the early phase of training (up to around 50 episodes), the agent has no understanding of the optimal policy and behaves randomly, similar to a random number generator. During the next phase (from episode 50 to 150), the agent explores different policy solutions by extensively testing possible actions, which explains the sharp decline in episode rewards. From this point onward, the agent learns the policy effectively and adjusts its behavior to consistently achieve higher rewards.

Due to the way the reward function is designed, the agent's goal is to minimize the received negative reward by taking appropriate actions. As the graph shows, this approach proves to be quite effective.

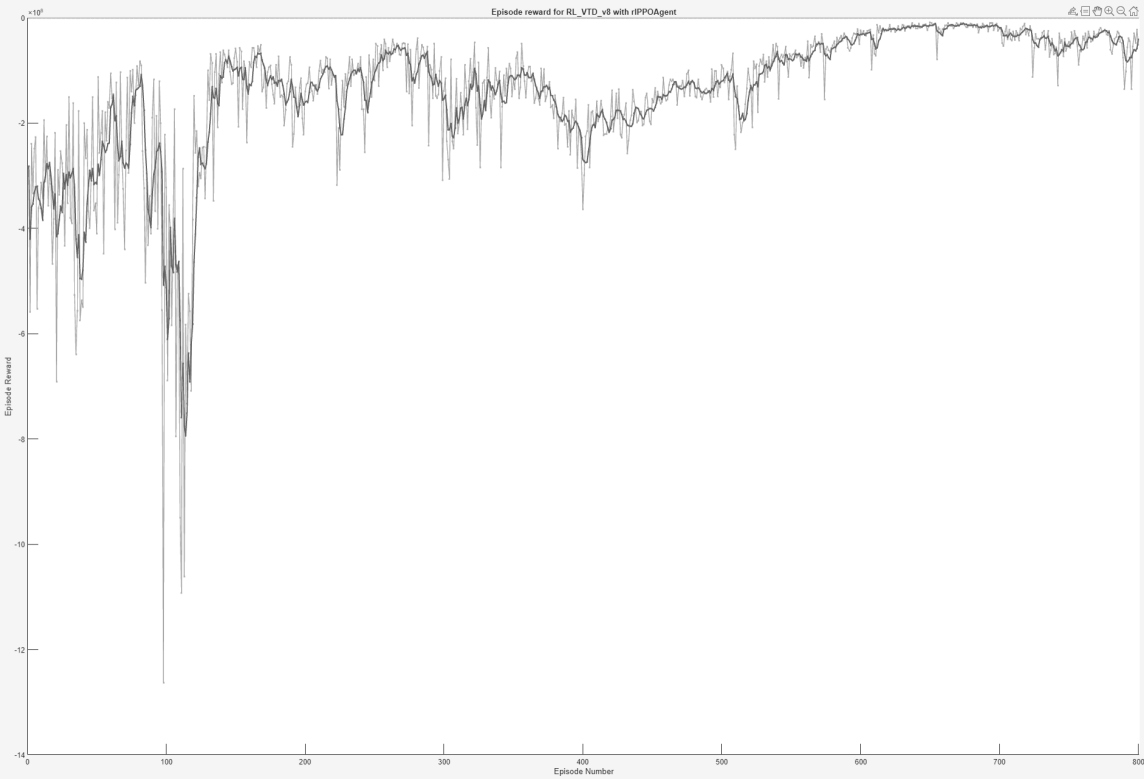

Figure 3.7: Example of training plot.

3.2.1 Key factors

Beyond analyzing the training graph, the most crucial task once the training concludes is to evaluate the agent's output to ensure consistency with the **desired outcomes**. Thanks to the output generated by the training in Simulink, it is possible to inspect the coverage of the *m2* matrix or the computed rewards for each time step of the simulation. For example, in this case study, if each episode consists of 1600 steps, it is expected that, by the end of the training, the *m2* matrix will be fully covered with values of 10 at the conclusion of each episode.

One recurring issue encountered during the more than 250 training sessions performed for this thesis is the agent learning a sub-optimal behavior, in which it repeats the same state without effectively transitioning between states. This behavior is problematic and unacceptable for obvious reasons. Therefore, monitoring the **number of zero-movement steps** became essential throughout the training process. In a successful training session, where the reward function and all metrics are properly defined, the number of zero-movement steps is expected to decrease over time. This improvement is illustrated in Figure 3.8, which records come from the same training of Figure 3.7.

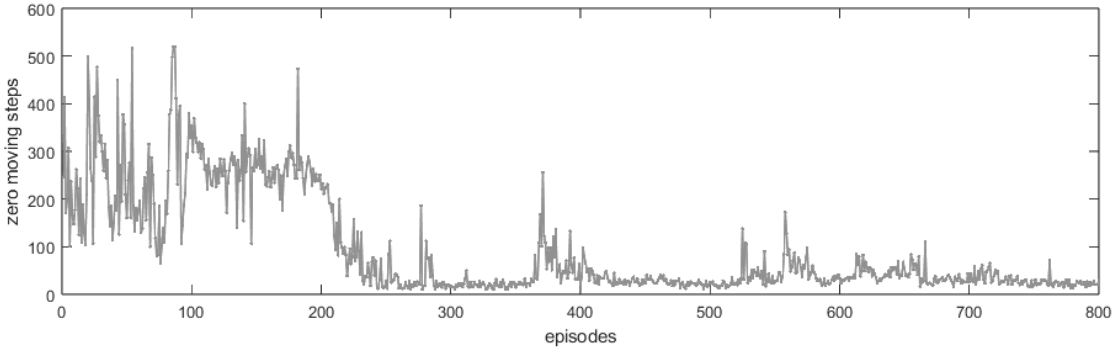

Figure 3.8: Number of zero moving steps during training.

The second key feature to verify once training concludes is the **mean Euclidean distance** of the performed steps. To demonstrate the advantages of this architecture compared to a simple random number generator, the mean distance must be lower than that achievable by a random generator. In the current case study, with a 5-dimensional state space in the form of $10 \times 2 \times 2 \times 2 \times 2$, and recalling the Section 2.2.2, the maximum Euclidean distance within this state space can be calculated by following equation 2.2, resulting in a central state coordinates equal to [5.5, 1.5, 1.5, 1.5, 1.5]. Then, by applying equation 2.3:

$$D^r_{max}([1,1,1,1,1],[5.5,1.5,1.5,1.5,1.5]) \approx 4.61, \tag{3.13}$$

where D^r_{mean} is the average Euclidean distance between states generated by a pseudo-random number generator.

A well-trained RL agent, however, should generate steps with a significantly lower mean distance. This can be demonstrated experimentally, as shown in Figure 3.9, which displays the computed average Euclidean distance of the agent's output across all training episodes. As observed, this distance decreases over time, eventually stabilizing around an average value of approximately 2.2, which is considerably smaller than the average distance computed for a random number generator.

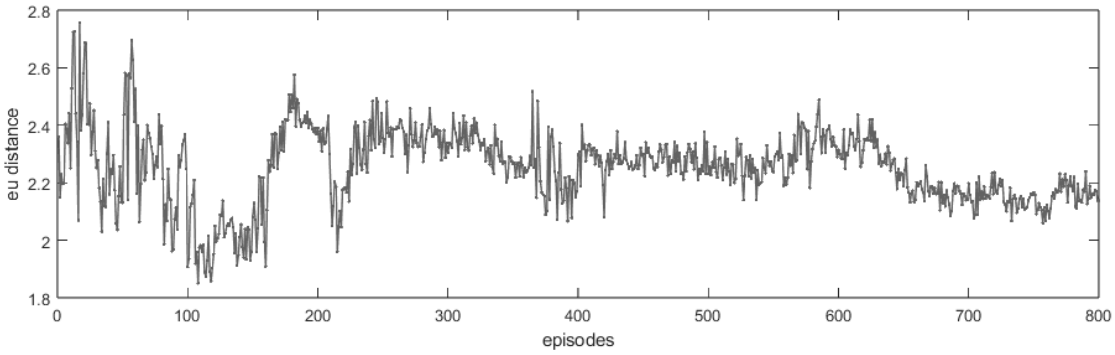

Figure 3.9: Average euclidean distance of the states generated during training.

3.2.2 Reward Function fine-tuning

During the development of the reward function, it became apparent that the computed Euclidean distances between states consistently yielded values below 10. This, combined with the fact that other components of the reward function produced values in the range of hundreds or thousands, made it necessary to significantly increase the multiplier associated with the Euclidean distance contribution up to 100 or more. This adjustment was essential to maintain a balance between the different contributions to the total reward. Without this scaling, the agent's focus on producing short-distance consecutive states would have been negligible, leading to an output that did not meet the expected behavior.

3.2.3 Before full-scale training

Before proceeding with training the RL agent using the chosen agent type and reward function, a good developer should always conduct several preliminary tests on the selected configurations. The best approach is to start by running a series of training sessions with a simplified version of the problem. This helps test the scripts and models that have been developed and provides an opportunity to fully understand their behavior before moving on to more complex scenarios. Moreover, this method offers a simplified yet meaningful preview of how the agent will behave during training.

In this case study, after finalizing the reward function, a simplified state space of 3 was used to train the agent. The agent performed very well, providing an initial confirmation that the reward function was appropriately designed. Many additional tests were conducted with increasing complexity to validate the architectural choices, eventually progressing to training on the full-scale state space.

The key point to consider is that if an agent, with a given reward function, cannot perform optimally in a simplified version of the problem, there is no realistic chance of successfully training the same agent with the same reward function on a full-scale task. During these tests, both DQN and PPO agents were tested in parallel. As expected, the PPO agent began to outperform the DQN when the state space dimension increased significantly (around 80 states). This was anticipated, as PPO is a more complex agent that employs two neural networks instead of just one.

For this reason, the training on the full-scale state space, as discussed in the previous sections, was conducted using a PPO agent.

Once again, the reproducibility of the generated test pattern is not an issue. Once the starting point is fixed, the agent will always produce the same test pattern. This is possible due to the deterministic nature of the neural network that the agent is based on. This ensures that each test can be reliably reproduced, allowing it to be conducted in different scenarios to evaluate the vehicle's response, or to compare the performance of two different vehicles using the same test pattern.

3.3 Deployment phase

The project transitions into the deployment phase once the developed agents demonstrate sufficient capability in achieving the defined goals. In this phase, the trained agents undergo field-testing to verify their effectiveness in performing the assigned tasks. Completed training sessions result in agents that MATLAB automatically saves as **.mat** files, containing all agent parameters alongside the computed weights

and biases from the training phase. Predictably, the file size varies based on the number of layers and nodes defined during agent network initialization.

Ultimately, this system aims to integrate seamlessly with the company's simulation environment, interfacing with the HiL and test bench. Thus, the optimal approach is to develop a fully autonomous system that can be used as a **black box** from which is expected an output at each time step, and integrated alongside other Simulink models already in use within the company testing facility.

3.3.1 Black box creation

As also specified in the previous sections, both a MATLAB script and a Simulink model must be developed in order to deploy the trained agent. This time the MATLAB script is way easier then the one used for the training phase, while the Simulink is similar.

MATLAB

The main objective of the development of a MATLAB script during the deployment phase is to initialize all necessary variables and the RL agent into the workspace, enabling Simulink to operate effectively and autonomously. Specifically, the trained agent can be loaded easily into the workspace using a simple function, as illustrated in the upcoming *MATLAB Tip*. Strictly speaking, using a MATLAB script for this initialization is not mandatory, as variables can be defined directly within Simulink. However, utilizing a script is highly recommended for several reasons.

In fact, it simplifies variable management, especially when adjustments are required or when selecting different agents for testing; having these definitions in a MATLAB script is far more manageable than embedding them within a Simulink model. This is because, once the Simulink model is integrated into the company's testing facility, it generates C code that is deployed on the HiL system. Each modification to the model necessitates regenerating the C code, which can be a lengthy process, particularly when integrated into large models—often requiring several hours to complete. For these reasons, initializing variables in a MATLAB script is a more efficient and flexible approach.

MATLAB Tip

Load RL agent and initialize m2.

```
load deploy_agents\Agent1200.mat

m2 = zeros(10,2,2,2,2);
```

Simulink

The Simulink implementation of the deployed agent serves as the core of the deployment phase and builds on the same *RL Agent block* used during training. Within this block, the specific agent name, loaded into the workspace via a MATLAB script, is referenced. However, in MATLAB versions prior to *2022b*, this method isn't feasible because the *RL Agent block* still didn't support C code generation. This limitation means that such block can't be used to integrate the agent into the company's testing framework. In those cases, a MATLAB function block would need to be used, employing specific functions to load the agent from the workspace and run neural network inferences.

In MATLAB versions *2022b* and later, the Simulink model used for the deployment is the one shown in Figure 3.10 which fully supports the *RL Agent block*. Here, the block's *reward* and *isdone* inputs are forced to zero during the whole execution, as they aren't necessary for deployment. However, the observations input remains active since it is required in this phase as well, to let understand the agent the already explored states and actuate its policy. The *environment* block setup also remains consistent with the configuration described in Section 3.1.2. Additionally, designers still have the option to use the reward calculator block to observe the reward the agent gets with the current behavior, although the agent effectiveness can be verified through alternative methods as preferred.

3.3.2 Final integration

In the final deployment model (shown in Figure 3.11), the RL model is embedded within a protective subsystem to avoid unnecessary modifications. The outputs of the *RLblock* are adjusted as follows to meet company requirements for seamless integration into their testing framework:

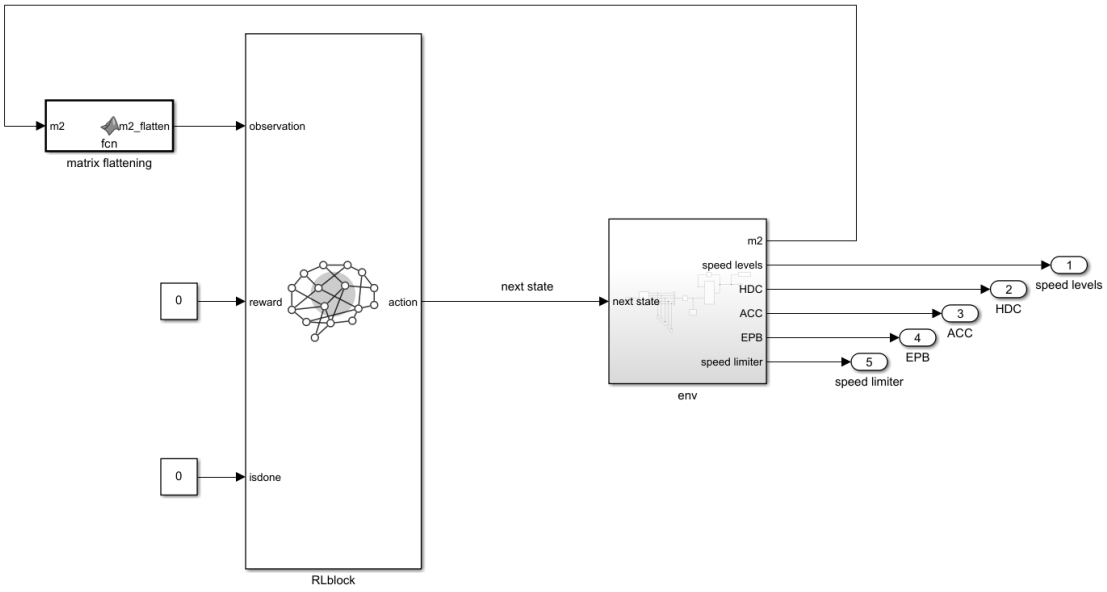

Figure 3.10: RLblock in deployment configuration.

- Speed Level Selection: The first output represents the network's choice across ten speed levels. However, the network outputs only a state-space coordinate (1–10) for longitudinal speed, rather than an exact speed. To map this output to actual speed values, a *switch case* is implemented within a *MATLAB function* block, which assigns the true longitudinal speed based on the network's state-space coordinate. This approach also allows for modifying the testing speed levels, letting untouched the RL agent, as long as the required number of speeds remains equal to ten.

- Button Press States: The remaining four outputs correspond to two-state buttons, with a final output requirement of either 0 or 1 (indicating unpressed or pressed). Since the network can only output values 1 or 2 due to the matrix-based state space, a *Compare To Constant* block is used. The block compares the input signal against 1 with a *greater-than* operand. The block outputs 0 if the input is 1 (indicating the button is unpressed) and 1 if the input is 2 (indicating the button is pressed).

These outputs provide the necessary reference signals for low-level controllers that direct the vehicle toward the target state.

During the C code generation, the used tools allows for adjusting the state-update frequency to best support the controller response time. The forecast optimal time

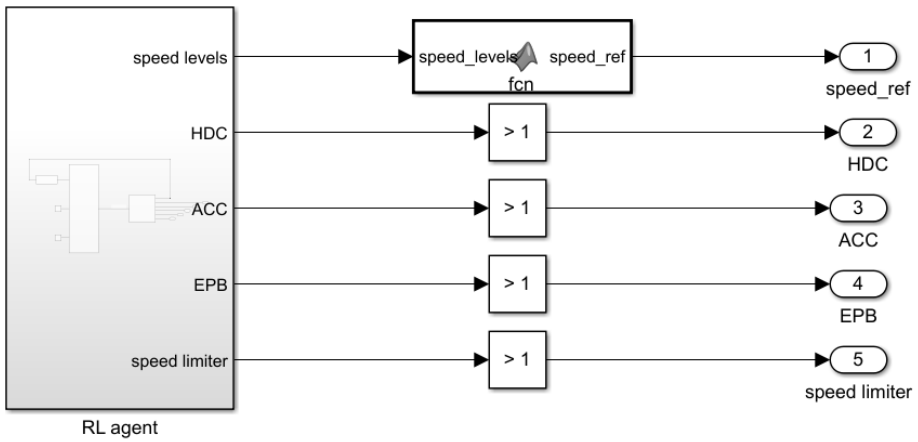

Figure 3.11: RL agent used as black box.

step ranges from 2 to 10 seconds. This interval gives the low-level controllers sufficient time to adjust the vehicle state before the next update.

Once the integration is complete, the vehicle can undergo testing with a new level of autonomy. A customized virtual circuit should then be designed specifically to test the ADAS functionalities of this case study, focusing particularly on the Hill Descent Control system. This circuit should incorporate varying elevations to assess the activation and deactivation of the HDC function, also straight stretches can be included, along with the presence of other vehicles to evaluate the coordination between HDC and Adaptive Cruise Control. The same can be done to validate the integration between HDC and the other ADAS included into the VTD like EPB and speed limiter.

The virtual circuit's design should fulfill all requirements set by the company for comprehensive ADAS testing and validation. Given that the vehicle, once equipped with the developed system, will exhaust all possible state combinations across prolonged testing sessions, the virtual circuit should ideally be a loop, this ensures continuous operation for the time necessary without human intervention. An example of a basic looped virtual test circuit is illustrated in Figure 3.12, offering a layout conducive to testing the HDC integration.

After the vehicle has undergone sufficient testing on the circuit—continuously looping on the circuit and dynamically adjusting its states—the test engineer can review the logged signals captured during the execution. This is crucial for identifying any potential malfunctions or software bugs that may have emerged. The logged data provides insight into how the ADAS features respond over time, highlighting areas

for further debugging.

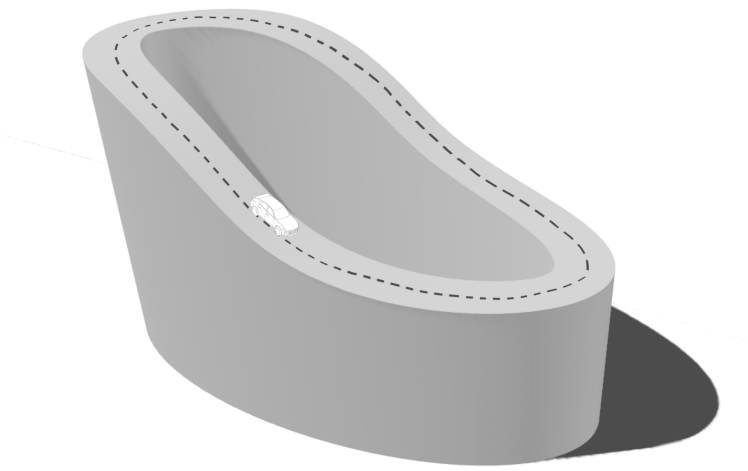

Figure 3.12: Example of virtual circuit for HDC integration testing.

3.3.3 Older version integration

The procedure described in this section illustrates a practical **workaround** to address the limitations posed by the use of a MATLAB version older then the minimum required one, this issue was encountered at the end the of the internship period when the was the necessity of test the developed system within the company testing framework. Unfortunately, the only environment available by the company during the internship period was using a MATLAB version older than the one used to develop the RL agent which did not support the generation of the C code for the *RL agent* block. The adapted methodology successfully leveraged the capabilities of the existing environment while ensuring the VTD's functionality was adequately tested.

Assumptions and core idea

- **Agent Dependency**: The car environment does not inherently require the RL agent to function; however, the agent's outputs are essential for simulating

driver actions dynamically.

- **Execution Platform**: Since the RL agent could not run in real-time directly on the HiL system, an external machine capable of executing the agent was utilized to generate the required outputs.

Workaround workflow

1. Execution of RL Agent on a Supported Machine:
   - The RL agent was executed in a MATLAB environment capable of running its computations.
   - Outputs from the RL agent, representing the vehicle's state transitions and actions, were recorded at each simulation time step.
2. Output Storage: The agent's outputs were stored in a structured format, specifically a *.mat* file. This file served as a precomputed dataset for the testing environment.
3. Integration with HiL Environment: a **proxy agent** was implemented within the HiL environment. This proxy agent mimicked the behavior of the RL agent by loading the *.mat* file containing the precomputed outputs and sequentially providing outputs to the environment, step-by-step, as if the RL agent was operating in real-time. The proxy agent retrieved the next state from the matrix at each time step, ensuring seamless integration and accurate representation of the RL agent's decision-making.

The described approach is not optimal since it does not directly employs the trained network into the testing framework, that was one of the main goals of this work, being a trade-off necessary for completing in time the preliminary tests for validating the core idea behind this project. However, the storage of the action series can be have some benefits instead, like the portability of the same test patterns over multiple platforms and vehicles without any problem of compatibility.

The next section will report the results of the first integration tests by using the most advanced VTD agent developed, even if not integrated with the forecasted method.

3.4 Preliminary test results

During September and October 2024, preliminary tests involving the VTD were conducted, yielding promising results. These tests primarily aimed to evaluate the integration of the HDC functionality alongside other ADAS features and broader vehicle functionalities, as detailed in Table 2.1. The outcomes demonstrate the VTD's potential to enhance testing efficiency and accuracy through automation and optimization. Key observations from these tests include the following:

1. **Reduction in Test Design Effort**: The system significantly reduces the workload for test engineers by automating the creation of the core components of test cases. Specifically, the time spent designing vehicle maneuvers has been reduced to near zero, allowing engineers to focus on higher-level analysis and validation tasks. This improvement streamlines the testing process and minimizes manual intervention.

   Additionally, automation facilitates rapid scalability. New scenarios or system updates can be integrated into the test suite with minimal effort, enabling the testing process to keep pace with fast development cycles. This adaptability is particularly beneficial in contexts where frequent updates or feature additions require continuous validation

2. **Comprehensive State Coverage**: The VTD demonstrates the ability to encompass the full range of possible vehicle states within a limited number of iterations. This ensures more thorough testing of system functionality compared to the manual methodology, which was constrained to exploring only a subset of the possible states. This increased coverage enhances the reliability and robustness of the tested features.

   By achieving near-complete state coverage, the system is better equipped to identify edge cases and rare conditions that could otherwise go undetected in traditional testing approaches. This increased coverage not only improves the reliability and robustness of the tested features but also provides greater confidence in the overall system performance under diverse conditions.

3. **Output Consistency**: As anticipated, the VTD demonstrates the ability to produce consecutive outputs with a mean Euclidean distance of approximately 2.2. While this result falls slightly short of the target value of 1.5, it indicates a promising level of stability and consistency. In practical terms, this means that the integrated RL agent generates consecutive states with an average separation of 2 in the state space. The distribution of distances between consecutive generated states is illustrated in Figure 3.13.

This distribution reflects the system's ability to strike a balance between two competing objectives: introducing sufficient variability to stress-test the system and the functionality logic itself by altering multiple states simultaneously, while avoiding excessive randomization that could diminish the relevance or interpretability of the test scenarios. By maintaining a controlled level of variation, the VTD ensures that the generated states effectively challenge the functionality under test without deviating into non-representative or overly chaotic conditions.

Further analysis of the distribution reveals that the majority of consecutive outputs remain within a reasonable distance range, with fewer instances of extreme deviations. This suggests that the RL agent's policy is capable of learning a structured exploration strategy, ensuring that state transitions are meaningful and contribute to the overall test coverage.

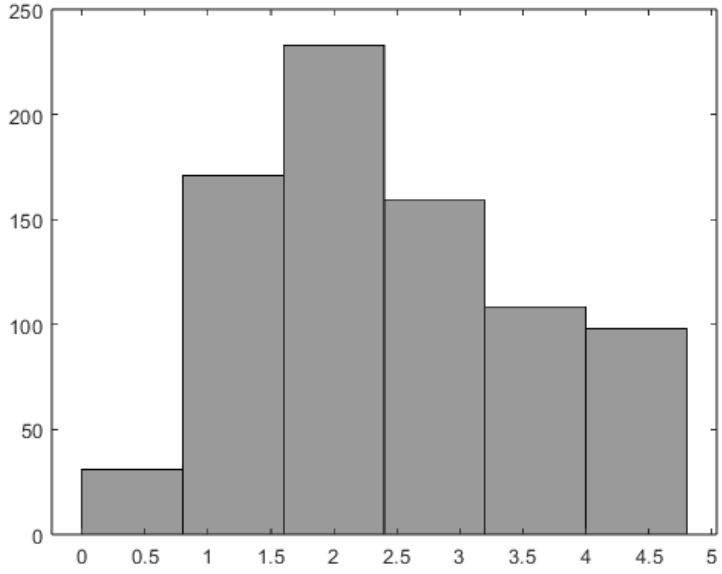

Figure 3.13: Example of distribution of Euclidean distance between generated states.

4. **Efficiency in Test Execution**: The VTD demonstrates a significant improvement in test execution efficiency compared to manual or semi-automated methods previously employed. As shown in Figure 3.14, the VTD is capable of executing approximately two full passes of the states matrix within a single 20-hour cycle, without interruptions in test generation or execution. In contrast, the manual approach is approximately 80% slower and constrained by human

working hours, typically limited to an 8-hour workday. Over the same three-day period, the manual methodology achieves only a fraction of the testing output produced by the VTD.

This high degree of automation in test case generation not only eliminates downtime but also ensures that tests are performed in an optimized sequence. This sequencing enables comprehensive coverage of the entire test matrix in a considerably shorter time frame. Moreover, the consistent and uninterrupted operation of the VTD minimizes human-induced variability and errors, which are more likely to occur in manual or semi-automated processes.

Under the manual methodology, the limited number of tests executed within the same time-frame often resulted in gaps in state coverage, reducing the likelihood of identifying certain edge cases or rare conditions. In contrast, the VTD's approach significantly increases the breadth of testing, which is particularly beneficial in uncovering subtle or infrequent software bugs.

Furthermore, given a fixed number of test cases, the VTD-based methodology is demonstrably more effective at identifying software bugs in less time. This capability is critical in scenarios where rapid identification of critical issues is essential, such as in pre-production phases or during tight development cycles.

By accelerating the testing process while ensuring higher coverage and reliability, the VTD reduces the overall time required for fully deploy vehicle functionalities. This enhanced efficiency translates directly into cost savings, resource optimization, and improved product quality, thus meaning that the VTD has the potentialities to become a pivoting tool in automated vehicle testing.

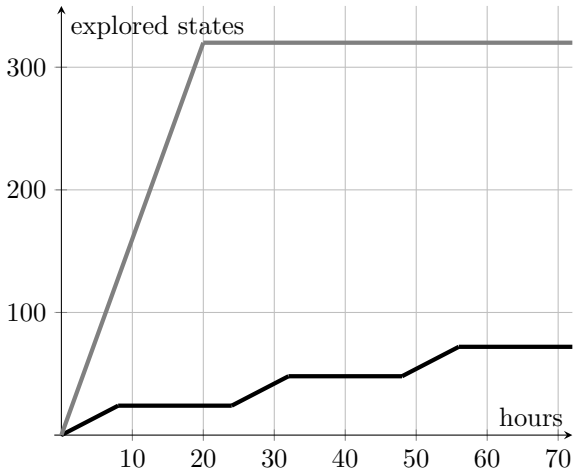

Figure 3.14: Comparison between number of tests performed.

Despite these encouraging results, the total number of tests performed at the time of writing was insufficient to fully exploit the system's potential. This limitation was primarily due to time constraints. Nevertheless, the findings should be interpreted as a proof of concept, validating the feasibility and benefits of the VTD-based approach.

To fully realize the potential of the developed system, additional tests are already scheduled. These future efforts will aim to evaluate the VTD's performance comprehensively and determine whether it can be integrated, with few modifications, as a standard tool within the company's testing framework. This next phase of testing will also explore the system's scalability, robustness, and adaptability to diverse testing scenarios, ensuring it meets the rigorous demands of industrial applications.

Chapter 4

Conclusions

Automatic testing is widely adopted in the automotive industry to streamline and accelerate labor-intensive processes, such as testing and validating the integration of electronic systems within complex vehicle architectures. However, many of these tasks still require human involvement to create test patterns and define testing logic. The introduction of a system capable of autonomously generating test patterns in a reliable and cost-effective manner would significantly enhance integration testing and validation by enabling the rapid and consistent reproduction of a broad range of test scenarios.

The project, as requested by the company, aimed to develop such a system to validate the integration of ECUs used in ADAS functionalities. This thesis documents resumes the candidate's work, conducted during an internship at the company, guided by both the academic tutor and the industry supervisor.

Through significant collaborative effort and after extensive research into state-of-the-art technologies, the system developed—named **Virtual Test Driver** (VTD)—represents a shift away from traditional testing methods by employing neural networks, specifically through the use of **reinforcement learning** as learning strategy, a novel and less conventional approach compared to the widely used data-driven methods.

The methodologies and algorithms required to build the VTD are thoroughly documented in this work, including the MATLAB scripts and Simulink models that must operate in tandem to achieve the desired outcome. Special focus is placed on selecting the appropriate **RL agent type** and designing an effective **reward function**, underscoring the robustness and potential of this approach. The VTD successfully generates a sequence of coordinate arrays to achieve comprehensive validation across all test cases specified in the coverage matrix.

Finally, the model has been successfully integrated into the company simulative environment. The first performed tests shown the validity of the core idea behind this

project. These results confirm that, with a carefully crafted design, neural networks can be effectively employed for ADAS integration purposes, both in testing and validation, offering substantial promise for future applications. Moreover, the way this system is integrated into the company testing framework allows the engineers to not only detect the software bugs and fault about the integrated ECUs under test, but also to highlight and resolve bugs that may show up over the HiL environment and for finding performance lacks about the software implemented on the ECUs under test.

The deep meaning of this project was to gain hands-on experience with a technology of vast potential. The know-how acquired throughout this journey—both for the company and for myself—is a valuable asset, unlocking future opportunities and adding depth to collective expertise. Furthermore, the results achieved thus far are encouraging and reveal promising possibilities.

4.1 Future works

The developed system is still far from perfect, there is still plenty of room for improvement, first of all the design of the reward function, along with the the choice of the RL agent to employ. Moreover, this work must be intended to serve as the first component in a larger framework. Indeed, there is further potential to enhance automation in the ADAS validation sector. For example, *Factors* and *Levels* could be generated automatically by a *seq2seq* network [18], capable of producing structured outputs, such as tables, from textual input, upon the arrival of new test documentation (see Section 2.1.1). Additionally, other aspects of ADAS testing and validation can be further automated, up to the generation and submission of test scenarios to analyze vehicle behavior during test execution.

There were also the intention of creating a framework able to semi-automatize the creation of the agents. This would be useful for the company to create fully working and trained agent to use when a new functionality, that involves different Factors and Levels from the ones described for this case study, needs to be tested. By carefully design such a system the company could design new agents without the involvement of a neural network specialist.

The idea is to create an environment in which the operator only inserts the Factors and Levels of the new project. Then the script automatically trains the agent by fine-tuning the hyperparameters depending on the characteristics of the new Factors and Levels. Once find the best one, the reward function is supposed to never change.

This can surely be done within few weeks of work, starting by modifying the scripts used to train the VTD agents seen in this work, by creating a more solid base and a GUI (Graphic User Interface). However, before doing this, much more experience on the proper agent type to use and about the fine-tuning of hyperparameters in needed.

Once the agent have been trained, the user could find the trained agent into the specified folder and could also be the possibility to automatically create one or more matrices containing the agent steps for off-line agent development (see Section 3.3.3). Only in this way the system developed in this work could become scalable and effectively be integrated within the company testing methodologies.

Appendix A

Overview on Neural Networks

A.1 Neural Network as a tool

Neural networks (NNs), inspired by the structure and function of the human brain, are a class of machine learning models designed to recognize patterns and relationships in data. They consist of layers of interconnected nodes, or "neurons", that process input data and generate outputs. Each connection between neurons is associated with a weight, and through training, these weights are adjusted to minimize the error in predictions. In each node, the neural network performs a series of operations, typically involving sums and multiplications. The common operation at each node is:

$$input \cdot weight + bias = output \tag{A.1}$$

This iterative process allows neural networks to model complex, nonlinear relationships, making them highly versatile for a wide range of tasks. The higher the absolute value of a weight w, the higher the correlation among the input and the neuron. Input x may not strictly be a variable fed by the input layer, but even the output of a previous hidden layer. In the latter case, a high absolute value of weight w usually means that those neurons have strong correlation and they correspond to the same pattern. Unit bias b is a measure of how easy is to activate a hidden unit. Bias is usually negative: the more it is negative, the higher the linear combination has to be in order to activate the neuron.

One of the most important capabilities of NNs is their ability to uncover relationships between inputs and outputs in data collected from processes that are difficult to describe using traditional physical laws. In fact, neural networks are high-capacity models capable of approximating highly nonlinear functions.

The architecture of a neural network is composed of three main types of layers:

- Input layer: it is needed to feed the network with input data.

- Hidden layer: it is made of several intermediate neurons. A hidden layer is capable of introducing non-linearity among inputs and outputs and one only hidden layer would be capable of creating any non-linear function. However, introducing more than one hidden layer allows to decrease the total number of neurons in a network obtaining the same result in terms of function complexity.

- Output layer: it provides the output of the NN, so its predictions. The number of neurons in this layer is equal to the total number of outputs.

Each layer consists of neurons, or perceptrons, and each neuron is connected to every neuron in the preceding layer, but not to other neurons within the same layer. Connecting neurons within the same hidden layer would not introduce any new information, as they would all be receiving the same inputs. As a result, the network would fail to learn more complex patterns in the data. The output of each neuron serves as input for the neurons in the subsequent layer. The greater the number of neurons and layers, the higher the number of parameters in the network, and consequently, the greater its capacity to model complex functions.

Since the goal of neural networks is to approximate the function represented by the training data and map inputs to the correct outputs, the relationships between the layers can be represented as a linear mapping, as shown in the simple example depicted in Figure A.1, where a NN with only three layers is shown:

$$x^{(1)} = A_1 x \tag{A.2}$$

$$x^{(2)} = A_2 x^{(1)} \tag{A.3}$$

$$y = A_3 x^{(2)}. \tag{A.4}$$

This forms a compositional structure so that the mapping between input and output can be represented as

$$y = A_3 A_2 A_1 x, \tag{A.5}$$

which becomes

$$y = A_M A_{M-1} \dots A_3 A_2 A_1 x, \tag{A.6}$$

when dealing with an architecture scaled up to M layers. In doing so, the mapping must generate M distinct matrices that give the best mapping. It should be noted that linear mappings, even with a compositional structure, can only produce a limited range of functional responses due to the *limitations of the linearity*.

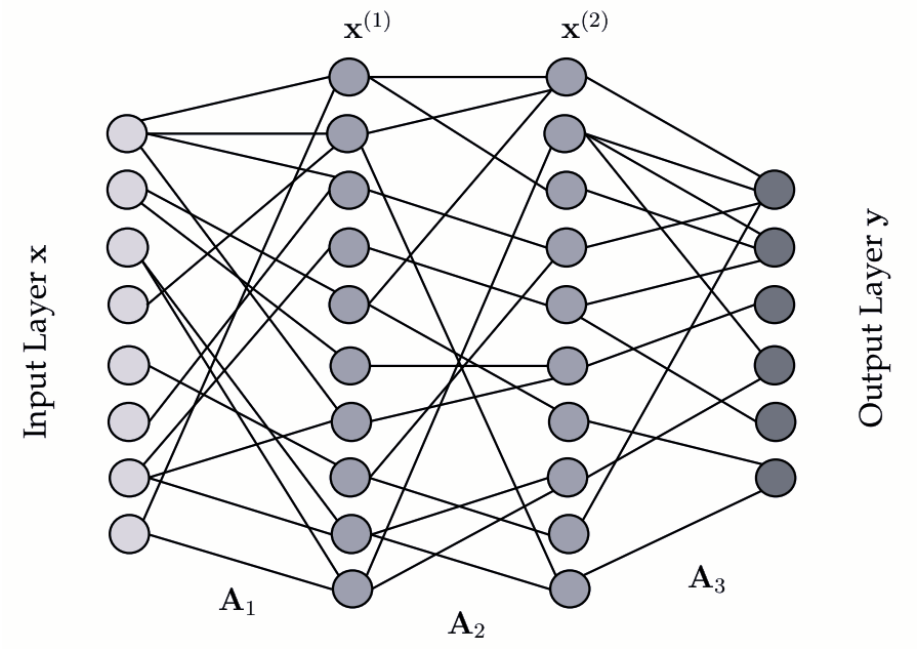

Figure A.1: Illustration of a neural network architecture. Source: [8]

A.2 Introduction of non-linearity

To overcome the limitations of purely linear transformations, it is essential to introduce the concept of **activation functions**. Without activation functions, as mentioned earlier, NNs would essentially consist of a series of linear transformations, where the final output would be a linear combination of the inputs. However, many relationships in the real world are non-linear, and a neural network composed only of linear layers would be unable to capture these complexities.

Activation functions introduce non-linearity into the network, enabling it to model more complex relationships between inputs and outputs. This allows the network to learn and represent non-linear patterns, making it capable of tackling real-world problems such as image recognition, natural language processing, and time-series forecasting. For instance, in deep neural networks used for image recognition, the initial layers may learn simple features like edges or corners, while subsequent layers can combine these features to represent more complex structures. This hierarchical learning of features would be impossible without activation functions.

The most commonly used activation functions in neural networks are:

ReLu (Rectified Linear Unit) Defined as

$$f(x) = \max(0, x), \tag{A.7}$$

(Figure A.2), ReLU is simple and computationally efficient, and it helps mitigate the vanishing gradient problem. However, it can lead to the dying ReLU problem, where neurons stop learning because they always output zero for negative inputs.

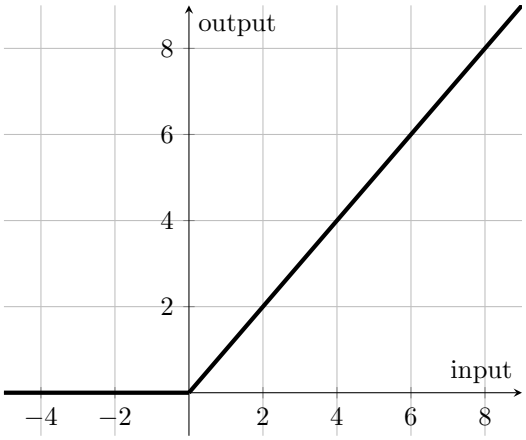

Figure A.2: ReLU activation function, $f(x) = \max(0, x)$.

Sigmoid Defined as

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{A.8}$$

(Figure A.3), the sigmoid function produces outputs between 0 and 1, making it suitable for binary classification problems. However, it suffers from the vanishing gradient problem, as gradients become very small for large or small input values.

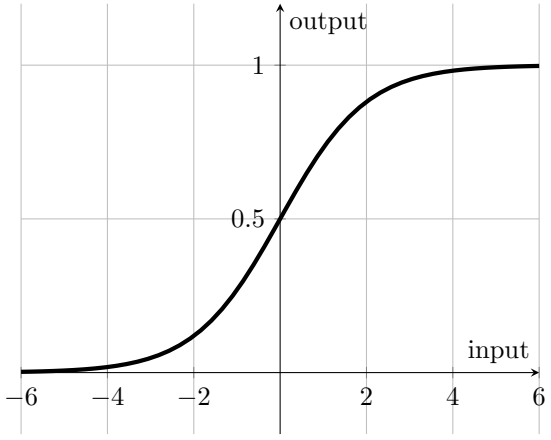

Figure A.3: Sigmoid activation function, $f(x) = \frac{1}{1+e^{-x}}$.

Leaky ReLu This is a variation of ReLU with the formula:

$$\begin{cases} f(x) = x \to x > 0 \\ f(x) = \alpha x \to x < 0 \end{cases} \tag{A.9}$$

where α is a small constant (usually around 0.01) that introduces a small slope for negative values (Figre A.4). This addresses the dying ReLU problem by allowing a small gradient to flow for negative inputs.

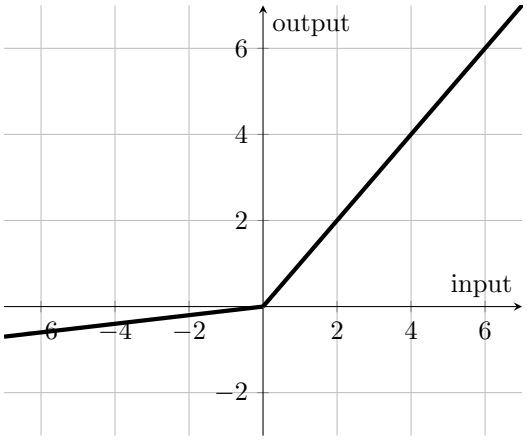

Figure A.4: Leaky ReLU activation function.

Tanh (Hyperbolic Tangent) Define as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{A.10}$$

(Figure A.5), the Tanh function produces outputs between -1 and 1. It provides better convergence than sigmoid but still suffers from the vanishing gradient problem.

Softmax This activation function is typically used in the output layer for multi-class classification problems. The formula is:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{A.11}$$

Softmax generates a probability distribution over multiple classes, making it ideal for classification tasks where the output needs to be interpreted as probabilities.

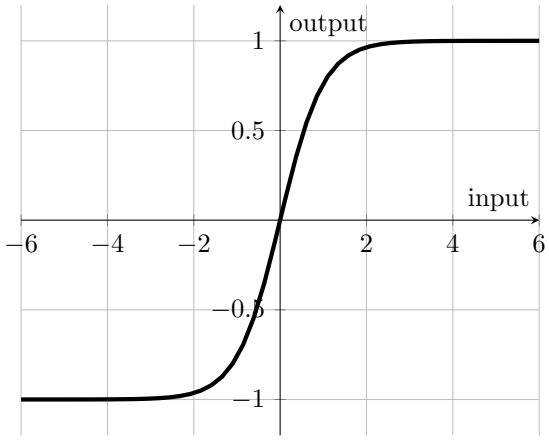

Figure A.5: Tanh activation function, $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Nonlinear activation functions enable a richer set of functional responses compared to their linear counterparts. This allows neural networks to capture more complex relationships between inputs and outputs. The connections between layers in such networks are typically represented as follows:

$$x^{(1)} = f_1(A_1, x) \tag{A.12}$$

$$x^{(2)} = f_2(A_2, x^{(1)}) \tag{A.13}$$

$$y = f_3(A_3 x^{(2)}). \tag{A.14}$$

When mapping the data from input to output over M layers, the general form of the equation becomes:

$$y = f_m(A_M \dots f_2(A_2, f_1(A_1, x)) \dots). \tag{A.15}$$

This nested representation illustrates how the input x is transformed progressively through multiple layers, each applying a nonlinear activation function f[1], allowing the model to learn more complex, hierarchical features.

Activation functions should not be confused with **pooling**, a technique primarily used in Convolutional Neural Networks (CNNs). Pooling functions are designed to reduce the spatial dimensions (width and height) of feature maps, thereby decreasing the number of parameters and the computational load. Additionally, pooling enhances the model's robustness against variations in the data, such as translations or distortions.

The most common pooling techniques are:

[1]Note that different nonlinear functions $f_j(\cdot)$ have been used between layers. Often a single function is used; however, there is no constraint that this is necessary.

1. Max Pooling: Selects the maximum value from a specific region of the input (Figure A.6), capturing the most prominent feature in that region.

2. Average Pooling: Computes the average of the values in a region of the input (Figure A.6), providing a smoother representation.

While activation functions introduce non-linearity into the model, allowing it to capture complex patterns, pooling focuses on simplifying the representation by retaining essential features and reducing noise, which is crucial for tasks like image recognition.

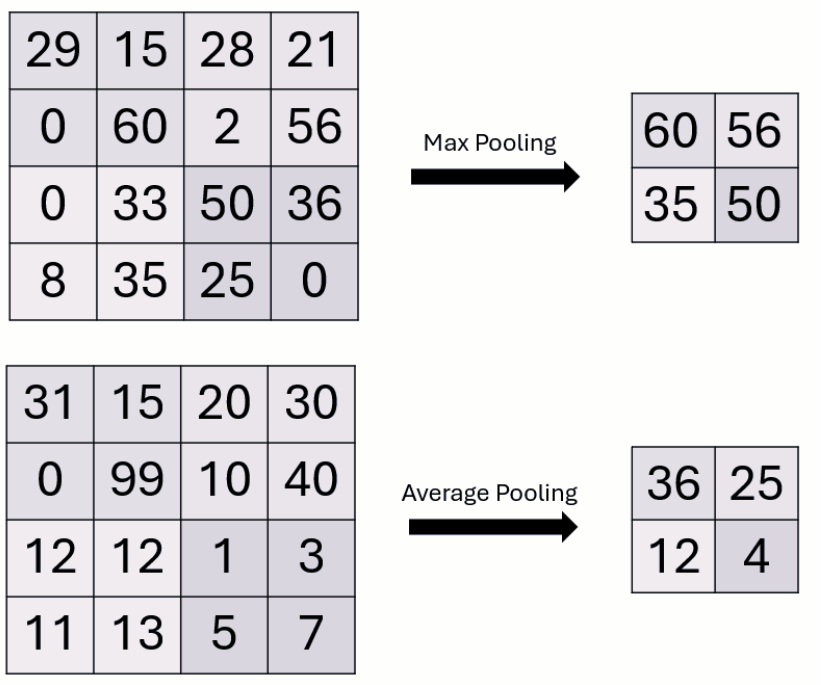

Figure A.6: Examples of max and average pooling

A.3 Deep Learning itself

Deep Learning is a sub-field of machine learning that focuses on neural networks with many layers, known as deep neural networks (DNNs). The increased depth of these networks allows them to automatically learn hierarchical features from data. In essence, Deep Learning revolves around developing and applying DNNs, as well as settle the techniques used to train these networks effectively. This includes not only designing various neural network architectures, such as multilayer perceptrons

(MLPs) or convolutional neural networks (CNNs), but also advancing learning techniques to fully leverage the potential of these architectures.

The central method for training DNNs is backpropagation, which computes gradients with respect to the network's weights using the chain rule. During training, the model calculates the error between predictions and true targets, and through a backward pass, updates the weights to minimize this error. The backpropagation algorithm takes advantage of the compositional nature of neural networks to frame an optimization problem that determines the optimal weights. It provides a foundation for gradient-based optimization methods, such as gradient descent.

Another key method for training DNNs is Stochastic Gradient Descent (SGD). While backpropagation efficiently computes the gradient of the objective function, SGD speeds up the process of finding the optimal weights by using small random subsets of the data (batches) at each step. Once the gradient is computed, optimization algorithms like SGD, Adam, or RMSprop are used to update the weights. These algorithms control the weight adjustments and are essential for ensuring fast and stable convergence during training.

Due to the high capacity of DNNs, they are prone to overfitting—when the model becomes too closely tailored to the training data and fails to generalize well to new data. To mitigate overfitting, various regularization techniques are employed. Overfitting occurs when the network memorizes the training data, preventing it from learning generalized patterns necessary for tasks such as classification. On the other hand, underfitting happens when the model is too simplistic and fails to capture the underlying structure of the data. Common regularization techniques include L2 regularization, dropout (which randomly disables neurons during training to prevent reliance on specific neurons), and early stopping (which halts training when overfitting begins to occur).

Other widely used techniques to improve training efficiency include batch normalization, transfer learning, and data augmentation. These methods, along with ongoing research in both academic and private sectors, have significantly contributed to the growth of Deep Learning. The field continues to expand rapidly, driving advancements in numerous applications and pushing the boundaries of what deep neural networks can achieve.

Appendix B

Overview on LSTM networks

B.1 Introduction

Long Short-Term Memory (LSTM) networks are a specialized form of Recurrent Neural Networks (RNNs). Traditional neural networks lack the ability to interpret current events based on past information. RNNs address this limitation by incorporating loops in their architecture, allowing information to persist. Unlike feed-forward neural networks, which propagate information in a single direction, RNNs include feedback connections, enabling information flow between neurons in the same layer or from higher layers to lower layers. As shown in Figure B.1, the chain-like structure of RNNs makes them naturally suited for processing sequences and lists, making them ideal for tasks involving sequential data.

LSTM networks, in particular, are designed to handle tasks where the order of data is crucial. They overcome the limitations of traditional RNNs, which struggle with capturing long-term dependencies due to issues such as the vanishing and exploding gradient problems during training, during which, as the time instants considered increase, the product chain determined by backpropagation through time tends to zero or tends to extremely large values. In the first case, we have a vanishing gradient, in the second case an exploding gradient. These problems occur because RNNs use backpropagation through time (BPTT), which can fail to preserve important information over long sequences, reducing the network's ability to learn effectively from past data.

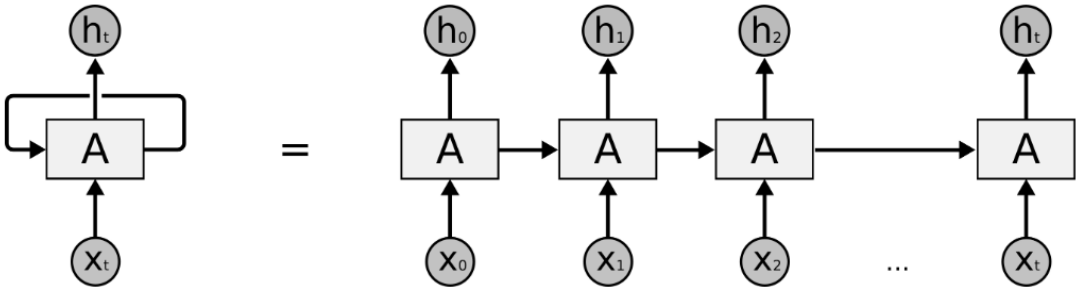

Figure B.1: Rolled vs. unrolled recurrent neural network.

B.2 Information Flow

LSTMs, introduced by Hochreiter and Schmidhuber in 1997, were designed to overcome these challenges by introducing a unique architecture (see Figure B.2) composed of memory cells and a set of gates that control the flow of information. This gating mechanism allows LSTMs to decide which information to retain, discard, or update, ensuring that relevant information is passed along through time.

An LSTM unit receives three vectors as input: two internal vectors generated by the LSTM at the previous time step $t-1$ (the cell state C and the hidden state H), and an external input vector X, which is provided at the current time step t. Using these three vectors, the LSTM regulates the internal information flow through its gates, updating both the cell state and the hidden state. The cell state C serves as long-term memory, while the hidden state H functions as short-term memory. The LSTM uses the recent past (short-term memory H) and new input data (X) to update the long-term memory (cell state C). Finally, the long-term memory C is used to update the short-term memory H, which is also the output of the LSTM at time step t. This output represents the behavior of the LSTM and is used for performing specific tasks, serving as the basis for assessing the performance of the LSTM.

B.3 Gates

From Figure B.2, the three main components of an LSTM, known as gates, are clearly visible: the forget gate, input gate, and output gate. These gates act as information selectors. Their role is to create selector vectors using the sigmoid (A.3) and tanh (Figure A.5) functions. The sigmoid function produces values between 0 and 1, while

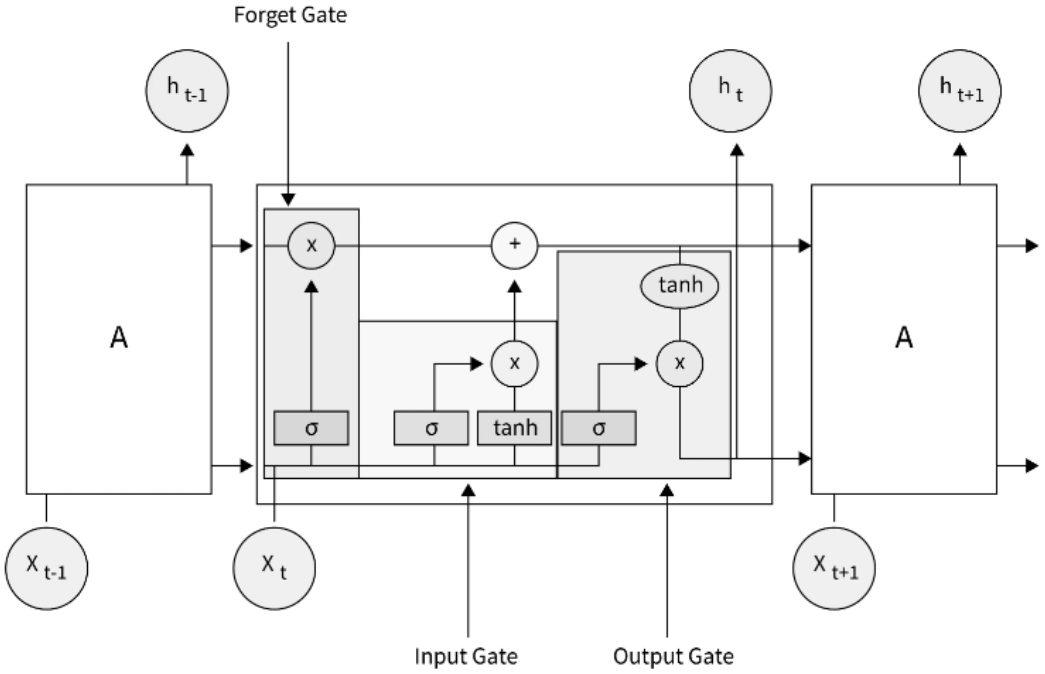

Figure B.2: LSTM network acrchitecture.

the tanh function generates values between -1 and 1, with most values near these extremes. These selector vectors are then multiplied by other vectors coming from previous layers to control the flow of information.

At any given time step t, an LSTM receives an input vector x_t along with the hidden state H_{t-1} and the cell state C_{T-1}, both of which were determined at the previous time step $t-1$. The first operation in the LSTM is performed by the **forget gate**, which decides—based on X_t and H_{t-1} —which parts of the cell state C_{T-1} should be discarded. This decision is encoded in a selector vector, which is then multiplied by the cell state vector, effectively removing unnecessary information.

After some information has been removed from the cell state, the next step is to add new information. This is handled by the **input gate**, which consists of two components: the input gate and the candidate memory. These two components generate a candidate vector, representing new information that may be added to the cell state. The multiplication of the candidate vector and the selector vector determines what new information will be integrated into the cell state, updating it accordingly.

Finally, the updated cell state is used by the **output gate** to determine the output

of the LSTM at time step t and the hidden state H_t, which will be passed to the next time step $t + 1$. The output generation also involves a multiplication between a selector vector and a candidate vector. However, in this case, the candidate vector is derived from the cell state itself by applying the tanh function, which normalizes its values between -1 and 1. Multiplying this with the selector vector, whose values range from 0 to 1, produces a hidden state with values between -1 and 1, which helps maintain the stability of the network over time.

B.4 Uses and Comparisons

Long Short-Term Memory (LSTM) networks are often compared with other types of Recurrent Neural Networks (RNNs) designed to address similar challenges, particularly the retention of long-term dependencies. One of the most prominent alternatives is the Gated Recurrent Unit (GRU) [17]. Both LSTMs and GRUs utilize gating mechanisms to regulate the flow of information through the network and mitigate the vanishing gradient problem. However, they differ in terms of structural complexity, computational efficiency, and, in some cases, performance across various tasks.

The choice between LSTM and GRU largely depends on the specific requirements of the task at hand. When capturing long-term dependencies is crucial and computational resources are not a limiting factor, LSTMs may be the preferred option due to their more sophisticated memory management. On the other hand, GRUs, with their simpler architecture, might be more appropriate for tasks where computational efficiency is critical or where shorter sequences still allow for strong performance. While many other RNN variants have been proposed in recent years, LSTMs and GRUs remain the most widely adopted due to their effectiveness in handling long-term dependencies.

Regardless of the chosen architecture, LSTM networks have proven to be highly effective in tasks where understanding context over time is essential, such as natural language processing (NLP), speech recognition, time-series forecasting, and other applications.

Bibliography

[1] Jessica Hopkins. (May 15, 2024). "What is an Electronic Control Unit in Automotive Systems?". Total Phase. https://www.totalphase.com/blog/2024/05/what-is-electronic-control-unit-automotive-systems/.

[2] (January 3, 2023). "Sensor technologies in the modern vehicle". Arrow.com. https://www.arrow.com/en/research-and-events/articles/sensor-technologies-in-the-modern-vehicle.

[3] Adam Hayes. (July 25, 2024). "What Is Vertical Integration?". Investopedia.com. https://www.investopedia.com/terms/v/verticalintegration.asp.

[4] Robert Fey. (December 20, 2022). "Introduction to Automotive Testing Terms". Synopsys.com. https://www.synopsys.com/blogs/chip-design/automotive-testing-terms.html.

[5] W.H. Janssen. "Routeplanning en-geleiding: Een literatuurstudie". 1979.

[6] Dong-Fan Xie et al. "A data-driven lane-changing model based on deep learning". In: Transportation research. Part C, Emerging technologies 106 (2019), pp. 41–60. issn: 0968-090X.

[7] Yang Zheng and John Hansen. "Lane-Change Detection From Steering Signal Using Spectral Segmentation and Learning-Based Classification". In: IEEE Transactions on Intelligent Vehicles (May 2017), pp. 1–1. doi: 10.1109/TIV.2017.2708600.

[8] S. Brunton and J. N. Kutz. (2021). "Data Driven Science & Engineering". Cambridge University Press.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. 2016. MIT Press. http://www.deeplearningbook.org.

[10] T. Pallacci, N. Mimmo, P. Sessa, and R. Rabbeni. 2023. "A Deep-Learning Model of Virtual Test Drivers"

[11] Silver, D., Huang, A., Maddison, C. *et al.*.(January 27, 2016). "Mastering the game of Go with deep neural". https://doi.org/10.1038/nature16961

[12] Mnih, V., Kavukcuoglu, K., Silver, D. *et al.*. (February 25, 2015). "Human-level control through deep reinforcement learning". https://doi.org/10.1038/nature14236

[13] Tang C., Abbatematteo B. *et al.*. (August 7, 2024). "Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes". https://www.arxiv.org/abs/2408.03539

[14] Duncan Brady. (Mar 06, 2024). "What Is Hill Descent Control? How This "Downhill Cruise Control" Works". https://www.motortrend.com/features/hill-descent-control-explained/

[15] Mnih V., Volodymyr, Silver D. *et al.* (February 4, 2016). "Asynchronous Methods for Deep Reinforcement Learning." https://arxiv.org/abs/1602.01783.

[16] Schulman J., Philipp M. *et al.* (October 20, 2018). "High-Dimensional Continuous Control Using Generalized Advantage Estimation". https://arxiv.org/abs/1506.02438.

[17] Cho K. *et al.* (September 3, 2014), "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". https://arxiv.org/pdf/1406.1078v3.

[18] Li T., Wang Z., *et al.* (Mat 31, 2023). "A Sequence-to-Sequence&Set Model for Text-to-Table Generation". https://doi.org/10.48550/arXiv.2306.00137.