Self-learning robot navigation with deep reinforcement learning techniques

Master's Thesis in Artificial Intelligence

Author:

Borja Pintos Gómez de las Heras

Director:

Rafael Martínez Tomás

Co-director:

José Manuel Cuadra Troncoso

September 2022

Universidad Nacional de Educación a Distancia (UNED)

Escuela Técnica Superior de Ingeniería Informática

Abstract

The autonomous driving has been always a challenging task. A high number of sensors mounted in the vehicle analyze the surroundings and provide to the autonomous driving algorithm useful information, such as relative distances from the vehicle to the different obstacles. Some robotic paradigms, like the reactive paradigm, uses this sensorial input to directly create an action linked to the actuators. This makes the reactivate paradigm capable to react to unpredictable scenarios with relatively low computational resources. However, they lack a robot motion planning. This can lead to longer and less comfortable trajectories with respect to the hierarchical/deliberative paradigm, which counts with a motion planning module over a predefined horizon. Although a local optimization of the robot trajectory is now possible under static scenarios, the motion planning module comes at a high cost in terms of memory and computational power. The hybrid paradigm combines the reactive and hierarchical/deliberative paradigms to solve even more complex scenarios, such as dynamic scenarios, but the memory and computational resources needed are still high. This work presents the sense-think-act-learn robotic paradigm which aims to inherit the advantages of the reactive, hierarchical/deliberative and hybrid paradigms at a reasonable computational cost. The proposed methodology makes use of reinforcement learning techniques to learn a policy by trial and error, just like the human brain works. On one hand, there is no motion planning module, so that the computational power can be limited like in the reactive paradigm. But on the other hand, a local planification and optimization of the robot trajectory takes place, like in the hierarchical/deliberative and hybrid paradigms. This planification is based on the experience stored during the learning process. Reactions to sensorial inputs are automatically learnt based on well-defined reward functions, which are directly mapped to the safety, legal, comfort and task-oriented requirements of the autonomous driving problem. Since the motion planification is based on the experience, the algorithm proposed is not bound to any embedded model of the vehicle or environment. Instead, the algorithm learns directly from the environment (real or simulated) and therefore it is not affected by uncertainties of embedded models or estimators which try to reproduce the dynamics of the vehicle or robot. Additionally, the policy is learnt automatically. The state-of-the-art algorithms invert many engineering hours to develop a policy or algorithm to fulfil all given requirements, while the method proposed in this work saves these costs and engineering time. Another interesting advantage of the proposed algorithm is the capability to adapt the logic under unknown scenarios. For that, an online learning process is implemented, but the memory and computational power required for that is high.

Keywords: deep reinforcement learning, self-learning, autonomous driving, deep deterministic policy gradient, Q-learning, dynamic environment

Contents

List of figures

List of tables

1. Introduction

This section is divided into two different parts. The first part gives a short overview about the different robotic paradigms and the usability of these paradigms depending on the requirements of the autonomous driving problem to be solved. To solve high complex autonomous driving scenarios where multiple requirements must be fulfilled, a motion planning algorithm is usually mandatory. Thus, the state-of-the-art solutions which make use of a motion planning module are highlighted, also introducing their main drawbacks. The second part briefly introduces the aim of this work and how the method proposed tries to find a solution to the main drawbacks of the state-of-the-art solutions highlighted in the first part.

1.1. Background

The autonomous driving is a domain of constantly research and evolution. One of the things that best represent this evolution are the different robotic paradigms that have emerged over time and continue to emerge today. New robotic paradigms have been created to better solve a specific problem inside the autonomous driving domain. However, there is no paradigm better than the other one. It all depends on the type of problem to be solved. Depending on the problem, one paradigm might fit better than the other ones. A problem in the autonomous driving domain can be defined by the following parts:

- Type of the scenario: open scenario or closed scenario. A closed scenario is defined by spatial boundaries. For example, a vehicle travelling across the city is considered a closed scenario, because the vehicle is limited to the drivable space determined by the road boundaries. A robot inside a room with obstacles is also considered a closed scenario. On the other hand, an open scenario does not have in principle spatial restrictions. An example could be a robot exploring the surface or another planet.
- Type of obstacles: static obstacles or dynamic obstacles. In case of dynamic obstacles, it is also important to define the unpredictability level of these moving obstacles.
- Type of robot: holonomic or non-holonomic robot.
- Requirements to be fulfilled by the robot. These requirements are sorted into 4 different areas:
    - Safety requirements: the main goal is to avoid crashing into obstacles and keep the vehicle inside the road boundaries.
    - Legal requirements: the main goal is not to exceed the maximum road speed limit.
    - Comfort requirements: the main goal is to keep the acceleration and the derivative of the acceleration (jerk) under a well-defined threshold to maximize the comfort feeling.
    - Task-oriented requirements: the main goal is to fulfil some predefined tasks, for example, drive the robot to the destination

pose or minimize the travelling time of the robot from the starting pose to the destination pose.

The algorithm proposed in this work tries to find a solution for the most complex combination of the previous points: a non-holonomic robot interacting in a closed scenario with static and dynamic obstacles, while fulfilling at every moment safety, legal, comfort and task-oriented requirements.

Methods based on the reactive paradigm perform well with holonomic robots under closed scenarios with static obstacles while fulfilling safety and legal requirements. However, non-holonomic robots might be difficult to integrate with the reactive paradigm since they need to plan ahead its movement in order to avoid the obstacles. Moreover, the control based on reactions to certain sensorial input (behaviors) without any local planning can lead to aggressive and long trajectories, making it difficult to integrate comfort or task-oriented requirements. On the other hand, the reactive paradigm demands low computational resources, which enables the usage of non-expensive microcontrollers.

The hybrid paradigm has a planification layer and can better deal with task-oriented requirements. However, the movement of the robot is still based on a reactive layer, making again difficult to integrate comfort requirements, especially for non-holonomic robots. Additionally, the hybrid paradigm demands high computational power and memory.

Finally, the hierarchical/deliberative paradigm has the better approach to find a solution to the autonomous driving problem previously proposed. The hierarchical/deliberative paradigm has a robot trajectory planification stage (motion planning) before the actuation stage (motion control). The motion planning of a robot consists of the generation of a trajectory from the starting pose to the destination pose. This trajectory described by the robot, which consists of a path and velocity profiles (Figure 1.1), must fulfil safety, legal, comfort and task-oriented requirements.

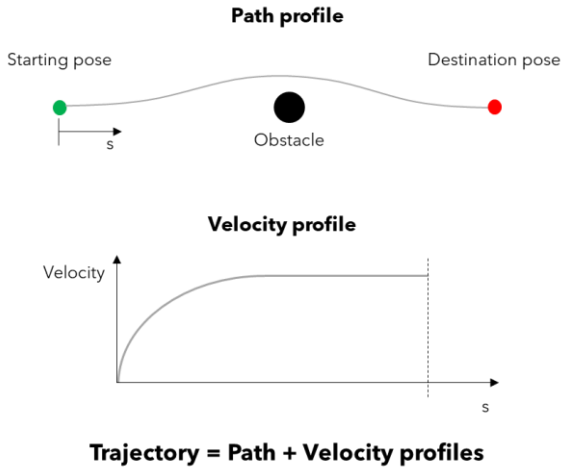


Figure 1.1: Trajectory generation. The motion planning of an algorithm consists not only of generating a path connecting the starting pose with the destination pose, but also assigning a velocity profile to that path profile

Currently, the state-of-the-art of the motion planning algorithms consists of generating an extremely high number of possible trajectories to finally select one of them based on the evaluation of a cost function. This cost function ensures that these well-defined requirements are fulfilled in the selected vehicle trajectory (for example, the cost function penalizes non feasible trajectories or trajectories that collide with obstacles, among many other things). However, the state-of-the-art solutions have three main drawbacks:

- The design of the previously mentioned cost function can be hard and challenging. Combining the safety, legal, comfort and task-oriented requirements into a single cost function and getting the optimal calibration or weighing between these different requirements is a complicated task.
- These algorithms demand a lot of computational resources (generating and evaluating thousands of trajectories every 100 milliseconds requires a high-performance computing unit, which usually costs a lot of money).
- It is difficult to plan feasible trajectories when the vehicle reaches dynamic limitations, such as emergency braking or a driving condition where the vehicle dynamic control system is enabled to prevent the vehicle from skidding and losing the control. Non-feasible trajectories can be rejected by means of the cost function, but it can be quite complicated since models to predict the coefficient of friction between wheels and road or models to predict if the vehicle dynamic system will be enabled within the target trajectory are needed.
- Dependency on embedded vehicles models, used either to reject unfeasible trajectories or to simulate the vehicle dynamics over a horizon to generate feasible trajectories. However, differences between modelling and reality might appear, coming back again to the problem of unfeasible trajectories.

The rise of artificial intelligence has enabled new robotic paradigms offering interesting approaches to overcome the issues of the hierarchical/deliberative paradigm. For example, the paradigm sense-think-act can come up with driving strategies which can successfully handle driving conditions close to the limit. This is possible by analyzing big amount of data (from simulation or real driving conditions) to train a neural network which minimizes some cost function. However, the availability of these data can be limited, and the design of the cost function is still challenging as it was in the hierarchical/deliberative paradigm. The introduction of reinforcement learning techniques allows the definition of another robotic paradigm: sense-think-act-learn. Through the introduction of this new robotic paradigm, we try to overcome the main issues of the sense-think-act paradigm. First, the idea is to generate data on-the-go instead of being forced to have in advance a big database. And secondly, we try to avoid the definition of a single and extreme complex cost function. Instead, we learn the cost function (critic model) based on the definition of multiple but very simple reward functions.

1.2. Aim

This work aims to study reinforcement learning techniques to easily come up with a high mature policy for complex autonomous driving scenarios. The state-of-the-art solutions for the hierarchical/deliberative robotic paradigm spend many engineering hours to develop high mature software so that all well-defined requirements are fulfilled. This work aims to replace these over complicated algorithms by some other simpler method, which automatically learns the policy just by trial and error. Through the definition of simple and multiple reward functions and considering all the huge amount of data delivered by the perception system of a robot or vehicle, the idea is to automatically come up with a policy just by learning it from the environment (either from a simulated or real environment). Another big problem of the state-of-the-art solutions is the generation of feasible trajectories. Usually, these state-of-the-art solutions rely on embedded models of the robot dynamics and its environment to create feasible trajectories. However, differences between modelling and reality always appear, compromising the feasibility of the trajectories generated. The reinforcement learning techniques studied in this work are model-free algorithms, meaning that they will learn a policy exploring directly inside the environment (real or simulated) and therefore considering all dynamic constrains from the robot or vehicle.

The learning process of these reinforcement learning techniques is similar to the one which takes place in our brain and involves trying multiple times and learning from the errors that we make. Within this work it will be highlighted how powerful the reinforcement learning techniques are by setting up some classical autonomous driving scenarios. The main advantage is that a simple definition of reward functions can automatically lead to highly mature policies, fulfilling safety, legal and comfort requirements of complex driving scenarios. Additionally, the policy obtained out of the learning process is not as computationally expensive as the state-of-the-art algorithms and can run on much less expensive computing units.

2. Related work

The related work is split into two sections. First, the three classical robotic paradigms are described, highlighting the most common algorithms of each paradigm. A table at the end of this section summarizes the main advantages and disadvantages of the algorithms presented. The second section introduces the sense-think-act-learn paradigm, where the reinforcement learning techniques applied in this work belong to. It is highlighted how the sense-think-act-learn paradigm finds a solution for the main drawbacks presented in the first section. Additionally, a study of already existing algorithms using reinforcement learning techniques has been described, including all the citations to these works in the bibliography section.

2.1. Classical robotic paradigms

The autonomous driving is a domain where there is not a fixed algorithm that can be applied for every single use case. It all depends on the scenario, robot type and the requirements considered during the autonomous driving navigation (safety, legal, comfort and task-oriented requirements, as defined in section 1).

If we think of motion planning and the previously described requirements, it is not always required to have a motion planning module inside our algorithm. For example, if the requirements to be considered are safety and legal requirements, the reactive paradigm (sense-act) fits well to solve the problem. The artificial potential fields [1] or the partial center of area [2] are good examples of the reactive paradigm. However, the artificial potential fields (Figure 2.1) can get trapped in local minima and the comfort requirements are difficult to integrate. On the other hand, Figure 2.2 shows the partial center of area method, which tries to follow the direction of the center of area. Although the center of area is a very robust and efficient algorithm, it still belongs to the reactive paradigm and lacks a motion planning stage, like the artificial potential field method. If additional requirements apart from safety and legal are to be implemented in the autonomous driving problem, the inclusion of a motion planning is highly convenient. The method proposed in this work can plan and select the most convenient trajectory based on the experience collected by exploring and trying multiple times different combinations to find out the best one.

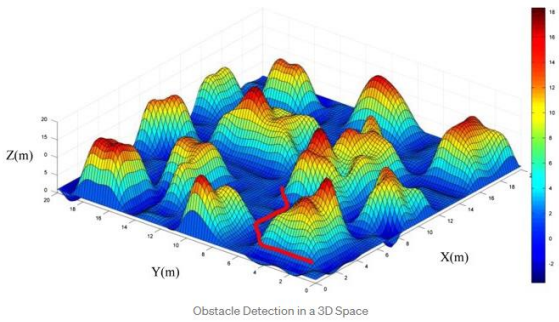


Figure 2.1: Artificial potential fields. The red line corresponds to the trajectory followed by the robot

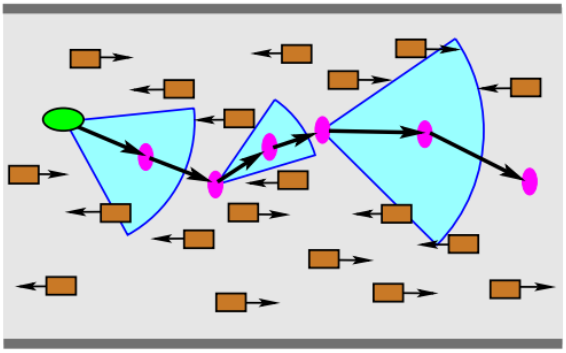


Figure 2.2: Partial center of area method. The magenta spots represent the calculation of the center of area where the robot is safely guided to avoid the obstacles

The reactive paradigm is no longer convenient if comfort or task-oriented requirements, such as minimize the travelling time, are to be integrated along the safety and legal requirements. In this case, a motion planning module fits very well to solve these new requirements. The motion will be planned for a predefined distance and only the best planned trajectory, which fulfils all the given requirements, will be selected. This is, trajectories with low comfort index will be discarded, so the comfort requirements can be also easily integrated. The robotic paradigm which includes a motion planning is the hierarchical/deliberative paradigm (sense-plan-act). Inside this paradigm, the Dijkstra algorithm [3] performs very good at avoiding obstacles and finding the shortest path. However, it is difficult to use with large grids because it is computationally expensive. The A* algorithm [4] was followed by the Dijkstra algorithm to reduce the computational load by using heuristic to find the solution faster (Figure 2.3). However, the Dijkstra and A* algorithms have also an additional and very important disadvantage: they do not consider the robot or vehicle dynamics in the motion planning. This means, the trajectory computed by these algorithms cannot be perfectly followed by non-holonomic robots, such as front steering vehicles. This is a very important problem because the planned trajectory cannot be perfectly followed by the robot and it could eventually collide with obstacles, not fulfilling therefore the safety requirements. The hybrid A* algorithm [5] tried to propose a solution for this problem. Although this method can consider some dynamic constrains, it does not consider all of them. It only considers a maximum and minimum limitation for the curvature of the path travelled by the robot (see Figure 2.4).

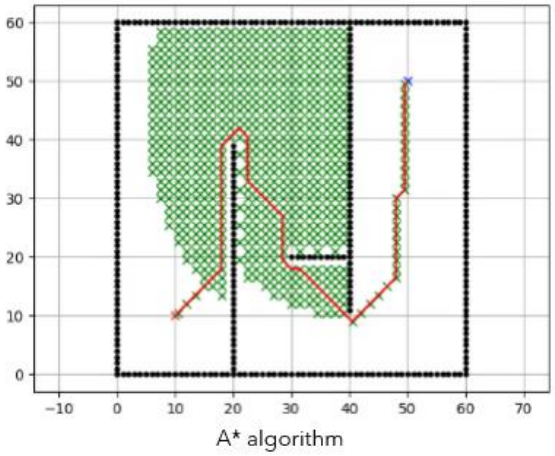


Figure 2.3: Trajectory determination using A algorithm. The red line represents the final trajectory generated by the A* algorithm*

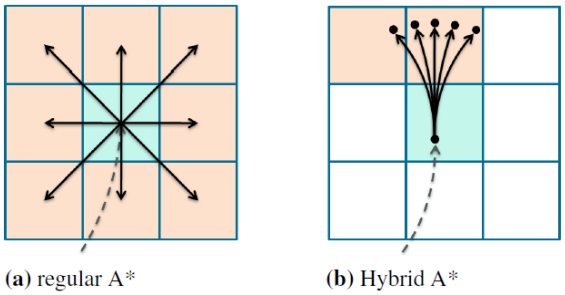


Figure 2.4: Comparison between regular A (a) and hybrid A* algorithms (b). The hybrid A* algorithm generates a path profile with restrictions in the maximum and minimum curvature to respect the dynamics of the robot*

More sophisticated methods were created to fulfil safety, legal, comfort and task-oriented requirements. One of these methods is the lattice planner [6]. Figure 2.5 shows the result of applying the lattice planner to a front steering vehicle. Longitudinal and lateral waypoints are generated from the starting pose to the destination pose. Path profiles are generated between each pair of waypoints, and several velocity profiles will also be assigned for each path profile. From this point on, a cost function will evaluate the best possible trajectory among the generated ones. Non feasible trajectories which does not fulfil dynamic constrains could be generated by the lattice planner algorithm, but the cost function will assign them a poor score leading these trajectories to be discarded. The cost function can now evaluate more aspects than just the curvature of the path, like the hybrid A* method does (for example, trajectories with high values of the curvature's derivative will be also rejected since the steering servo motor has limitations in the maximum angular speed, as implemented in [7]). On the other hand, the cost function can effectively reject non feasible trajectories under scenarios of low velocity, such as parking [8], or scenarios with moderate accelerations [9]. But scenarios close to the limit, like the road-tire friction limits or the activation of the vehicle dynamic system are difficult to consider under this approach. As it will be discussed later, some other approaches like model predictive control are more suitable for planning trajectories under conditions

close to the limit. Regarding the comfort requirements, they are easy to integrate by means of the cost function. However, it is hard to design and parametrize a single cost function valid for every traffic situation, as stated in [5]. We will see in this work that one of the main advantages of the methodology proposed is that the cost function (critic model) is learnt, and it is not necessary to manually design and implement it. Another disadvantage of the lattice planner is that the computational cost is still high in comparison with the reactive paradigm methods, as stated in [10]. If a high number of waypoints are generated, the computational cost will be high since Newton-Raphson iterative method must be applied to each pair of waypoints to create a path profile [6].

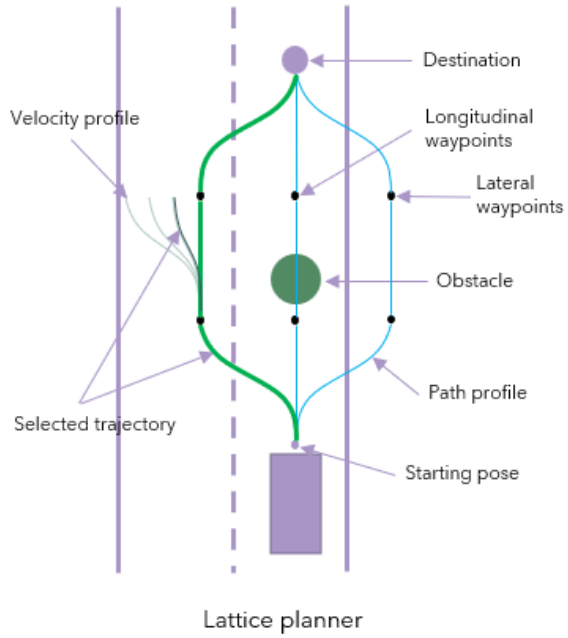


Figure 2.5: Trajectory determination using lattice planner algorithm. Longitudinal and lateral waypoints are generated between the starting and destination poses. Path profiles are generated connecting each pair of waypoints and several velocity profiles are assigned to each path profile. Finally, the cost function evaluates all generated trajectories and select the best one

In order to include into consideration more dynamic constrains and be able to generate feasible trajectories, the model predictive control approach [11] was proposed. Here, a model of the robot is used to generate the path and velocity profiles along a predefined horizon (Figure 2.6). Therefore, the trajectories generated by the model predictive control respects the dynamics given by the model. However, differences between the model and the reality might appear, running again into the problem of non-feasible trajectories. In [12], the parameters of the model are even updated over the prediction horizon under situations close to the limit based on a predictive friction estimate to obtain even more reliable and feasible trajectories. However, we still rely on models or estimators that might have differences with respect to the reality.

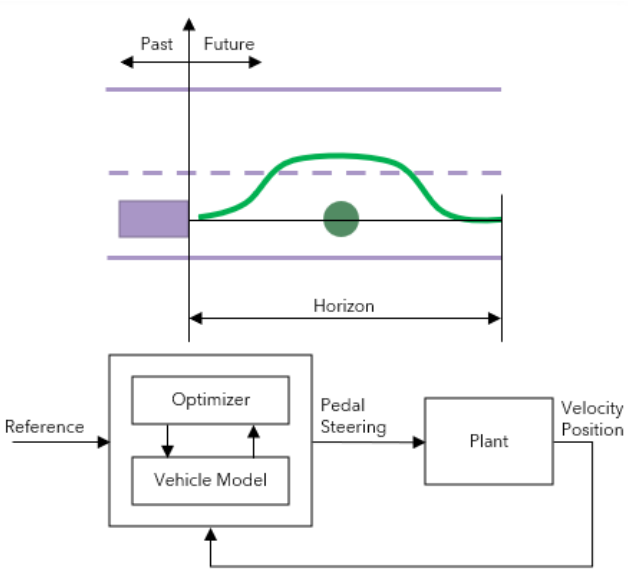


Figure 2.6: Model Predictive Control algorithm. An embedded model simulates the dynamics of the robot over a predefined horizon. The computed trajectory can therefore take into account the robot dynamics, but differences between the model and reality might appear

This mismatch between modelling and reality can be solved with the introduction of artificial intelligence and the analysis of big data. Neural networks used to model the vehicle dynamics and nonlinear model predictive control (NMPC) have been combined to solve the problem of generating feasible trajectories. In [13], the problem of controlling a neural network with NMPC is addressed for the task of motion planning. However, NMPC demands high computational resources. In [14], a solution to this high computational effort is proposed by replacing the nonlinear optimization trajectory planner by a neural network.

Additionally, the artificial intelligence is also extremely interesting to solve motion planning problems under complex driving scenarios (such as multiple dynamic obstacles in scenarios like intersections or highway autopilot). For this purpose, new robotic paradigms have emerged, like the sense-think-act paradigm [15]. One example of this paradigm is the neural network path planner [16], which uses temporal data with recursive neural networks (RNN) to generate feasible trajectories under scenarios with multiple dynamic obstacles. On the other hand, the main disadvantage of all methods featuring neural networks is that we need to count in advance with a large database to train the models, which is not always available.

The method proposed in this work, unlike the model predictive control, is not bound to any model of the vehicle and the environment. Instead, the reinforcement learning algorithm proposed in this work is a model free algorithm, which means that it does not use any model of the vehicle inside its logic. The main benefic is that the algorithm learns directly from the environment (simulated or real), and it is not affected by uncertainties of embedded models or estimators which try to reproduce the dynamics of the vehicle or robot. And unlike the sense-think-act paradigm, it does not require a big database in advance. It generates the data on-the-go, stores these data in a buffer and uses these recorded

experiences to update the policy based on the cumulative reward obtained along these experiences.

The Table 2.1 gathers all the advantages and disadvantages of the main robotic paradigms previously mentioned:

Paradigm	Drawbacks	Benefits
Hierarchical/ deliberative	High computational cost	Local trajectory planning where safety, comfort, legal and task-oriented requirements can be optimized under static environments
	Difficult to plan trajectories under dynamic environments	
	Challenging design of a cost function	
	Difficult to plan feasible trajectories under situations closed to the limit	
Reactive	No local trajectory planning	Low computational cost
	Difficult to integrate comfort requirements if the robot reacts to unpredictable situations	Robot is capable to react to unpredictable situations, even under dynamic environments
Hybrid	High computational cost	Local trajectory planning where safety, legal and task-oriented requirements can be optimized under static and dynamic environments
	Difficult to integrate comfort requirements if the robot reacts to unpredictable situations	Robot is capable to react to unpredictable situations, even under dynamic environments
Neural Networks	Large amount of data must be available	Can learn all dynamics effects
	Challenging design of a cost function	Can solve complex driving scenarios

Table 2.1: Comparison of robotic paradigms

As mentioned before, there is no paradigm better than the other ones, and the selection of one or another depends on the requirements of the problem. However, none of the paradigms listed in the previous table is the best fitting when safety, legal, comfort and task-oriented requirements need to be considered at the same time at a reasonable computational power. The closest ones are the hierarchical/deliberative paradigm and the sense-think-act paradigm (neural networks). But the difficult design of the cost function and the computational power of the hierarchical/deliberative paradigm and the obligation to have big databases of the sense-think-act paradigm makes them challenging to implement. Therefore, the sense-think-act-learn paradigm will be presented in

this work with the aim to fix the disadvantages of the previous paradigms while keeping their advantages.

2.2. Sense-Think-Act-Learn paradigm

The sense-think-act-learn paradigm (Figure 2.7) uses reinforcement learning techniques to generate data on-the-go to train by trial and error and obtain a policy that drives the robot safely [17]. In case of deep reinforcement learning [18], neural networks are trained to perform the autonomous driving navigation. One advantage of this paradigm is that, instead of having to define a single and complex cost function, the reinforcement learning requires to define multiple but very simple reward functions [19] to automatically learn this cost function. These reward functions are directly mapped to safety, legal, comfort and task-oriented requirements.

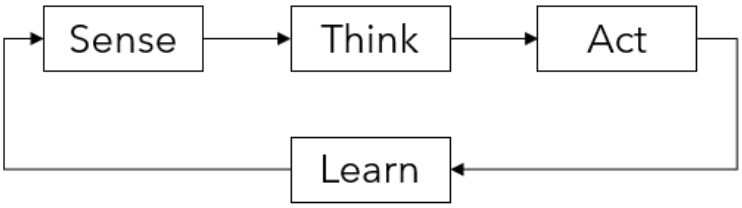


Figure 2.7: Sense-Think-Act-Learn paradigm

To sum up, nowadays the best paradigm to approach a problem where a non-holonomic robot is interacting in a closed scenario with static and dynamic obstacles, while fulfilling at every moment safety, legal, comfort and task-oriented requirements is either the hierarchical/deliberative paradigm or the sense-think-act paradigm. Most of the state-of-the-art solutions for the hierarchical/ deliberative paradigm are based on a motion planning software module implementing lattice planner or model predictive control algorithms. However, as stated in this chapter and in section 1.1, these algorithms present challenges in the design of the cost function, in the interaction with dynamic environments and in the generation of feasible trajectories, apart from the computational consumption. The method proposed in this work tries to improve this situation while keeping the computational consumption low, like in the reactive paradigm. The Table 2.2 gathers all the drawbacks previously mentioned and how the method proposed in this work, based on the robotic paradigm sense-think-act-learn, can propose a solution to them:

Drawbacks	Solution
Challenging design of a cost function	Cost function is learnt instead of designed based on **multiple and simple reward functions**
Difficult to plan trajectories under dynamic environments	Agent can **learn** from **dynamic environments**
Difficult to plan feasible trajectories under situations closed to the limit	Agent is **free from embedded** robot and environment **models** and learns directly from the environment

	(simulated or real), being able to learn the right policy under situations closed to the limit
High computational cost	Low computational cost because there is **no motion planning module**, like in the reactive paradigm
No local trajectory planning	Agent learns a **local trajectory planning based on the knowledge from multiple experiences**
Difficult to integrate comfort requirements if the robot reacts to unpredictable situations	Easily included with **simple reward functions**
Large amount of data must be available	Data is generated **on-the-go** (learning from trial and error)

Table 2.2: Strengths of sense-think-act-learn paradigm

Some works have already investigated the benefits of the reinforcement learning techniques applied to the autonomous driving domain. [20] uses the Q-learning to find the shortest path from the starting pose to the destination pose. Reinforcement learning techniques with continuous actions and states are introduced in [21] with very promising results. Deep deterministic policy gradient (DDPG) was introduced in [22] to learn from a real scenario how to drive and keep the vehicle inside the road boundaries by providing just a single camera image of the road as input. However, this work only implements partial safety requirements (without obstacle avoidance) and does not consider any comfort, legal or task-oriented requirements. Additionally, the perception, motion planning and motion control modules are embedded in the deep neural networks of the deep reinforcement learning algorithm. In [23], the autonomous driving algorithm rely not only on input images of the cameras as in [22], but also on preprocessed data as velocities or relative distances. In [24], comfort requirements are included for lane change maneuvers as simple reward functions penalizing high lateral acceleration values and the Q-learning algorithm is used to control a single action (yaw acceleration) to perform the lane change. In contrast to [22], it separates the perception module from the reinforcement learning task, and the agent consumes already post processed signals from the perception module, such as vehicle velocities or relative distances, instead of images. The algorithm presented in this work tries also to keep the perception module and motion planning and control separated from each other. Images are not directly provided to the agent. Instead, all the relevant information from the perception module is provided to the agent, so that it can learn the best possible trajectory based on the cumulative reward. Additionally, this work tries to improve the previously mentioned algorithms by combining at the same time safety, legal, comfort and task-oriented requirements. For that, the action space will consider not only the angular velocity or steering angle, but also the longitudinal velocity by means of the pedal and brake (multi action space). Dynamic situations will be also validated with this algorithm, such as the overtaking maneuvering or lane keeping when the road is blocked by slow dynamic obstacles (section 5.2.4). Lane keeping with (section 5.2.1) and without obstacles (section 5.2.2) will be also validated.

Besides, it will be proved that this algorithm is also suitable for robots navigating in an open world from a starting pose to a destination pose, fulfilling at the same time the given requirements (section 5.2.4).

The next sections will introduce and prove the effectiveness of the reinforcement learning techniques in cases where safety, legal, comfort and task-oriented requirements must be fulfilled together with the dynamic constrains of the robot used for the autonomous driving navigation. The sense-think-act-learn paradigm can be very useful for the case of autonomous driving applied for passenger vehicles (front steering vehicles). In this case, it is extremely important to fulfil all type of requirements while considering at the same time the vehicle dynamic constrains.

3. Reinforcement learning algorithms

This section gives a short overview about the reinforcement learning methods used for the self-learning robot navigation proposed in this work. The theory of the q-learning and the deep deterministic policy gradient (DDPG) methods are explained, and the pseudocode is introduced at the end of each chapter.

3.1. Q-learning

The q-learning method is a model-free algorithm which learns the best action given a state. It is model free because the q-learning does not use any embedded model of the system dynamics in the algorithm. Instead, the q-learning algorithm learns directly from the environment. Therefore, the algorithm will eventually learn how to optimally deal with the dynamics of the environment by using the information stored in the so-called q-table after the learning process. The q-table evaluates the quality of each action inside a given state. Then, the action with the highest quality score is selected. Following the optimal policy, given by the optimal q-table, will eventually lead to the maximum cumulative reward. The cumulative reward is defined according to Equation 3.1:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{t=0}^{\infty} \gamma^k R_{t+k+1} \qquad \textit{Equation 3.1}$$

The main goal of the q-learning algorithm is therefore to maximize the Equation 3.1, which is the cumulative reward. The factor γ is called the discount factor. A discount factor closes to 1 will take into consideration the short-term and long-term rewards on the cumulative sum. On the other hand, a discount factor closes to 0 will take into consideration only the short-term rewards, since the long-term rewards are almost reduced to 0 with the discount factor.

We are almost ready to introduce the q-learning method. However, there is an additional property to be fulfilled if we want the process to be successful. It is essential to fulfil the Markov property (Equation 3.2):

$$p(S_{t+1}|S_t) = p(S_{t+1}|S_1, S_2, \ldots, S_t) \qquad \textit{Equation 3.2}$$

The Markov property says that the future state only depends on the current state. That is, the current state has all the information from the history, so other states different from the current one are irrelevant for the future state. Let's visualize this important property with one example. Let's imagine an autonomous driving vehicle which is programmed to follow a reference line driving at a constant speed. The variable selected as state is the lateral distance from the vehicle to the reference line. Then, let´s assume three different cases represented in Figure 3.1:

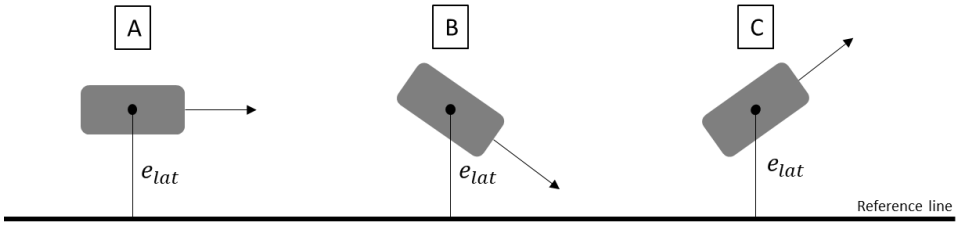


Figure 3.1: Example showing the Markov property. The observation for all 3 cases is the lateral distance e_{lat}. For case A, the optimal action is a smooth turn to the right, for case B a smooth turn to the left and for case C and hard turn to the right. Thus, the agent cannot compute the optimal action only by knowing the lateral distance. Then, it is said that the problem does not fulfil the Markov property

The goal in the previous case is to follow the reference line without lateral error. In case A, the optimal action is to turn the steering to the right, so that the lateral distance can decrease to 0. However, for case B, the optimal action is turning to the left, although the state is the same as case A. In case C, which is under the same state as case A and B, the optimal action is turning to the right, like case A, but a little bit more strongly in comparison with case A. The problem here is that the q-table cannot decide which is the optimal action. In other words, knowing the lateral distance, do we need to steer to the right or to the left? We do not know since the heading of the vehicle is an unknown variable for the agent. If we would have known the history of the lateral distance, we could have deduced the vehicle heading, and therefore could have chosen the optimal action. But the q-learning algorithm depends only on the current state, and the state selected does not fulfil the Markov property. The consequence is that the learning process will not end up successfully.

Now we are ready to introduce the q-learning method. As previously mentioned, the q-learning uses the action-value function. Equation 3.3 shows the action-value function:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \qquad \textit{Equation 3.3}$$

So, the action-value function will output the expected cumulative return G_t starting from state s, taking action a in state s, and after on following policy π.

The action-value function has the advantage that is very easy to select the optimal action given the state s. Simply by looking which action maximizes the expected cumulative reward, the optimal action can be selected (Equation 3.4):

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, q(s,a) \qquad \textit{Equation 3.4}$$

We can also easily calculate the value of a state out of the action-value function by means of Equation 3.5:

$$V_\pi(s) = q_\pi\big(s, a = \pi(s)\big) = \max_a q_\pi(s,a) \qquad \textit{Equation 3.5}$$

Or if the policy is formulated with probabilities in a stochastic way, we use Equation 3.6:

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a)$$ *Equation 3.6*

If we compute the optimal action-value function after the learning process, we will get the optimal policy as well (Equation 3.7):

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, q^*(s,a)$$ *Equation 3.7*

Therefore, the key point of the q-learning is to obtain the optimal action-value function $q^*(s,a)$. As the q-learning method works with discrete states and actions, the action-value function is a table, called the q-table. The following image shows how the q-table looks like:

q-table		Actions		
		A_1	…	A_m
States	S_1	q_{11}	q_{12}	q_{1m}
	…	…	…	…
	S_n	q_{n1}	q_{n2}	q_{nm}

Table 3.1: Q-table

To get the optimal action-value function $q^*(s,a)$, the q-learning method implements the Bellman equation. Equation 3.8 shows the formula for a deterministic environment:

$$q_\pi(S_t, A_t) = R_{t+1} + \gamma \max_a q_\pi(S_{t+1}, a)$$ *Equation 3.8*

The q-learning method guarantees the convergence to the optimal action-value function $q^*(s,a)$. However, for the convergence to occur, all states in the q-table must have visited enough times (theoretically infinite). In addition, to make the process more stable, a learning rate parameter α can be introduced to the Bellman equation (Equation 3.9).

$$q_\pi(S_t, A_t) \leftarrow (1-\alpha) q_\pi(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a q_\pi(S_{t+1}, a))$$ *Equation 3.9*

To ensure that all states in the q-table are visited, an exploration methodology needs to be implemented. The epsilon-greedy policy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly.

Finally, to sum up, the q-learning pseudocode implemented in python is shown in the following picture:

Q-learning Algorithm

Initialize q-table
Initialize parameters $\varepsilon, \gamma, \alpha$
for episode = 1:number_episodes **do**
    $s_0 = reset\ to\ initial\ state$
    while True
        Generate random_number
        if random_number > ε
            Select random action a_t
        else
            $a_t = arg\max_a q(s_t, a)$
        Execute action a_t and observe new state s_{t+1}, reward r_t and done
        $q(s_t, a_t) = (1-\alpha)q(s_t, a_t) + \alpha(r_t + \gamma \max_a q(s_{t+1}, a))$
        $s_t = s_{t+1}$
        if done == True
            break

Figure 3.2: Q-learning pseudo code

3.2. Deep Deterministic Policy Gradient (DDPG)

The DDPG (Deep Deterministic Policy Gradient) algorithm has a lot of similarities with the q-learning algorithm. It learns a q-function (the q-learning method learns a q-table) and uses the Bellman equation (same as q-learning) to iteratively update the q-function. The DDPG method, like the q-learning method, is a model free algorithm. That means, both algorithms do not use any embedded model of the environment inside the agent logic. Instead, they learn the optimal policy directly from the environment. However, there are some differences in comparison with the q-learning approach. The principal difference is that the DDPG method is intended for environments with continuous states and actions, while the q-learning method is only used for discrete environments.

If we recall from the previous section, the q-learning method finds the optimal action just by searching for the action which gives the maximum q-value, provided a state (Equation 3.4). For a discrete algorithm such as the q-learning, this problem is solved with a simple search inside the q-table. But now there is no q-table anymore. Instead, there is a continuous q-function. Therefore, it would be computationally very expensive to go through all the continuous space of the actions and select the ones which maximize the q-value. On the other hand, we can take advantage from the properties of continuous functions and differentiate it with respect to the action. In other words, it is possible to find the maximum q-value just by differencing the q-function with respect to the action.

Let's assume in Equation 3.10 a deterministic policy modelled by a neural network μ with weights θ:

$$a_t = \mu(s_t|\theta) = \mu_\theta(s_t) \qquad \textit{Equation 3.10}$$

Let's also assume that the q-function is another neural network Q with weights ϕ (Equation 3.11):

$$Q(s_t, a_t|\phi) = Q_\phi(s_t, a_t) = Q_\phi(s_t, \mu_\theta(s_t)) \quad \text{Equation 3.11}$$

The goal is then to learn a policy μ_θ which maximizes the q-function Q_ϕ (Equation 3.12):

$$\max_{a_t} Q_\phi(s_t, a_t) = \max_{\theta} Q_\phi(s_t, \mu_\theta(s_t)) \quad \text{Equation 3.12}$$

Since the q-function is continuous, we can take the gradient of the q-function and update the weights θ of the policy in the direction of the gradient. Different states might indicate different gradient directions, so we can average the value by taking an expectation with respect to the set of samples $\mathcal{D}$ of the state (Equation 3.13):

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_\theta Q_\phi\left(s, \mu_\theta(s)\right)\right] \quad \text{Equation 3.13}$$

We apply the chain rule to the gradient in Equation 3.14:

$$\theta_{k+1} = \theta_k + \alpha \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q_\phi(s, a)\big|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)\right] \quad \text{Equation 3.14}$$

The deterministic policy gradient theorem comes to the same results. Defining the objective function J (Equation 3.15) as the cumulative discounted reward (Equation 3.16), the goal is to maximize this objective function:

$$J(\mu_\theta) = \mathbb{E}[r_1^\gamma|\mu] \quad \text{Equation 3.15}$$

$$r_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k) \quad \text{Equation 3.16}$$

The deterministic policy gradient theorem computes the gradient of the objective function (Equation 3.15) which matches the previously computed gradient of the action-value function:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q_\phi(s, a)\big|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)\right] \quad \text{Equation 3.17}$$

Equation 3.18 expands the expectation considering a sample $\mathcal{D}$ of transitions:

$$\nabla_\theta J(\mu_\theta) = \frac{1}{N}\sum_i \left(\nabla_a Q_\phi(s_i, a_i)\big|_{a_i=\mu_\theta(s_i)} \nabla_\theta \mu_\theta(s_i)\right) \quad \text{Equation 3.18}$$

Therefore, we can use Equation 3.18 to learn the weights θ of the neural network $\mu_\theta(s_t)$. This neural network is also called actor network. Equation 3.18 makes use of the action-value function $Q(s, a)$. So, we need also to approximate the action-value function by learning the weights ϕ of the neural network $Q_\phi(s, a)$. This second neural network is also called critic network, because it evaluates how good the actions are, provided a specific state. Like the q-learning method, the Bellman equation is used to learn the critic network iteratively (Equation 3.19).

$$Q_\phi(s_t, a_t) = \mathbb{E}_{s\sim\mathcal{D}}\left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_\phi(s_{t+1}, a_{t+1})\right] \quad \textit{Equation 3.19}$$

And since the actor network maximizes the critic network, Equation 3.19 can be rewritten as Equation 3.20:

$$Q_\phi(s_t, a_t) = \mathbb{E}_{s\sim\mathcal{D}}\left[r(s_t, a_t) + \gamma\, Q_\phi\big(s_{t+1}, \mu_\theta(s_{t+1})\big)\right] \quad \textit{Equation 3.20}$$

If we build a mean squared error function, we can minimize this function and learn the weights ϕ of the critic network. The loss function is shown in Equation 3.21:

$$\begin{aligned} L &= \mathbb{E}_{s\sim\mathcal{D}}\left[\Big(r(s_t, a_t) + \gamma\, Q_\phi\big(s_{t+1}, \mu_\theta(s_{t+1})\big) - Q_\phi\big(s_t, \mu_\theta(s_t)\big)\Big)^2\right] \\ &= \frac{1}{N}\sum_i \Big(r(s_i, a_i) + \gamma\, Q_\phi\big(s_{i+1}, \mu_\theta(s_{i+1})\big) - Q_\phi\big(s_i, \mu_\theta(s_i)\big)\Big)^2 \end{aligned} \quad \textit{Equation 3.21}$$

Additionally, this actor-critic method uses two tricks to improve efficiency and make the learning process more stable: the replay buffer and the target networks.

The replay buffer is a high dimension vector which stores experiences from the past. The dimension of the vector is called the buffer capacity. The experiences are defined according to the tuple (s_t, a_t, r_t, s_{t+1}). During the learning process, we can select a set of experiences from the buffer. This will improve the stability of the learning because we will not use constantly the most recent experiences which might be correlated between them. However, the replay buffer has a disadvantage: the higher the set of experiences are, the more memory is needed, and it can slow down the learning process. So, a trade-off between the memory usage and the learning stability is needed.

The second trick is the target network. The target networks help to improve the stability of the training process. If we recall from the previous section, the Bellman equation is used to update the Q-function:

$$y_i = Q_\phi(s_i, a_i) = r(s_i, a_i) + \gamma\, Q_\phi\big(s_{i+1}, \mu_\theta(s_{i+1})\big) \quad \textit{Equation 3.22}$$

Like mentioned before, the loss function is defined as the mean squared error function.

$$L = \frac{1}{N}\sum_i \Big(y_i - Q_\phi\big(s_i, \mu_\theta(s_i)\big)\Big)^2 \quad \textit{Equation 3.23}$$

That means, the learning algorithm will minimize the loss function by making y_i as close as possible to $Q_\phi\big(s_i, \mu_\theta(s_i)\big)$. The problem here is that the weights ϕ that we are trying to optimize affect not only the action-value function $Q_\phi\big(s_i, \mu_\theta(s_i)\big)$ but also the quantity y_i. This can make the learning process unstable, as both the quantity y_i and the action-value function $Q_\phi\big(s_i, \mu_\theta(s_i)\big)$ are changing their value during the learning process. The idea to solve this issue is to make the quantity y_i independent from the weights ϕ which are optimized during the learning process. Following this idea, the quantity y_i will keep the same value as we recalculate the weights ϕ to adapt the value of $Q_\phi\big(s_i, \mu_\theta(s_i)\big)$ and therefore

minimize the value of the loss function. Thus, target networks for the actor and critic networks are then introduced in the quantity y_i:

$$y_i = r(s_i, a_i) + \gamma\, Q'_{\phi'}\left(s_{i+1}, \mu'_{\theta'}(s_{i+1})\right) \qquad \textit{Equation 3.24}$$

The target critic network Q' is parametrized with the weights ϕ' and the target actor network is parametrized with the weights θ'. Before ending a learning episode, the weights of the target networks are updated. The factor τ can move between 0 and 1 to select how fast we want to update the target weights from the actor (Equation 3.25) and critic networks (Equation 3.26).

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \qquad \textit{Equation 3.25}$$

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi' \qquad \textit{Equation 3.26}$$

Let's now take a closer look to the neural networks. Both the actor and critic neural networks need to learn a policy and an action-value function that are highly nonlinear. Therefore, it is necessary to have hidden layers in the neural network and the activation function of the neurons must be also nonlinear. The activation function selected is the ReLU function (Figure 3.3).

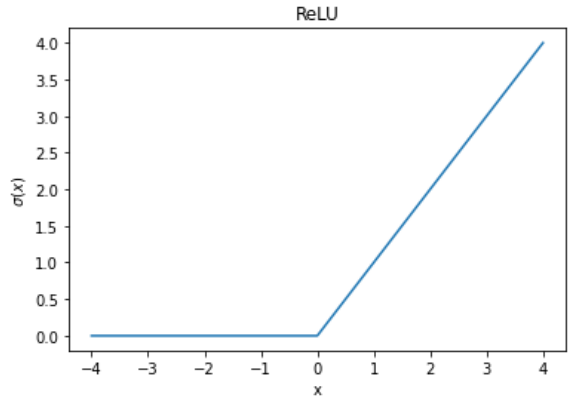


Figure 3.3: ReLU activation function

As shown in Figure 3.3, the activation function is nonlinear. The gradient is 0 for negative inputs and +1 for positive inputs. This has a positive impact during the backpropagation since the gradient will not vanish. The gradient of early layers is obtained by multiplying the gradient of later layers. The ReLU activation function has a constant gradient value of +1 for positive inputs. So, the gradient of early layers will not vanish with ReLU activation functions. On the other hand, the gradient of the sigmoid function tends to 0 if the weighted input of the neuron is a big positive or negative number (see Figure 3.4). Multiplying small numbers together will cause the gradient of early layers to vanish.

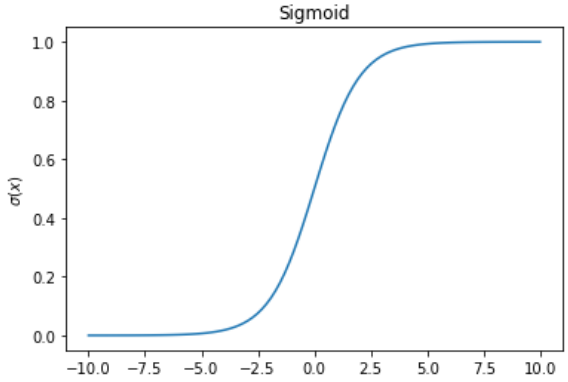


Figure 3.4: Sigmoid activation function

Another typical problem during the training process of neural networks is the explosion of the gradient. If the gradient explodes, the values obtained for the gradient are too large and the learning process becomes unstable. We can avoid this situation with a proper initialization of the weights or by clipping the gradient value. The weights initialization will be discussed later.

The activation function of the output layer of the actor and critic models cannot be a ReLU function. In case of the actor model, the outputs are normalized between -1 and 1 (or between 0 and 1). The ReLU function will output only values between 0 and $+\infty$, so it does not fit very well. Instead, we can use the tanh function, which outputs values between -1 and 1 (or the sigmoid function for values between 0 and 1):

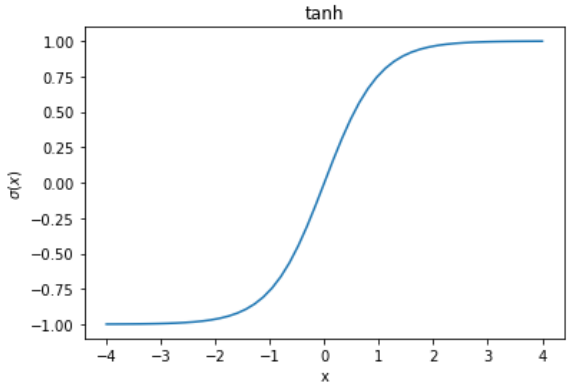


Figure 3.5: Tanh activation function

Normalizing the inputs and outputs of the neural network between -1 and +1 has shown better results than feeding the neural networks with raw signals without normalization. The only exception is the output of the critic network, which has not been normalized. The critic network evaluates how good a specific action is. This evaluation uses positive or negative numbers. An action will have a better quality if the critic network gives a larger number. For that reason, the activation function must output both positive and negative numbers and this output must not be limited. Thus, the linear activation function fits very well to the last neuron layer of the critic network.

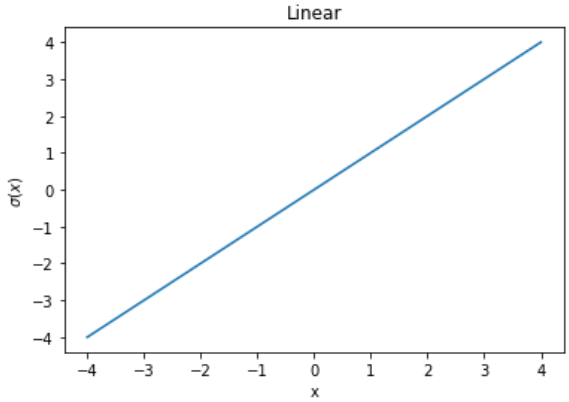


Figure 3.6: Linear activation function

Apart from the nonlinear activation functions, we need to provide the neural networks with hidden layers to learn highly nonlinear functions. The actor network has been set according to the following picture. The different symbols in bracket refers to the number of neurons in that specific layer:

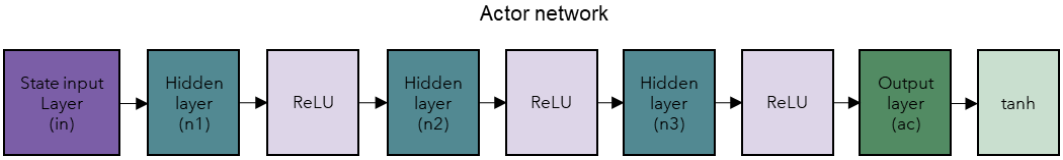


Figure 3.7: Actor network architecture

The critic network looks a little bit more complicated than the actor network. This is because the critic network considers not only the state as input variable, but also the actions. Thus, the input layer merges the state and action variables.

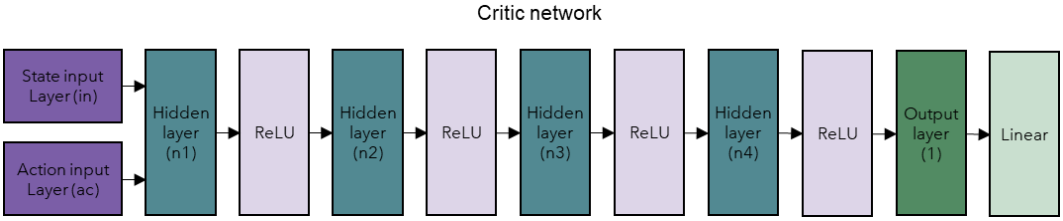


Figure 3.8: Critic network architecture

As mentioned before, a proper initialization of the neural network weights is also required. A proper initialization sets the weights in such a way that the gradient will not vanish or explode. We have already prevented the gradient vanishment with the ReLU activation functions, but it can explode and cause problems during the learning process. The best results have been achieved with the Glorot uniform initialization function (Equation 3.27). Glorot initialization considers the size of the input layer (n_{in}) and the size of the output layer (n_{out}) to adjust the limits from where to sample the initial values of the weights (it samples values inside the interval [-limit, +limit]). In case of Glorot Normal (Equation 3.28), it adjusts the variance of the normal distribution:

$$Limit[W^L] = \sqrt{\frac{6}{n_{in} + n_{out}}} \quad \text{Equation 3.27}$$

$$Var[W^L] = \frac{2}{n_{in} + n_{out}} \quad \text{Equation 3.28}$$

Once the variance or limits are determined, it samples the values of the weights for that specific layer. Adjusting the variance or limits for each layer of the neural network during the initialization ensures a smooth forward and back propagation of the information.

Another important part of the reinforcement learning algorithm is the exploration and exploitation. If there is no exploration, given a specific state, the actor network will always output the same value. Therefore, we need to explore if any other different action gives more cumulative reward than the one initially calculated by the actor model. There have been implemented two different variants for the exploration. The first exploration variant implemented is a noise signal added to the actor network output (Equation 3.29). This exploration noise has a normal distribution of zero mean value. The amplitude of the exploration is controlled with the variance of the normal distribution. There are two main drawbacks of this variant: first, it is not suitable to work with real hardware, as the noisy exploration might cause damage to the actuators of the robot. Thus, it can be used only in simulated environments. The second drawback is that this exploration strategy does not allow big exploration maneuvers. It allows just a local exploration around a given state.

$$a_t = \mu_\theta(s_t) + \mathcal{N}(0, \sigma) = \mu_\theta(s_t) + \mathcal{N}(0, \sigma) \quad \text{Equation 3.29}$$

The second variant for the exploration tries to overcome the previous two drawbacks. A low frequency sinusoidal wave is added to the actor response (Equation 3.30):

$$a_t = \mu_\theta(s_t) + sin(amp, freq) = \mu_\theta(s_t) + \sin(amp, freq) \quad \text{Equation 3.30}$$

The higher the amplitude is, the wider is the area where the robot can explore. Additionally, due to the low frequency sinusoidal wave, this exploration strategy is suitable to be used with real hardware.

As the learning process progresses and the actor and critic networks learn the optimal values, less and less exploration is needed. Therefore, a decay for the exploration has been programmed. This decay also helps to stabilize the learning process and not to get very noisy actor and critic networks. The decay factor is applied at the beginning of the learning episode according to Equation 3.31:

$$Exploration = Decay * Exploration \quad \text{Equation 3.31}$$

The exploration is only used during the learning process. During the validation process, the exploration is removed since the model has already learnt the optimal action.

Finally, the DDPG pseudocode implemented in Python is shown in Figure 3.9:

DDPG Algorithm

Initialize actor model $\mu_\theta(s)$ with weights θ
Initialize critic model $Q_\phi(s,a)$ with weights ϕ
Initialize target networks $\mu'_{\theta'}(s)$ and $Q'_{\phi'}(s,a)$ with weights $\theta' \leftarrow \theta$ and $\phi' \leftarrow \phi$
Initialize replay buffer R
Initialize the learning parameters α, τ and γ
for episode = 1:number_episodes **do**
    $s_0 = reset\ to\ initial\ state$
    while True
        $a_t = \mu_\theta(s_t) + \mathcal{N}(0,\sigma)$
        Execute action a_t and observe new state s_{t+1}, reward r_t and done
        Record the transition (s_t, a_t, r_t, s_{t+1}) in the buffer R
        Sample a batch of transitions $\mathcal{D}$ from the buffer R
        Compute y_i using the Bellman equation: $y_i = r_i + \gamma\ Q'_{\phi'}\left(s_{i+1}, \mu'_{\theta'}(s_{i+1})\right)$
        Update the weights ϕ of the critic network by minimizing the loss function:

$$L = \frac{1}{N}\sum_{i\sim\mathcal{D}}\left(y_i - Q_\phi\left(s_i, \mu_\theta(s_i)\right)\right)^2$$

        Update the weights θ of the actor network using the deterministic policy gradient theorem:

$$\nabla_\theta J(\mu_\theta) = \frac{1}{N}\sum_{i\sim\mathcal{D}}\left(\nabla_a Q_\phi(s_i,a_i)\Big|_{a_i=\mu_\theta(s_i)}\ \nabla_\theta \mu_\theta(s_i)\right)$$

        Update the weights ϕ' of the target critic network:

$$\phi' = \tau\phi + (1-\tau)\phi'$$

        Update the weights θ' of the target actor network:

$$\theta' = \tau\theta + (1-\tau)\theta'$$

        if done == True
            break

Figure 3.9: DDPG pseudo code

4. Environment model

This chapter gives an overview about the project structure. The project structure organizes the software in software components to have a clean and intuitive software architecture. This project structure also defines folders to store different environment models and robot models, as well as dedicated folders for configuring the environments and store the output results. On the other hand, the different components of an environment are explained in this chapter (vehicle model, reward function and state variables). The motion control software module is also included as a part of the environment since the environment is everything that the reinforcement learning agent sees. Finally, the difference between online and offline learning is explained.

4.1. Project structure

The python GYM toolkit has been used for the implementation and comparison of different reinforcement learning algorithms under different environments. The folder structure of the project is shown in Figure 4.1:

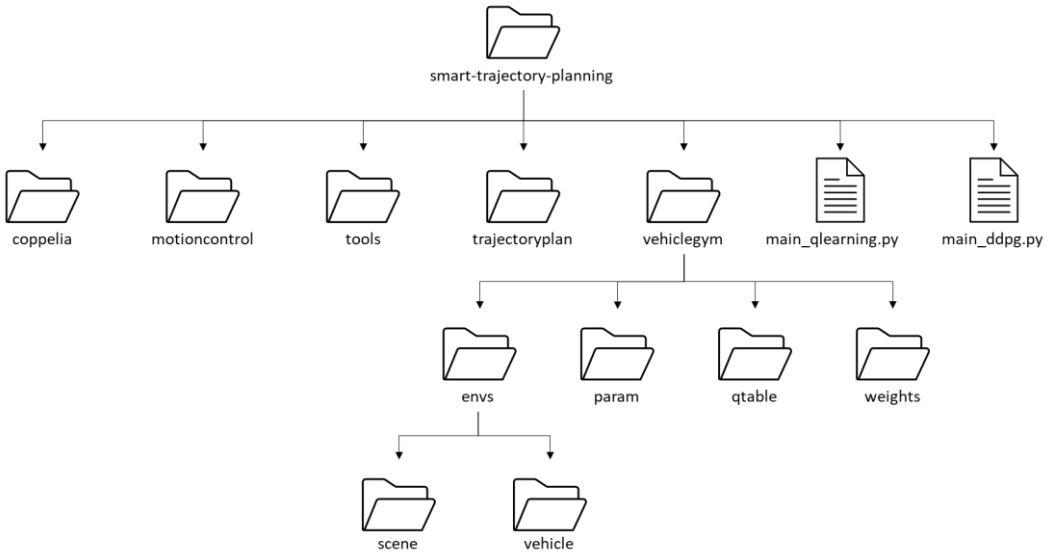


Figure 4.1: Project folder structure

- smart-trajectory-planning: project's folder
- motioncontrol: implementation of low-level control algorithms to control the movement of the robot or vehicle
- tools: folder to store helper functions
- trajectoryplan: reinforcement learning algorithms storage folder
- vehiclegym/envs/scene: folder to store the scene where the robot is moving and interacting. The scene is loaded in CoppeliaSim software
- vehiclegym/envs/vehicle: folder to store the vehicle or robot used inside the scene. Connection with CoppeliaSim is needed
- vehiclegym/qtable: folder to store the q-table after the q-learning process

- vehiclegym/weights: folder to store the weights of the neural networks after the DDPG learning process
- vehiclegym/param: folder to store the parametrization used in the different test cases
- main_<method>.py: script to launch the environment and learning process

The *trajectoryplan* folder contains a python script with all the necessary functions for the q-learning and DDPG methods.

The script main_<method>.py will register the selected environment and launch the learning process with the reinforcement learning method. It is also possible to give a policy as input and simulate the environment with the robot following that given policy.

To define a custom GYM environment from scratch, we need to fulfill some standards and define 4 important functions:

- Reset function: it will reset the state of the environment to the initial default state. Therefore, this function will be triggered at the beginning of the learning episode. An episode will be terminated every time that a terminal state is entered.
- Step function: it will simulate the environment considering the current state, the action selected by the agent and the dynamics of the environment.
- Render function: this function will visualize the current environment state. It is possible to render 3D elements in a graphical window to visualize the current state of the environment. This is an optional function, but very useful to get a first impression of what the agent and the environment are doing.
- Init function: function which initialize all variables used to build the environment.

4.2. Vehicle model

This section describes the vehicle or robot models used in the experiments. There are two different types of models considered. First, a discrete robot model. The reason to use this simple model is because it is discrete, and the q-learning method can be used with it. The fact that it is discrete makes much simpler the debugging process. Secondly, a continuous differential robot. This model tries to reproduce the movements and dynamics of a real robot. Thus, the final reinforcement learning algorithm to control the robot will be validated with it. Since the model is continuous, the q-learning algorithm does not fit any more and deep reinforcement learning algorithms need to be implemented for this case.

4.2.1. Discrete robot model

The vehicle modelled inside this environment does not include any lateral or longitudinal dynamics. There are only 3 possible actions: move forward, turn left, or turn right, as represented in Figure 4.2:

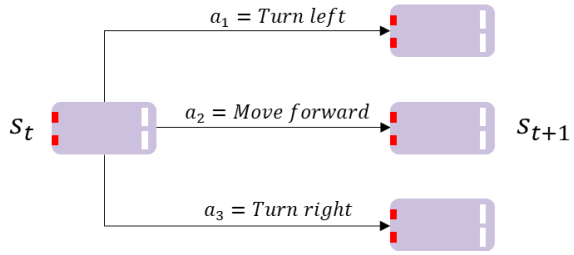


Figure 4.2: Discrete actions of the robot model

Thus, the vehicle will move instantly (without dynamics) from one state to the next one depending on the action taken. The principal advantage of this simplification is that the actions and the states are discrete. Discrete states and actions allow the usage of simple but powerful reinforcement learning methods. This is the case of the q-learning method, which works with discrete states and actions.

The vehicle can freely move inside a circuit defined by the circuit boundaries. If the vehicle leaves the circuit boundaries, the learning episode gets terminated.

4.2.2. Differential robot

A differential robot is a robot with 2 independent driven wheels. Another free turning undriven wheel is mounted to stabilize the robot. The 2 driven wheels are connected to two electrical motors. The angular speed of these motors can be controlled (see next section 4.3) so that the robot can travel at a desired linear speed v_{robot} and a desired angular speed w_{robot}. Figure 4.3 shows a picture of the differential robot kinematics:

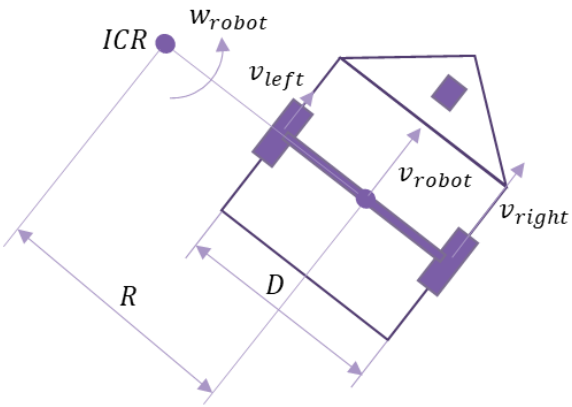


Figure 4.3: Differential robot kinematics

The robot linear speed v_{robot} can be easily calculated by means of the formula $v_{robot} = w_{robot} * R$, where R is the instantaneous radius of curvature (ICR = instantaneous center of rotation). In the same way, the left and right linear wheel speeds can be calculated according to Equation 4.1 and Equation 4.2:

$$v_{left} = w_{robot} * \left(R - \frac{D}{2}\right) \qquad \textit{Equation 4.1}$$

$$v_{right} = w_{robot} * \left(R + \frac{D}{2}\right)$$ *Equation 4.2*

Solving Equation 4.1 and Equation 4.2, the robot angular speed w_{robot} and the instantaneous radius of curvature R can be obtained:

$$w_{robot} = \frac{v_{right} - v_{left}}{D}$$ *Equation 4.3*

$$R = \frac{D * (v_{right} + v_{left})}{2 * (v_{right} - v_{left})}$$ *Equation 4.4*

And finally, the robot linear speed v_{robot} is obtained in Equation 4.5:

$$v_{robot} = w_{robot} * R = \frac{v_{right} + v_{left}}{2}$$ *Equation 4.5*

Regarding the set of sensors mounted in the differential robot, they must provide measurements to all the variables that the reinforcement learning agent get from the environment. An encoder sensor is mounted in both powered wheels to measure the angular wheel speed and therefore calculate the robot linear and angular speeds according to Equation 4.3 and Equation 4.5. An accelerometer is also mounted to measure the robot angular and linear acceleration. Finally, a lidar sensor measures the relative position of the obstacles with respect to the robot. The lidar sensor covers 180 degrees.

4.3. Motion control

The motion control is responsible of controlling the actuators to realize the trajectory planned by the motion planning module. Therefore, the motion planning is directly connected to the motion control software module. Figure 4.4 shows the software architecture for autonomous driving vehicles and the connection between motion planning and motion control modules.

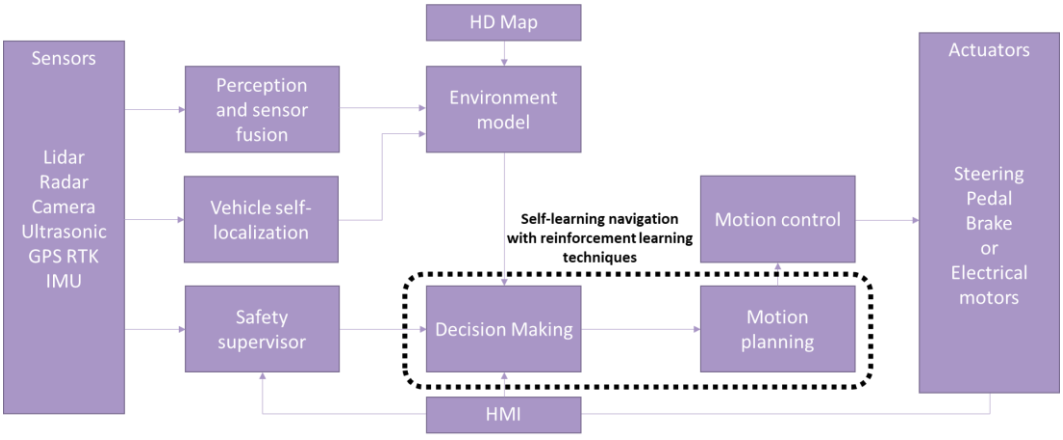


Figure 4.4: ADAS software architecture

The motion control module commands the actuators of the vehicle. In case of a differential robot, the motion control will control the rotational speed of the two electrical motors mounted in each wheel. In case of front steering vehicles, the motion control will command the steering angle position, pedal position, and brake position.

The motion control normally runs in an electronic control unit at a frequency of at least 100 Hz. This frequency is generally enough to control the actuators to the desired set point values.

The motion control is included in the environment seen by the agent. The motion planning generates the robot's linear speed and the robot's angular speed. These two values are taken by the motion control and transformed into values to directly feed the actuators. Figure 4.5 shows the main signal flow between software modules:

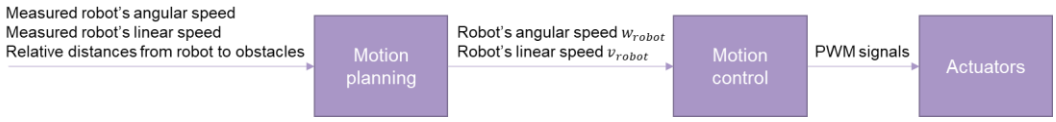


Figure 4.5: Motion planning and motion control signal interface

In the case of a differential robot, the linear speed and angular speed are converted first into the left and right wheel speeds using Equation 4.6 and Equation 4.7:

$$v_{left} = v_{robot} + w_{robot}\frac{D}{2} \quad \textit{Equation 4.6}$$

$$v_{right} = v_{robot} - w_{robot}\frac{D}{2} \quad \textit{Equation 4.7}$$

Figure 4.6 shows a differential robot and its main variables. v_{left} is the linear speed of the left wheel, v_{right} the linear speed of the right wheel, v_{robot} the linear speed of the robot, w_{robot} the angular speed of the robot and D is the distance between the two wheels.

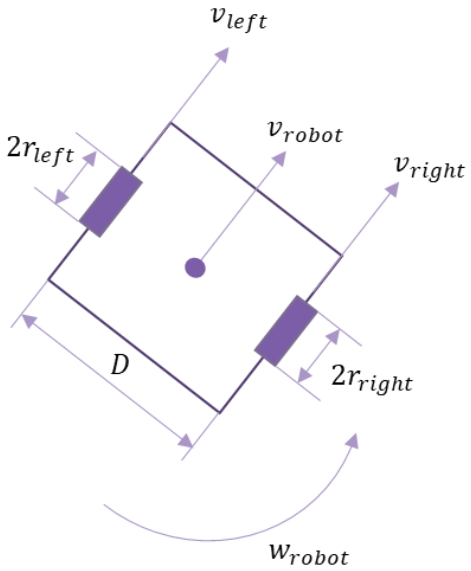


Figure 4.6: Differential robot geometry and linear velocities

The linear speeds are converted in angular speed by means of the radius of the wheel:

$$w_{left} = \frac{v_{left}}{r_{left}} \qquad \textit{Equation 4.8}$$

$$w_{right} = \frac{v_{right}}{r_{right}} \qquad \textit{Equation 4.9}$$

These two set point values of the wheel's angular speed are controlled by two independent PID controllers. These PID controllers convert the angular speed into a duty cycle for a PWM signal. The PWM signal can adjust the voltage of the electrical DC motor to control its angular speed. The complete control diagram is shown in the following picture. The policy out of the deep reinforcement learning controls the position of the robot by adjusting the linear speed and the angular speed. On the other hand, these linear and angular speeds are converted into wheel rotational speeds and two PID controllers control the measured angular speeds to these values:

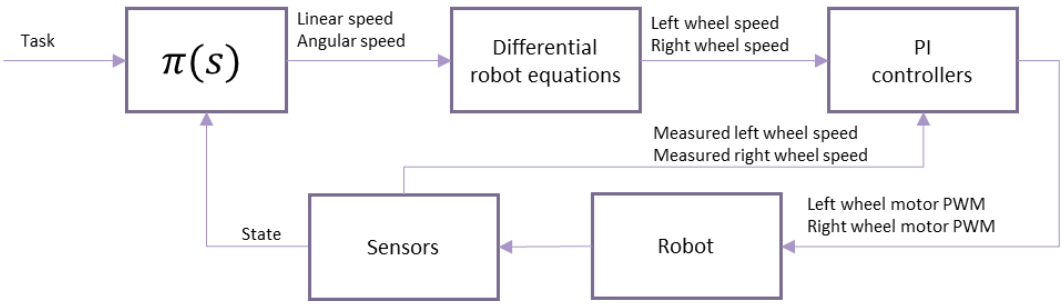


Figure 4.7: Cascade motion control. The agent controls the position of the robot by adapting the linear and angular speeds based on the observation from the environment. Then, the motion control uses PI controllers to control the linear and angular speeds

From Figure 4.7 we can see a cascade control for the robot position. To make the system stable, the wheel speed control must be at least 5 times faster than the control of the position of the robot. Therefore, the proportional part, integral time, and derivative part of the PID controllers are calibrated to achieve this required fast dynamic response avoiding at the same time overshooting of the angular speed.

4.4. Reward functions

The reward functions are the key to the reinforcement learning algorithm. The reward function is not telling us which action to take, as the supervise learning would do. Instead, the reward functions are a measurement of how good or bad is to end up in a new state after taking an action. This measurement of good or bad is closely related to the sensation of pain or pleasure which takes place in our brain. For example, let's take the case of a robot travelling at a constant speed towards an obstacle. An incorrect reward would be to say that if the robot continues driving straight, this will be a bad reward. In fact, at the beginning of the learning process we do not know if driving the robot towards the obstacle is good or bad. This information will be stored in the critic function (cost function) after the learning process. Instead, the reward must be related to the pain or pleasure produced by the actions taken. This is, if the robot performs and action and, as a result of it, the robot crashes, the reward obtained will be bad. Next time, the robot will try to avoid selecting this action for that given state, which results in the robot crashing into an obstacle and producing pain. This is because the critic function tells us that, for this given state, continue driving straight will end up in a bad situation (or bad cumulative reward). Therefore, the critic function gathers all the experience seen during the learning process to build a function that evaluates which action is better, given a specific state.

Regarding the type of rewards, they have been divided into safety, legal, comfort and task-oriented requirements. The safety, legal and comfort requirements are based on the pain that the robot might suffer if they are not fulfilled. On the other hand, the task-oriented requirements are based on the sensation of pleasure that is obtained by fulfilling the proposed tasks. Therefore, depending on the robot mission, the task-oriented requirements can be built following different criteria.

Finally, it is very important to follow some rules by the time of defining the reward functions:

- In problems with continuous actions and states, or in case of too large discrete spaces, it is very convenient to shape the reward function instead of using sparse rewards. The reason behind is that the robot learns by means of exploration. If the robot starts randomly exploring, the probability of the robot seeing this sparse reward can be very low, leading to an either unsuccessfully learning process because the robot does not find the reward or to an extremely long learning process. On the other hand, in case of shaped rewards (rewards with smooth continuous gradient), the robot can adapt its actions in the direction of maximizing the cumulative reward right from the beginning. For example, let's imagine a robot that

must avoid obstacles. Defining a negative reward if the robot hits an obstacle and otherwise no reward is not convenient. It can take a long learning time until the robot hits an obstacle to get the negative reward and learn from it. Instead, a safety margin around the obstacle can be defined to write a shaped reward function. If we consider another example of giving a positive reward at the destination pose and otherwise no reward, the situation can be even more critical, since the robot might never find the way to the destination pose by random exploration.

- It is better to avoid pure negative rewards. If the reward functions are always negative, excluding the 0, the reinforcement learning might find a solution by terminating the learning episode as soon as possible rather than keeping on driving and therefore accumulating more negative reward. For example, if a robot is always penalized with negative rewards, the logic can decide to crash the robot to stop accumulating for a long period of time small negative rewards until it reaches the destination.
- Positive rewards are also tricky, and they must be treated carefully. If the robot finds a positive reward, the logic might end up in a loop solution. For example, if the robot gets a positive reward as the robot approaches the destination, the logic might find a solution of the robot spinning around the destination without reaching it to continuously accumulate reward. In this work, positive rewards have been avoided. But in case of its usage, conditions to terminate loops or a large sparse positive reward at the destination might be needed.
- Always carefully design the learning scenarios to facilitate finding rewards from a beneficial starting situation. For example, let's think of a robot that starts getting negative reward at a distance lower than 1 meter of the obstacle. If we design a scenario with left and right walls separated less than 2 meters, the robot will get constantly negative rewards and it will face the problem described in the previous second point. Instead, start with a wall separation slightly higher than 2 meters. If the robot learns a right policy, later it will be able to generalize to bottle neck situations correctly, where the walls are closer than 2 meters.

We can then conclude that depending on the actions taken, the robot obtains different rewards values, and these reward values are sum up until the end of the learning process. Therefore, the goal of the reinforcement learning algorithm is to find out the execution of series of actions that maximize the cumulative reward obtained until the learning process is terminated.

4.4.1. Safety requirements

The safety requirements implement a penalization or a negative reward if the robot or vehicle crashes into obstacles. Situations where the vehicle drives too close to the obstacle are also penalized although no contact between vehicle and obstacle happens. For that reason, a circular safety margin around the obstacle is defined. A very simple reward function (Equation 4.10) is therefore defined: 0 if the vehicle drives outside the safety margin, and a linear interpolation between 0 and -1 if the vehicle drives inside the safety margin (see Figure 4.8).

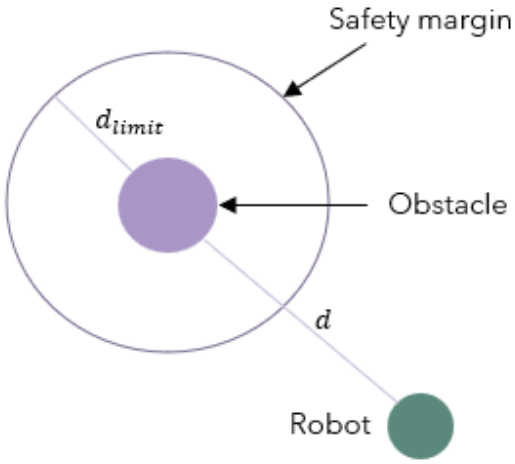


Figure 4.8: Safety requirements. A safety margin is defined around the obstacles. If the robot enters this area, the environment will send a negative reward to the agent

$$Reward_{safety} = \begin{cases} 0 & , \quad d > d_{limit} \\ \dfrac{d - d_{limit}}{d_{limit}} & , \quad d \leq d_{limit} \end{cases}$$

Equation 4.10

4.4.2. Legal requirements

The legal requirements are closely related with the road maximum speed. The clearest example is the road speed limit of a front steering vehicle. If the road maximum speed is 120 km/h, the vehicle must control its speed so that it does not exceed this value. In the same way, a robot might move in an area where its maximum speed must be limited to some predefined value.

The implementation of this reward function is very simple as well. There are two speed thresholds. If the vehicle speed does not exceed the speed value of the lower threshold, the reward will be 0. If the vehicle speed is between the thresholds, the reward will be linearly interpolated between 0 and -1 (Figure 4.9).

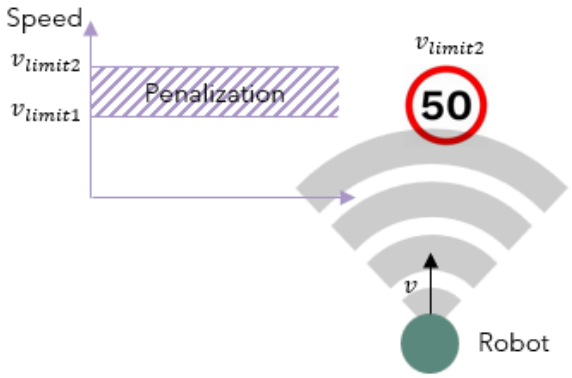


Figure 4.9: Legal requirements. A penalization band is defined for a range of linear velocities. Below the lower limit v_{limit1}, the robot gets no penalization

$$Reward_{safety} = \begin{cases} 0 & , \quad v < v_{limit1} \\ -\dfrac{v - v_{limit1}}{v_{limit2} - v_{limit1}} & , \quad v_{limit1} \leq v \leq v_{limit2} \\ -1 & , \quad v > v_{limit2} \end{cases}$$ *Equation 4.11*

4.4.3. Comfort requirements

Comfort can be defined as a pleasant feeling of the people travelling inside a vehicle. The feeling of discomfort can come from different sources. The three main sources which can decrease the feeling of comfort are listed below:

- High acceleration values (longitudinal and lateral/angular)
- High jerk values (longitudinal and lateral/angular)
- Low frequency acceleration over long periods of time

The acceleration can be seen as a force if we apply it to person of mass m. The generated force due to the acceleration can be calculated as $F = ma$. The higher the acceleration is, the higher the force applied to a person travelling inside the vehicle. A high acceleration or force can be very uncomfortable. If the force is too high, the target person might not be able to stay still inside the vehicle, or even loose completely the control and crash into the interior of the vehicle, causing possible damage. Even if the person can control itself, there is a maximum value of the acceleration, above which the people will start to find themselves uncomfortable. Therefore, it is important to make sure that the vehicle acceleration never goes above a predefined maximum value or under a predefined minimum value.

The second source of discomfort is the high jerk values. The jerk is defined as the derivative of the acceleration. Thus, the jerk can be seen as a changing force in a person. If the force changes too fast, even if the force magnitude along the process is not high, it could be possible that the person cannot even react on time to control its body and remain still. In the extreme case of steps in the acceleration, the person will feel these infinite impulse jerk values like a slap. Therefore, it is also required to control the maximum and the minimum values of the jerk.

Finally, the third source of discomfort is the low frequency acceleration maintained over a long period of time. Low frequency forces applied to a person are associated to a feeling of sickness. Short period of time will not cause any issue, but a persistent low frequency signal over a long period of time can cause discomfort. In this case, the problematic range of frequencies must be detected and avoid long period of time under these conditions.

In this work we are going to control only the first source of discomfort, the high angular acceleration values, which is also the most common one. To avoid high angular acceleration values, a simple reward function will be defined. The reward function defines a window given two acceleration values, $\dot{w}_{max1}$ and $\dot{w}_{max2}$. Accelerations below $\dot{w}_{max1}$ will not be penalized (penalization is a negative reward). Accelerations within $\dot{w}_{max1}$ and $\dot{w}_{max2}$ will be penalized using a linear

interpolation. The maximum penalization value will be clipped if the acceleration is higher than $\dot{w}_{max2}$. Figure 4.10 shows the acceleration window where the penalization is applied and a schematic of a robot and an IMU sensor, used to measure the acceleration values of the robot or vehicle. Equation 4.12 implements the reward formula for the comfort requirements.

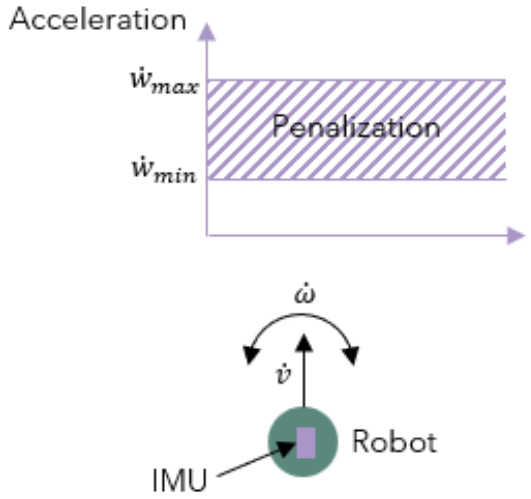


Figure 4.10: Comfort requirements. A penalization band is defined for a range of angular accelerations. Below the lower limit $\dot{w}_{min}$, the robot gets no penalization

$$Reward_{comfort} = \begin{cases} 0 & , \quad \dot{w} < \dot{w}_{max1} \\ -\dfrac{\dot{w} - \dot{w}_{max1}}{\dot{w}_{max2} - \dot{w}_{max1}} & , \quad \dot{w}_{max1} \leq \dot{w} \leq \dot{w}_{max2} \\ -1 & , \quad \dot{w} > \dot{w}_{max2} \end{cases} \qquad \textit{Equation 4.12}$$

The longitudinal comfort has not been considered to keep the experiment simple, but it can be easily added just by replacing the angular acceleration by the longitudinal acceleration.

4.4.4. Task-oriented requirements

Task-oriented requirements are requirements intended to fulfil a specific task. It is possible to define multiple tasks and they strongly depend on the autonomous driving problem to be solved. For example, in the case of a front steering vehicle travelling in the highway, there might be a task-oriented requirement to reward the vehicle if it drives in the right lane when no overtaken maneuver is intended. In case of a differential robot which is exploring a room to reach a specific destination or location in the room, a task-oriented requirement could be to drive the robot to this desired destination. As a complement to this previous requirement, another task could be to reward the robot with higher speeds to decrease the travelling time. Since this work is making use of a differential robot, the last two tasks are going to be implemented.

For the first task, the robot will be rewarded when facing the right direction to the target destination. The direction to the target destination θ_{target} can be determined by visual sensors such as cameras or providing waypoints to the

robot with the target destination coordinates. Figure 4.11 shows the target heading angle θ_{target} and the robot's heading angle θ_{robot}. If the difference between these angles is not 0, the robot will be penalized according to Equation 4.13. Notice that the penalization, like the other ones, is normalize between 0 and -1.

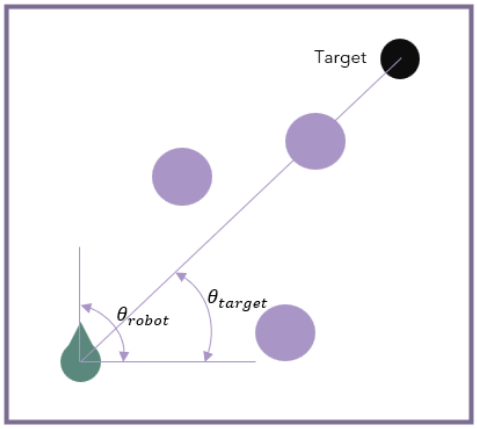


Figure 4.11: Task-oriented requirements to reach a destination pose. The robot gets penalized if it does not face the direction of the destination pose

$$Reward_{destination} = -\frac{|\theta_{target} - \theta_{robot}|}{\pi} \quad \text{Equation 4.13}$$

For the second task, a simple reward function will be defined (Equation 4.14). The robot will get penalized if the speed is 0 and it will get no penalization if the robot travels at the maximum target (legal) speed (reward is clipped to a minimum value of -1 and a maximum value of 0). Since the legal requirements are also implemented, if the robot exceeds the maximum legal speed, it will be penalized.

$$Reward_{speed} = -\frac{v_{robot} - v_{target}}{v_{target}} \quad \text{Equation 4.14}$$

Finally, the reward due to the task-oriented requirements is obtained by adding the single functions:

$$Reward_{task} = Reward_{destination} + Reward_{speed} \quad \text{Equation 4.15}$$

4.4.5. Final reward function

The final reward function can be simply built by adding the single reward functions (Equation 4.16). Each reward function is normalized between -1 and 1 (negative reward values are a penalization and positive values are a reward). So, if we simply add together all the reward functions, we are giving the same level of importance to all the requirements represented by these reward functions. Instead, to give more importance to some requirements, a weighted addition of the single reward functions is performed:

$$Reward = K_{safety} * Reward_{safety} + K_{legal} * Reward_{legal} + K_{comfort} * Reward_{comfort} + K_{task} * Reward_{task} \quad \text{Equation 4.16}$$

In general, the order of importance within the requirements is the following: safety, legal, comfort and task-oriented. For that reason, the weights are selected so that the Equation 4.17 is fulfilled:

$$K_{safety} \geq K_{legal} \geq K_{comfort} \geq K_{task} \qquad \text{Equation 4.17}$$

The reinforcement learning algorithm is then implemented so that the cumulative reward of the previous reward function is maximized.

4.5. State variables

The state variables are essential during the reinforcement learning process, and they must be selected carefully. The state variables are those variables which the reinforcement learning agent observes from the environment. Therefore, the agent uses this information to learn the policy by trial and error. Figure 4.12 shows the classical reinforcement learning image where the agent gets from the environment an observation and a reward and uses this information to learn a policy and produce an optimal action.

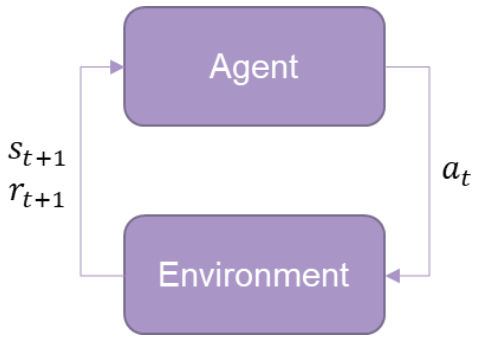


Figure 4.12: Agent-environment interaction

The selection of the state variables must fulfil 4 important points:

- State variables must be measurable signals

First at all, the state variables must be signals that can be measured by physical sensors. For example, the robot linear speed can be easily measured with encoders in the wheels, and it can be used as a state variable for the agent. This is very important because the selection of simulated variables which cannot be measured in real life will not make possible to implement the reinforcement learning algorithm using real hardware.

- State variables must be able to generalize to unseen scenarios

State variables must be capable of generalization so that the policy obtained after the learning process can solve unseen scenarios. This property is very important because if the policy cannot generalize to unseen scenarios, we would be forced to generate infinite possible scenarios during the learning process, which is unfeasible. Figure 4.13 shows an example of a set of state variables that can generalize and another different set of state variables that are not able to generalize:

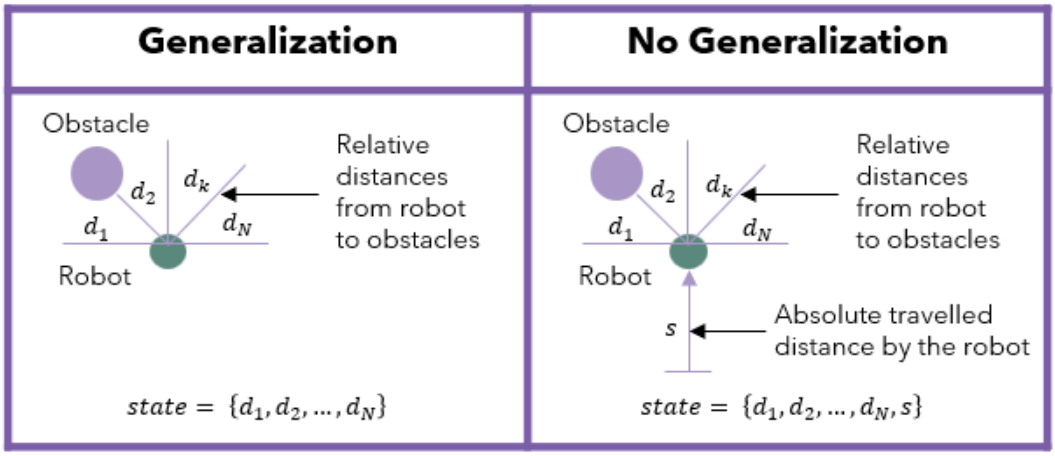


Figure 4.13: Generalization of the state variables. The left set of variables includes only relative distances. The agent can generalize to new unseen situations because, no matter where the obstacle and the robot are located, if the relative position is the same, the observation from the environment is identical. The right set of variables includes relative and absolute distances. The agent cannot generalize because it needs to learn the same policy for each absolute distance

The left configuration of state variables are all relative distances from the robot to the obstacles. This will allow the robot to avoid the obstacle after the learning process. However, if the obstacle is in another location of the environment, the robot will still avoid it safely. Therefore, we do not need to generate infinite scenarios where the obstacle is placed in each location of the environment for the policy to avoid the obstacle. On the other hand, the right configuration has considered as a state variable the absolute distance travelled by the robot. In this case, a similar policy must be learnt for all the values of the distance travelled. Since the distance travelled is a continuous variable, it will mean having to generate infinite number of scenarios to learn the right policy.

The generalization of the state variables has also an additional and important consequence. Since the generalized variables are relative distances, velocities, or angles with respect to the vehicle, we can end up in the very same state as we were before after taking an action. If we are working with continuous reward functions, this can cause a problem during the learning process. To explain the root cause of this problem, we can have a look to the Bellman equation which is used to iteratively update the action-value function:

$$q_\pi(S_t, A_t) = R_{t+1} + \gamma \max_a q_\pi(S_{t+1}, a) \qquad \textit{Equation 4.18}$$

Assuming that we are on the state S_t and take the action A_t, due to the fact that the variables are generalized, we can end up in the same state, i.e., $S_{t+1} = S_t$. Under this situation, if a reward is received from the environment, we run into the problem that the action-value function will grow up to infinite (or minus infinite). This unlimited cumulative reward brings instability to the learning process and the policy generated will not be able to fulfil the requirements. To limit the action-value function, a discount factor less than 1 is needed. In cases where the variables are not generalized and absolute values are used as state variables, the discount factor is used to give more importance to instant rewards. For example, if we chose a discount factor less than 1 and absolute state variables,

the policy would find the path to the destination using the shortest way, since a delay arriving at the destination will cause the reward to be discounted and therefore be less quantity than arriving without delay.

- State variables must fulfil the Markov property

The theoretical part with a practical example has been already discussed under the q-learning chapter (section 3.1). The state variables need to fulfil the Markov property, otherwise the learning process will be unstable. If the Markov property is not fulfilled, the given state is not fully determined. This means that the selection of the optimal action depends on the history of states. Then, it is not possible to determine which action is the optimal one because we do not have information about this history of states. Figure 4.14 shows two cases, one where the Markov property is not fulfilled and another one where it is fulfilled.

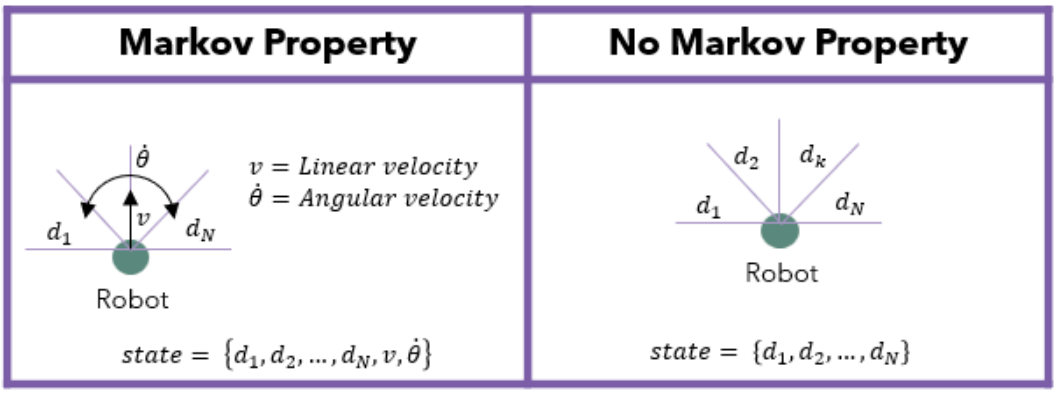


Figure 4.14: Markov property of the state variables. The left set of variables includes the linear and angular speeds. The missing of the linear and angular speeds in the right set of variables makes it necessary to know the history of relative distances to guess the direction of movement of the robot

The left configuration of state variables fulfils the Markov property. Given the state variables and after the selection of one desired action, we can determine the next state of the robot. On the other hand, the right configuration, where the linear velocity and the angular velocity of the robot are not provided, does not fulfil the Markov property. Under this configuration of state variables and assuming an action provided by the policy, we cannot determine the next state of the robot. Considering two cases where the first case is the robot moving forward and the second case is the robot remaining standstill, the same action will result in very different states. As we do not know the history of states, it is not possible to determine the next state. Therefore, the policy will be also confused, and it will not be able to select the optimal action, as the policy does not know the history of states either.

4.6. Online learning versus offline learning

The main difference between online and offline learning is that offline learning resets the environment after the learning process is terminated, while the online learning does not reset it. Thus, the online learning makes possible to navigate and learn at the same time with a physical robot in a real environment. Therefore, the online learning must count with a powerful computing unit mounted directly in the robot or deploy edge computing architectures.

Table 4.1 shows the differences and the similarities between the online and offline learning:

Offline learning	Online learning
Low computational cost	High computational cost
Suitable for microcontroller	Suitable for GPUs or edge computing technologies
Suitable to speed up learning process and generate high maturity policies (front loading)	Suitable when the robot is working under environments of unknown dynamics (robot operating in another planet like Mars)
It can learn from both simulation and real scenarios	

Table 4.1: Offline versus Online learning

The sequence of the offline learning is as follows:

1. Robot starts operation in the environment following current policy
2. If obstacle is detected too close to the robot (or bumper sensor is activated) the robot stops and learning episode gets terminated
3. Environment is reset (the position of the robot might change to a completely different location for the next learning episode)
4. Restart new learning episode

The sequence of the online learning is as follows:

1. Robot starts operation in the environment following current policy
2. If obstacle is detected too close to the robot (or bumper sensor is activated) the robot stops and learning episode gets terminated
3. The robot backs up a little bit (predefined distance)
4. The robot turns around a predefined angle
5. Restart new learning episode

For both the online and offline learning, the learning process must be terminated after some conditions are met. There are two conditions defined to consider that the policy is mature enough to fulfil the requirements. First at all, the number of total episodes must be greater than a minimum number. With this condition we ensure that the exploration rate has decayed until almost 0 to avoid very noisy and unstable policies, as discussed in the chapter 3.2. Secondly, no termination due to non-compliance of the safety, legal or comfort requirements must be detected for a minimum number of episodes to ensure proper learning. In order words, the termination of the learning episode must be only related to task-oriented requirements, such as if the robot reaches the destination pose. If the episode gets terminated because the robot crashes into an obstacle or any acceleration value is greater than the maximum threshold, the counter to consider that the policy is ready gets reset to 0.

5. Experiments and results

This section introduces the experiments and results of the self-learning navigation algorithm with reinforcement learning techniques under different driving scenarios and robot models. First, two different robot models are considered: a discrete robot model and a differential robot model. The reason behind using a discrete robot model is to make use of the q-learning algorithm. The q-learning algorithm has a lot of similarities in comparison with the DDPG algorithm. But since the q-learning is for discrete states and actions, a problem that might happen in the DDPG algorithm can be debugged easily by setting up a similar experiment with the q-learning algorithm. On the other hand, the following experiments have been set up:

- Circuit with obstacles: circuit whose drivable space is delimited by walls. Obstacles and traps are included inside the scenario. The goal of this scenario is to select the optimum number of laser beams between these possible options: 8, 45 or 180 laser beams. In each case, the lidar always covers 180 degrees. Traps like zigzags or U-traps are set up inside the scenario to prove if a reduction of laser beams can successfully drive the robot to the destination without getting trapped in the circuit. Only safety requirements are considered to keep the experiment only focus on the optimal selection of total number of laser beams.
- Circuit without obstacles: after the selection of the total number of laser beams, a circuit without obstacles has been set up. The goal here is to keep the scenario as simple as possible and to stepwise include the different requirements. Four different use cases have been considered: first, only safety requirements. Second, safety and legal requirements. Third, safety, legal and comfort requirements. And finally, all requirements together, safety, legal, comfort and task-oriented requirements. Therefore, this experiment tries to demonstrate that the reinforcement learning algorithm can successfully find an optimal policy fulfilling all the given requirements.
- Circuit with dynamic obstacles: after the number of laser beams selection and the successful commissioning of all autonomous driving requirements (safety, legal, comfort and task-oriented), a scenario with dynamic obstacles is considered. The goal of this experiment is to prove under complex driving scenarios that the reinforcement learning algorithm can find an optimal solution.
- Room with obstacles: this scenario tries to reproduce an open world scenario where the robot must reach a destination pose from a starting pose. However, spatial boundaries have been introduced in the scenario by means of walls. The reason is to limit the area where the robot can explore until it finds the destination, and therefore decrease the total learning time, which is normally very high (more than 5 hours). The goal of this experiment is to prove that the reinforcement learning logic can

successfully perform in open world scenarios. Additionally, the online learning approach has been validated under this experiment.

Some experiments implement two different versions of the target scenario. For example, the “circuit with obstacles” experiment implements two scenarios, where the distribution of the obstacles inside the circuit is different. On one hand, a first and simplified scenario is implemented for the learning process. On the other hand, once the learning process is completed, a more complicated scenario is created to validate the policy obtained during the learning process. It is very important to carefully prepare a simplified first scenario for the learning process. Otherwise, the learning process might not end up successfully. There are two general rules to pay attention to while designing a scenario for the learning process:

- We must ensure that the robot sees a reward soon enough during the learning episode. For example, if we define a single reward value at the destination pose, the probability that the robot finds out this reward after following all the way to the destination is extremely low. The consequence is that the policy does not learn how to avoid crashing into the walls. Then, the robot will always crash into the walls before reaching the destination and getting the reward. Thus, the learning process will end up unsuccessfully. This is also closely related to the way that the robot is exploring the surroundings. The robot can explore new actions in the hope to find out rewards that later will shape the policy function to maximize the cumulative reward. But the exploration range defined in this work is local around the current position of the robot, either by using a gaussian noise function or by using a sinusoidal wave with a predefined amplitude. But the exploration does not have a large or global range. For example, the exploration is not defined in this work as a predefined path to take, where there might be a probability to reach the destination by choosing the correct random path. Therefore, as the exploration is locally defined around the current position of the robot, we must ensure that the robot sees rewards soon enough for a successfully learning process.
- We must include in the scenario all the elements that later can result in a robust and complete policy, trying to cover all the range of sensor measurements. For example, if the robot is intended to avoid obstacles, it is always good to include an obstacle on the left part of the road and another obstacle on the right part of the road. If the robot finds the obstacle at the left or right position, the readings from the lidar sensor are completely different. Following this rule, the robot will generate a complete set of measurements covering all the range of the sensors and therefore the robot will learn a complete and robust policy.

The previously mentioned experiments, together with the learning process of the reinforcement learning algorithm, have been run in a system with the following characteristics:

- Processor: AMD Ryzen 5 2600 Six-Core Processor 3.40 GHz

- Installed RAM: 8 GB
- GPU: Not installed

Due to large learning times (see tables of total learning times inside this section), the learning process have been terminated before the optimal policy is determined. By letting the learning process run more time, more mature policies can be obtained. The conditions to terminate the learning process are explain in section 4.6. With a better and more appropriate setup for this type of experiments, the total learning time can be drastically reduced. A server with a dedicated GPU can speed up the neural network training process (GPU usage) as well as the robot and environment simulation (server with large RAM memory and high number of cores).

The sections below explain in more detail the experiments previously mentioned.

5.1. Discrete robot model

The first model is a simplified and discretized robot model. The goal to start with a very simplified model of the robot is to justify that the reinforcement learning techniques can successfully solve complex autonomous driving problems. As the model of the environment is discretized, we are going to use the q-learning algorithm.

The model of the robot is very simple and admits only 3 actions: turn left, move forward, or turn right (see section 4.2.1). The observation variables will include only relative distances from the robot to the obstacle (lateral relative distance and longitudinal relative distance). It is important not to include any absolute distance, like the longitudinal travelled distance. If so, the robot will not be able to generalize to new scenarios. This is, the robot will avoid the obstacles in the scenarios used during the learning process. However, if the scenario changes after the learning process is finished, the policy will not be able to avoid the obstacles.

Another advantage of not including absolute distances is that the q-table is much smaller. Thus, it is easier to visit all the states enough times so that the q-table converges to the optimal policy. Additionally, there is no need to start the learning process far away from the destination since the longitudinal distance has been eliminated. In case of a definition of a final reward, this is also very beneficial, because sometimes the probability to see this final reward at the destination is very low if the path to travel is very long. The shorter the path to the destination is, the easier to get the final reward and the easier to converge to the optimal policy.

Finally, the reward strategy is also very simple, as it considers only safety requirements. The robot will be rewarded with -1 if it hits any obstacle or leave the circuit boundaries defined by the border frame (safety requirements). If the robot hits any obstacle, leaves the circuit, or reaches the destination, the learning episode will be terminated. Figure 5.1 shows the discrete grid where the vehicle can move along. Depending on the action taken, the vehicle can jump to the spot in front of the actual pose, the spot in the front-left location, or the spot in the

front-right location. The red spots of Figure 5.1 denote the static obstacles and the gray spots denote the starting and destination poses. The numbers in bracket denote the absolute lateral distance and the absolute longitudinal distance. The absolute lateral distance can take any integer number from -2 to 2. On the other hand, the absolute longitudinal distance can take any integer number from -1 to 19. The vehicle will leave the road boundaries if the lateral distance is either -2 or 2. In this case, the road boundaries are considered like walls and therefore treated like static obstacles.

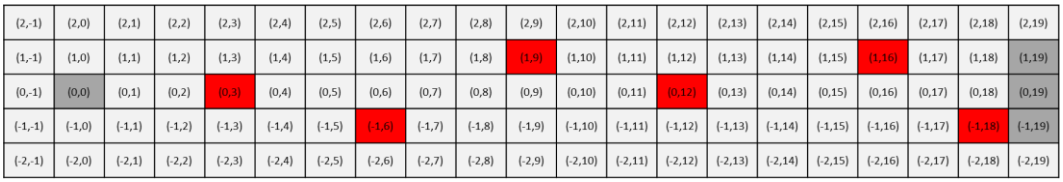


Figure 5.1: Discretized environment. The red spot are obstacles, and the grey spots are the starting and destination poses. The first number in bracket is the lateral distance with respect to the midline, and the second one is the longitudinal distance

After the learning process, the calculated policy has been validated with other obstacle distributions different from the one used during the learning process to prove that the policy is independent of the circuit or obstacle distribution used during learning. These new obstacle distributions have not been seen yet by the agent during the learning process, so that we can test if the policy can generalize to new unseen situations. The path travelled by the vehicle is highlighted in green in the following figures. The vehicle not only avoids all the obstacles, but also reaches the destination maximizing the cumulative reward.

- Obstacle distribution used during the learning process

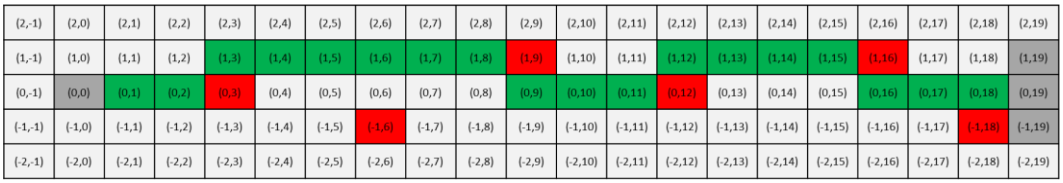


Figure 5.2: Travelled path (green) for the obstacle distribution used during the learning process. The obstacles are marked in red. Starting and destination poses are marked in grey

- Obstacle distribution 1 not seen during the learning process

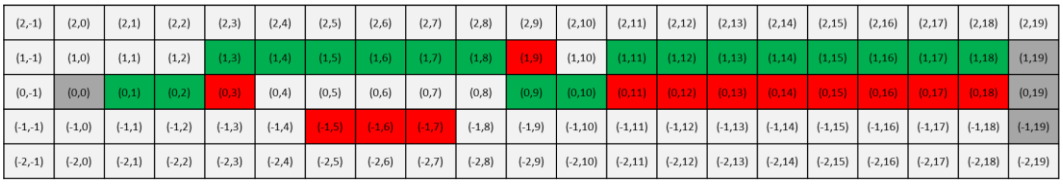


Figure 5.3: Travelled path (green) for obstacle distribution 1. The obstacles are marked in red. Starting and destination poses are marked in grey

- Obstacle distribution 2 not seen during the learning process

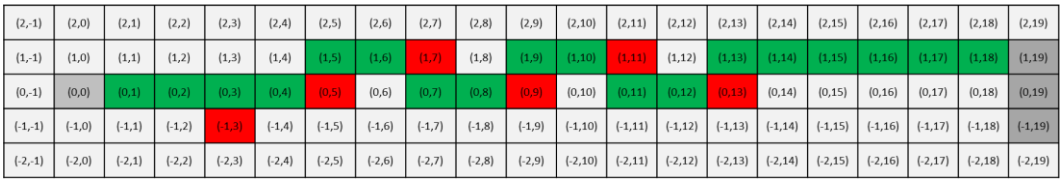


Figure 5.4: Travelled path (green) for obstacle distribution 2. The obstacles are marked in red. Starting and destination poses are marked in grey

- Obstacle distribution 3 not seen during the learning process

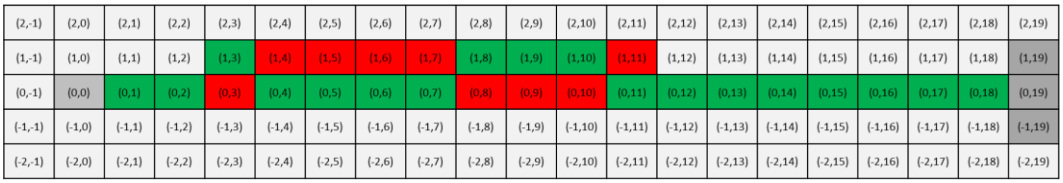


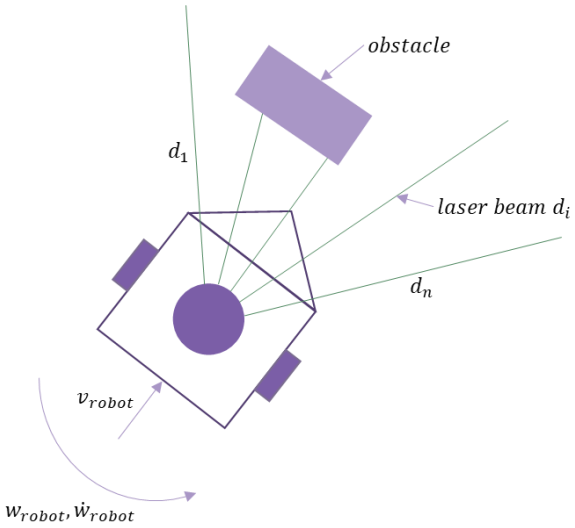
Figure 5.5: Travelled path (green) for obstacle distribution 3. The obstacles are marked in red. Starting and destination poses are marked in grey

The results show a very good behavior of the optimal policy, not only for the scenarios used during the learning process, but also for unseen scenarios by the agent during the learning process. Therefore, the characteristic of generalization of the optimal policy has been accomplished for this simplified setup with the q-learning algorithm.

5.2. Differential robot

In this section, the robot driving around the environment is a differential robot. Unlike the previous section, the model of the robot is a continuous function. Thus, the q-learning algorithm is not any longer suitable for this type of environment. A reinforcement learning algorithm which can handle continuous states and actions is needed under this section. The deep deterministic policy gradient (DDPG) is a good approach for this problem.

The continuous states variables are highlighted in Figure 5.6. Only relative distances of the laser beams from the robot to the obstacles are considered. A study of how many laser beams are to be considered to learn an optimal policy is performed in this work. Apart from the relative distances of the laser beams, the robot linear and angular speeds are considered. Additionally, the angular acceleration is also considered as state variable. All distances, speeds and accelerations are normalized between -1 and 1 (or 0 and 1). The normalization is important to bring stability during the learning process.



$$S = \{d_1, .., d_n, v_{robot}, w_{robot}, \dot{w}_{robot}\}$$

Figure 5.6: State variables of a differential robot

5.2.1. Circuit with obstacles

The first scenario to be considered is a circuit with obstacles. The circuit is surrounded by walls and the robot is supposed to drive always forwards avoiding crashing into the walls or obstacles. Considering the scenario only with the walls and without the obstacles along the way, it would be equivalent for validation the autopilot for front steering vehicles. The front steering vehicle detects the left and right lanes, which are equivalent here to the left and right walls. Figure 5.7 shows a schematic of the scenario used in CoppeliaSim.

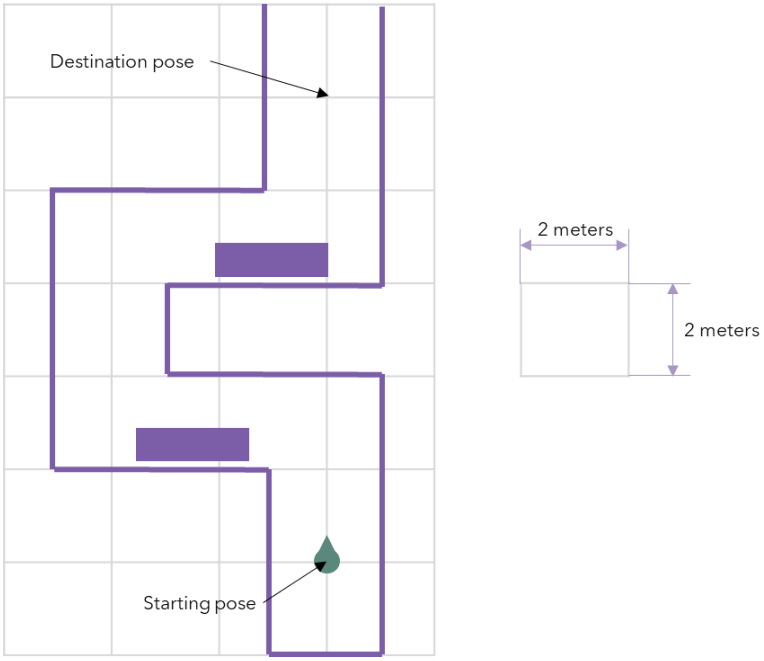


Figure 5.7: Circuit with obstacles 1

As shown in Figure 5.7, there are two obstacles along the way. One on the left side of the robot, and another one on the right side. The robot must reach safely

the destination pose. For that, only the safety requirements will be considered under this experiment. The safety requirements are represented by a simple, linear reward function (see section 4.4.1). Having the reward function defined, the DDPG algorithm will update the actor and critic networks to maximize the cumulative reward. At the beginning of the learning process, the actor and critic networks are initialized randomly, so the robot will hit the walls. As the robot drives, the state variables and actions are recorded by the DDPG algorithm and used to learn a new actor and critic models which perform better than the old ones. To see how the algorithm records and updates the actor and critic networks, see section 3.2. Every time that the robot hits an obstacle or wall, the learning episode gets terminated. Since we are using offline learning for this scenario, the position of the robot is reset back to the starting pose, wherever the robot is located along the circuit when the episode gets terminated. The episode also gets terminated if the robot reaches the destination pose. The goal of this simplified scenario is to determine the optimal number of laser beams to be considered in the state variables. We are going to analyze 3 different possibilities: 180, 45 or 8 laser beams. The configuration of the robot and state variables (number of laser beams), as well as the parametrization of the DDPG method is shown in Table 5.1:

Scenario	Circuit with obstacles 1
Sensors	Lidar + IMU
Number of laser beams	180
State variables	Normalized laser distances and normalized angular velocity
Actions	Angular velocity
Maximum number of episodes	100
Exploration	Gaussian noise $\mathcal{N}(0,0.5)$
Exploration decay	0.99
Actor network hidden layers	(500,500,500)
Critic network hidden layers	(500,500,500,500)
Actor learning rate	0.00001
Critic learning rate	0.0005
Safety requirements	$d_{limit} = 1$
Type of learning	Offline learning

Table 5.1: Robot configuration with 180 laser beams

Figure 5.8 shows the path travelled by the robot. As we can see, the robot avoids all the obstacles safely. From Table 5.1 we can read that the safety requirements generate negative rewards if the robot travels closer than 1 meter away from an obstacle (d_{limit} parameter, see section 4.4.1). Since the distance between walls in the circuit is set to 2 meters, as shown in Figure 5.8, the robot will get maximum reward if it travels along the middle lane of the circuit. Therefore, the path travelled by the robot (blue line) shows a correction when an obstacle is coming, adapting the trajectory from the middle lane between walls to the middle lane between obstacle and wall.

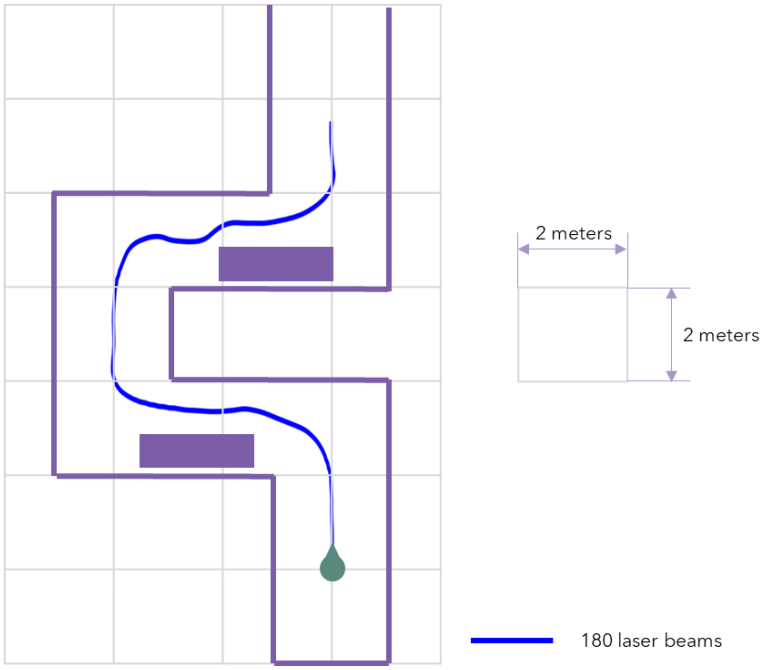


Figure 5.8: Travelled path. Circuit with obstacles 1, robot with 180 laser beams. The robot collects more cumulative reward when it travels along the midline. Thus, there is a sharp turn at the beginning of the second obstacle to follow the midline between wall and obstacle

Figure 5.9 shows the commanded angular speed and the angular acceleration.

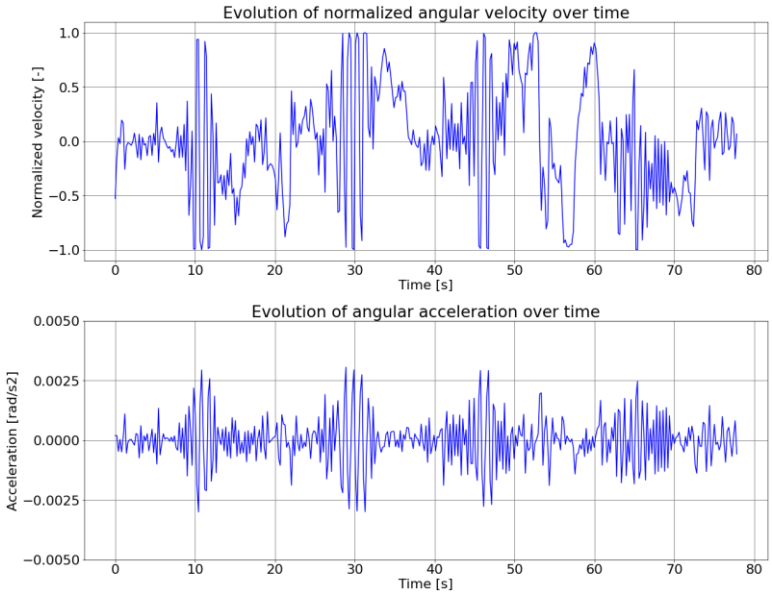


Figure 5.9: Angular speed and acceleration. Circuit with obstacles 1, robot with 180 laser beams. The angular speed and acceleration correspond to the path of Figure 5.8

It can be seen from both graphs that the angular speed and therefore the angular acceleration are very aggressive. This can not only damage the actuators of the robot, but it also feels very uncomfortable in case of any passenger travelling inside the robot. The reason why the commanded angular speed looks so aggressive is because the comfort requirements are still not included in this section. We will see later that the policy will produce a much smoother response to fulfil the comfort requirements and extend the life of actuators.

Once the actor and critic networks are determined after the learning process, an additional scenario has been prepared to validate the policy and check if it is capable to generalize and solve unseen scenarios. The new scenario is also a circuit surrounded by walls. However, this scenario has 3 different traps that the robot must successfully overcome:

1. U-trap
2. Zigzag
3. Bottle neck

The robot is expected to turn soon enough to the right direction to avoid getting trapped in the U-trap. The zigzag is a sequence of obstacles where the robot must react very quickly in order not to hit the obstacles. Finally, the robot must still be able to drive at the middle lane of the bottle neck to maximize the cumulative reward. A bird view of the scenario is shown in Figure 5.10.

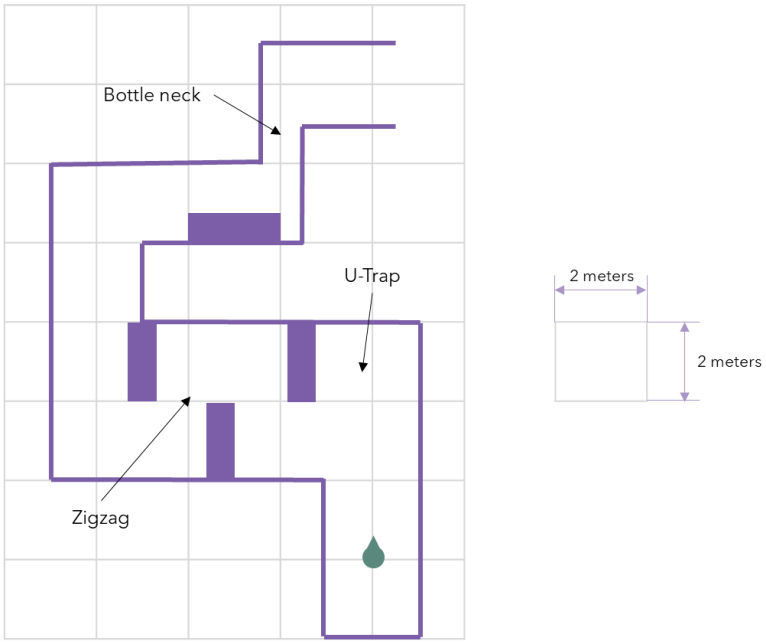


Figure 5.10: Circuit with obstacles 2

The results show that this configuration of 180 laser beams can successfully overcome the 3 different traps and the policy can generalize and solve these tricky situations. The path travelled by the robot can be checked in Figure 5.11. The path travelled (blue line) tries always to be located at the middle distance between obstacles. However, at the beginning of the U-trap, the robot does not manage to travel at the middle lane. This is because the robot gets confused about which path to take until the very end, when the robot discovers that there is more free space turning to the left than turning to the right, since the laser readings from the left side report longer distances. This is when the robot finally turns to the left, getting a little bit closer to the U-trap obstacle but without colliding into it (otherwise the simulation would have been terminated). A slightly pitching of the robot's heading can also be seen when the robot performs a 90-degree turn. This effect will eventually vanish if more learning episodes are considered during the learning process. This is because the robot, by means of the

exploration, will find out that turning earlier in the direction of the curve will get earlier the heading in line with the straight lane, and therefore the cumulative reward will decrease.

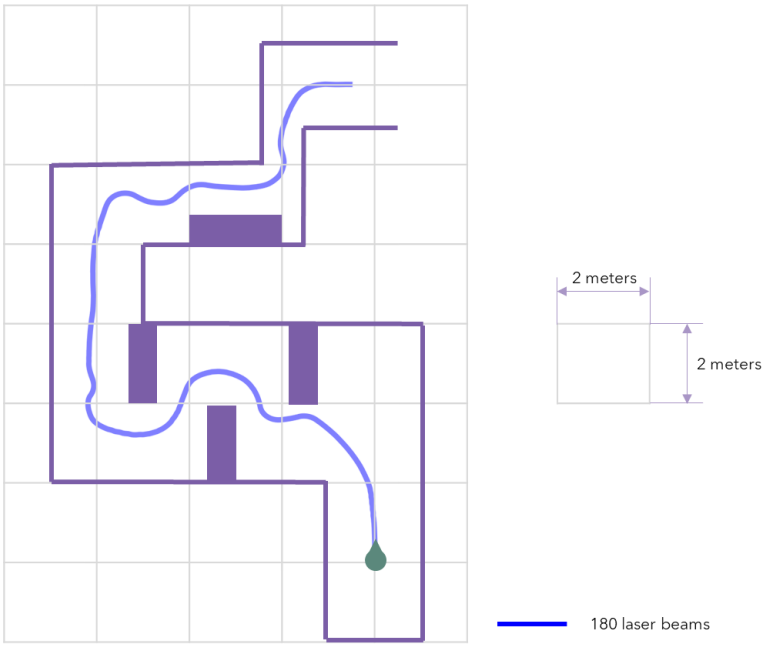


Figure 5.11: Travelled path. Circuit with obstacles 2, robot with 180 laser beams. The robot gets closer to the first obstacle due to a late detection of the true free space. The pitching of the heading can be improved considering more learning episodes and loading more scenarios during the learning process

The second configuration of the robot under this scenario will consider 45 laser beams. Table 5.2 shows the parametrization of this second environment:

Scenario	Circuit with obstacles 1
Sensors	Lidar + IMU
Number of laser beams	45
State variables	Normalized laser distances and normalized angular velocity
Actions	Angular velocity
Maximum number of episodes	100
Exploration	Gaussian noise $\mathcal{N}(0,0.5)$
Exploration decay	0.99
Actor network hidden layers	(50,50,50)
Critic network hidden layers	(50,50,50,50)
Actor learning rate	0.00001
Critic learning rate	0.0005
Safety requirements	$d_{limit} = 1$
Type of learning	Offline learning

Table 5.2: Robot configuration with 45 laser beams

The results show a good behavior of the policy under the scenario used for the learning process (Figure 5.12), but if we compare the blue line (path travelled by the robot with 180 laser beams) with the red line (path travelled by the robot with 45 laser beams), the red line does not perfectly follow the midline. The fact that the robot now does not perfectly follows the midline is not only due to a reduction

of laser beams. The number of neurons in the hidden layers have also been reduced, and they play an important role to produce high non-linear solutions. Therefore, the combination of less laser beams and, above all, the least number of neurons in the hidden layer, does not allow the robot to perfectly follow the midline.

Regarding the validation scenario, the robot gets trapped in the U-trap, as shown in Figure 5.13. Here the number of laser beams is highly important, as the free space on the left part of the U-trap cannot be detected with 45 laser beams due to a too high angle between two consecutives laser beams. This high angle represents a penalization if the robot wishes to detect free spaces far away from the robot's current position.

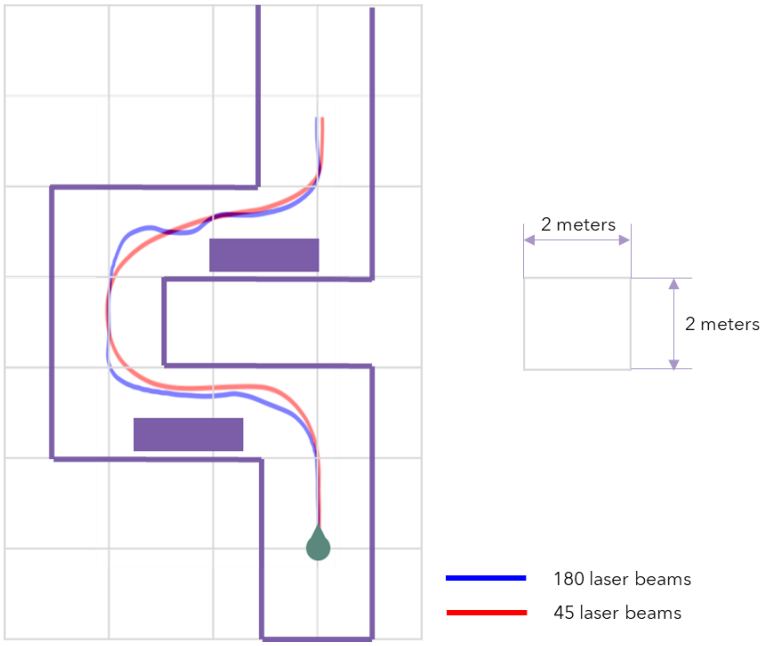


Figure 5.12: Travelled path. Circuit with obstacles 1, robot with 45 laser beams (red) and robot with 180 laser beams (blue). The 45 laser beams robot cannot perfectly follow the midline due to a reduction of laser beams and a reduction of the number of neurons in the hidden layers of the policy and critic networks

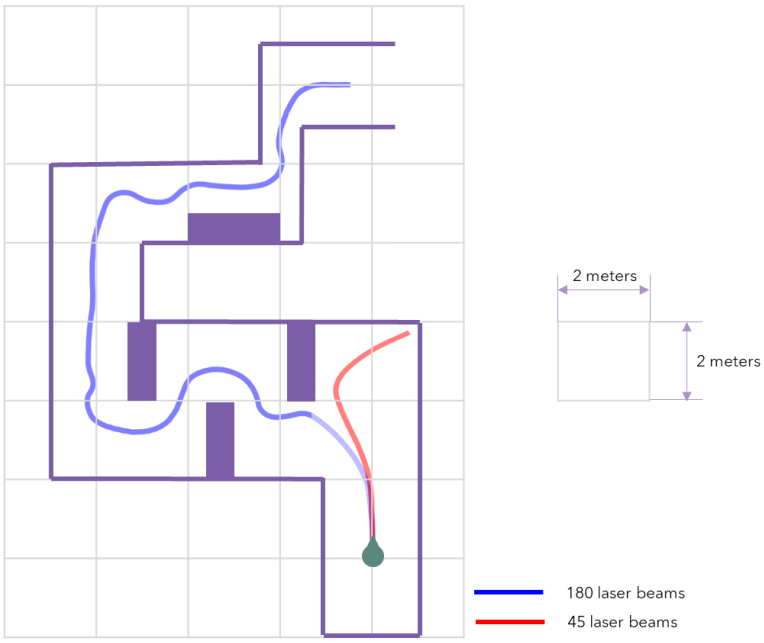


Figure 5.13: Travelled path. Circuit with obstacles 2, robot with 45 laser beams (red) and robot with 180 laser beams (blue). The 45 laser beams version is not capable to detect the free space at the left part of the first obstacle

Finally, the third configuration evaluates only 8 laser beams. The parametrization is shown in Table 5.3:

Scenario	Circuit with obstacles 1
Sensors	Lidar + IMU
Number of laser beams	8
State variables	Normalized laser distances and normalized angular velocity
Actions	Angular velocity
Maximum number of episodes	100
Exploration	Gaussian noise $\mathcal{N}(0,0.5)$
Exploration decay	0.99
Actor network hidden layers	(50,50,50)
Critic network hidden layers	(50,50,50,50)
Actor learning rate	0.00001
Critic learning rate	0.0005
Safety requirements	$d_{limit} = 1$
Type of learning	Offline learning

Table 5.3: Robot configuration with 8 laser beams

The results are very similar to the case of 45 laser beams, as we can check from Figure 5.14 and Figure 5.15. For the first obstacle scene, the path travelled by the robot of 180 laser beams (blue path) is very similar to the case of 45 laser beams (red path) and 8 laser beams (green path). In the U-trap, both 45 and 8 laser beams configurations get trapped.

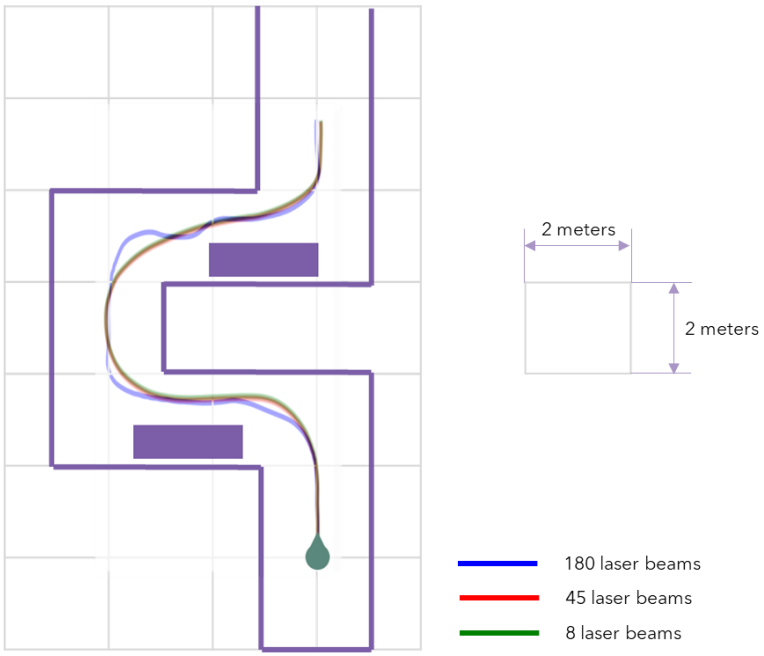


Figure 5.14: Travelled path. Circuit with obstacles 1, robot with 8 laser beams (green), robot with 45 laser beams (red) and robot with 180 laser beams (blue). The 8 and 45 laser beams versions output similar results

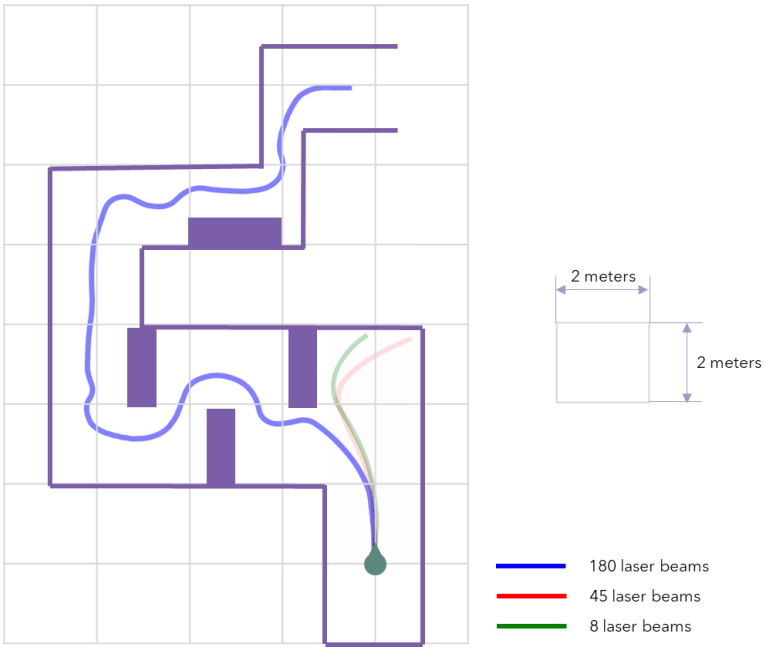


Figure 5.15: Travelled path. Circuit with obstacles 2, robot with 8 laser beams (green), robot with 45 laser beams (red) and robot with 180 laser beams (blue). The 8 and 45 laser beams versions output similar results

Therefore, the best configuration for the laser sensor are 180 beams since it has a better capability to detect small obstacles and thus a lower probability to get trapped between obstacles, such as the U-trap of the previous scene. The reason is because the angle between consecutive laser beams in the case of a total number of 45 or 8 is too large and a small obstacle can remain unseen between 2 consecutive laser beams, leading the robot to take the wrong direction and getting trapped in the U-trap. Since it has been proved that 180 laser beams are

the best configuration for a proper and fine control of the robot, we are going to consider 180 laser beams for the next experiments.

5.2.2. Circuit without obstacles

The goal of this second experiment is to introduce in a stepwise manner the different requirements defined in section 4.4 (safety, comfort, legal and task-oriented) and prove that the logic is working as intended.

The scenario has no obstacles apart from the right and left walls. The robot is intended to follow the path indicated by the walls without crashing into them. Figure 5.16 shows a schematic of the scenario used under this section:

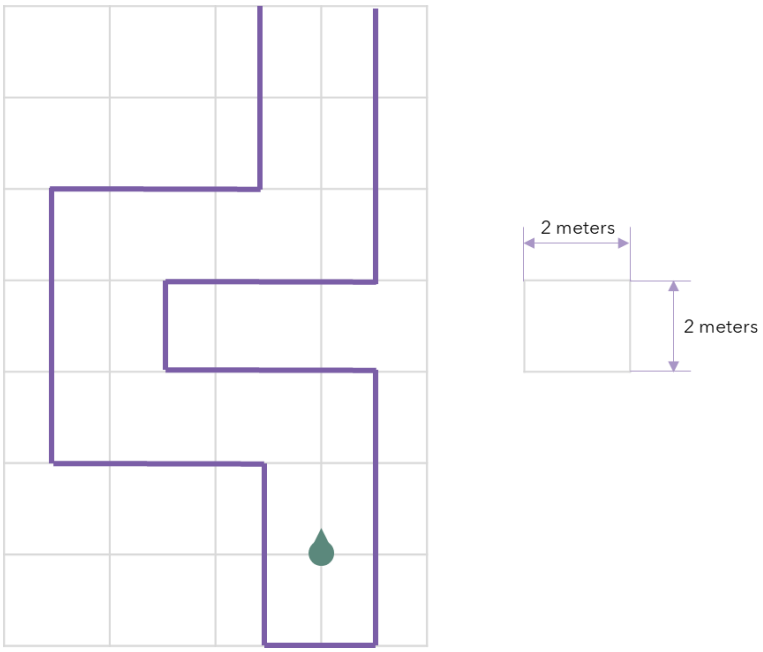


Figure 5.16: Circuit without obstacles

There are 4 different test cases proposed in this section:

1. Single action. Safety requirements: An agent which controls only the robot angular velocity. The linear velocity is fixed to a predefined value. The requirements to be considered under this subsection are only the safety requirements.
2. Multiple actions. Safety + legal requirements: An agent which controls the robot angular velocity and the linear velocity. The requirements to be considered in this test case are the safety and legal requirements.
3. Multiple actions. Safety + legal + comfort requirements: An agent which controls the robot angular velocity and the linear velocity. The requirements to be considered in this test case are the safety, legal and comfort requirements.
4. Multiple actions. Safety + legal + comfort + task-oriented requirements: An agent which controls the robot angular velocity and the linear velocity. The requirements to be considered in this test case are the safety, legal and

comfort requirements. Task-oriented requirement to stop the robot when it reaches the destination pose.

The results obtained in each test case are shown below:

1. Single action. Safety requirements

The angular speed of the robot is controlled by the agent. The agent will get a negative reward every time that the robot gets too close to an obstacle or to the circuit boundaries. The policy is considered to be mature enough once the conditions to complete the learning process have been fulfilled (see section 4.6). The path travelled by the robot with the resulted policy after the learning process is shown in Figure 5.17:

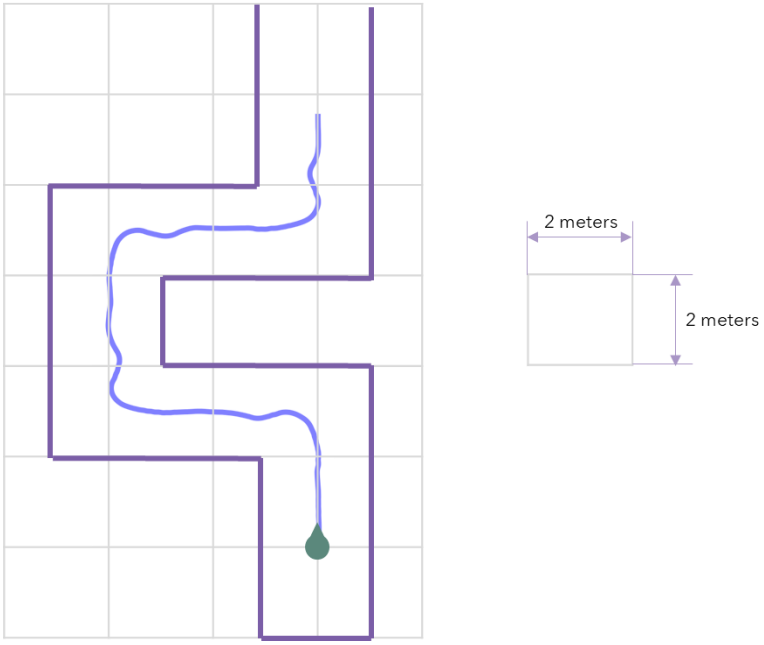


Figure 5.17: Travelled path. Circuit without obstacles, safety req. The robot collects more cumulative reward when it travels along the midline. The pitching of the heading can be improved considering more learning episodes and loading more scenarios during the learning process

As seen from Figure 5.17, the robot gets to the destination pose avoiding at every moment the circuit boundaries. Like the previous experiment (section 5.2.1), there is a slightly pitching of the robot's heading after the 90-degree curve. A longer learning process would have reduced this effect. On the other hand, the angular speed of the robot (and therefore also its derivative, the angular acceleration) is very nervous (see Figure 5.18). Since the comfort requirements are not yet applied during the learning process, the resulted policy in Figure 5.18 looks very aggressive.

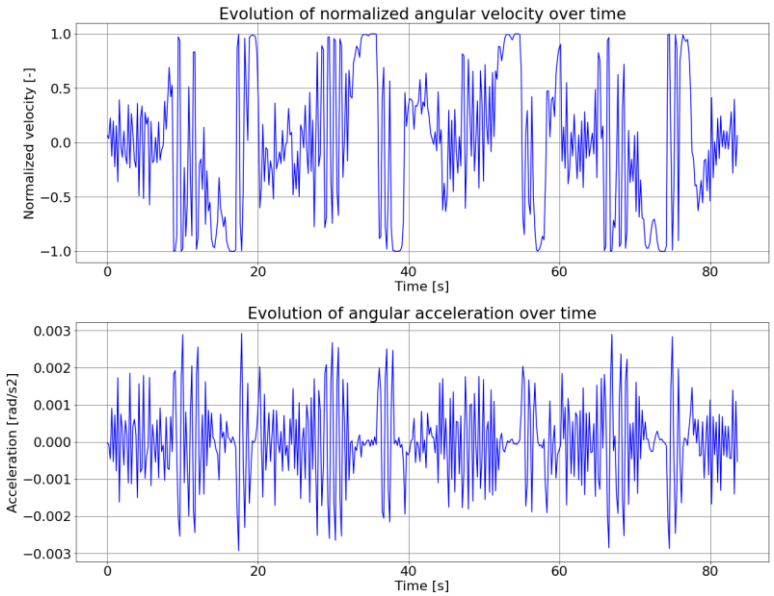


Figure 5.18: Angular speed and acceleration. Circuit without obstacles, safety req. The angular speed and acceleration correspond to the path of Figure 5.17

2. Multiple actions. Safety + legal requirements

Under this experiment the agent controls both the angular speed and the linear speed. Here, the robot gets penalized if the linear speed exceeds a predefined limit value. There is also a task-oriented requirement defined in this experiment: the robot shall drive close to the limit speed to try to reduce the travelling time. Therefore, the robot gets also penalized if the linear speed does not increase up to the target speed. The path travelled by the robot and the angular and linear velocities are shown in Figure 5.19 and Figure 5.20. There is still the pitching in the robot's heading, which can be corrected, as mentioned previously, with more exploration and learning episodes.

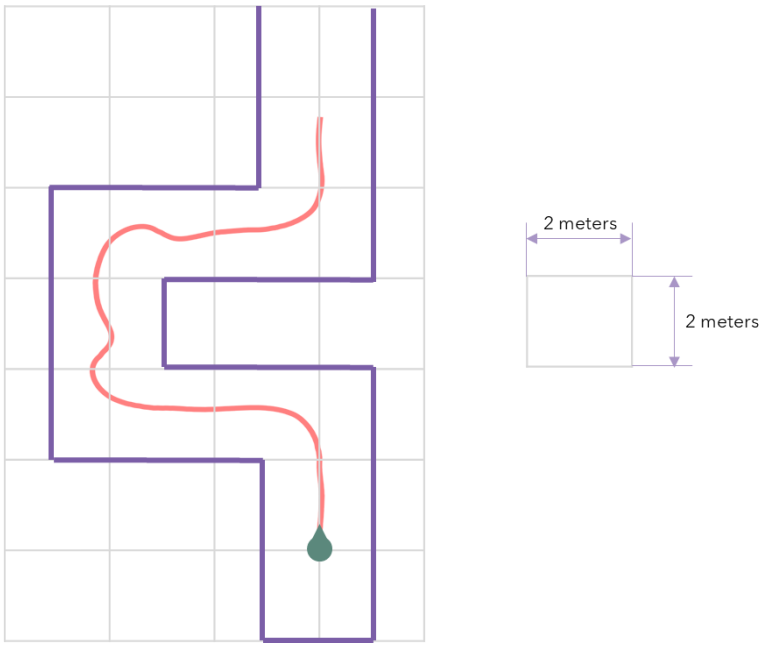


Figure 5.19: Travelled path. Circuit without obstacles, safety + legal req. The robot collects more cumulative reward when it travels along the midline. The pitching of the heading can be improved considering more learning episodes and loading more scenarios during the learning process

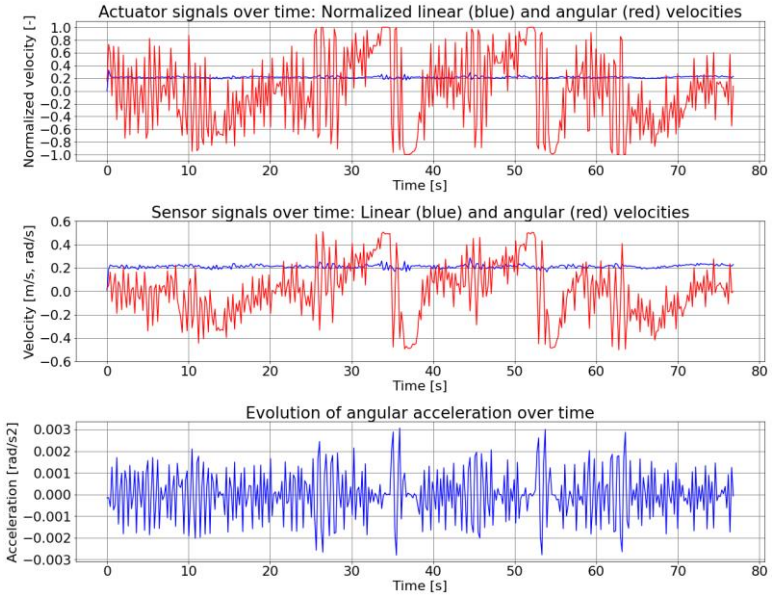


Figure 5.20: Angular speed, angular acceleration, and linear speed. Circuit without obstacles, safety + legal req. The angular speed, angular acceleration and linear speed correspond to the path of Figure 5.19

The plot shows in red the angular speed and in blue the linear speed. The top plot belongs to the commanded angular and linear speeds by the agent. The middle plot shows the measured angular and linear speed with the sensors mounted in the robot. Finally, the angular acceleration is also plotted in the bottom figure. As we can see, the differential robot always travels at 0.2 m/s, which is the maximum speed allowed under this scenario.

Regarding the state variables, the normalized linear speed has been included. This is because the reward must be a function of the state variables.

3. Multiple actions. Safety + legal + comfort requirements

The difference with respect to the previous experiment is the introduction of the comfort requirements. The robot gets penalized if the IMU sensor of the robot reads an angular acceleration higher than a predefined limit value. Again, the path travelled by the robot and the commanded signals by the agent are plotted in Figure 5.21 and Figure 5.22:

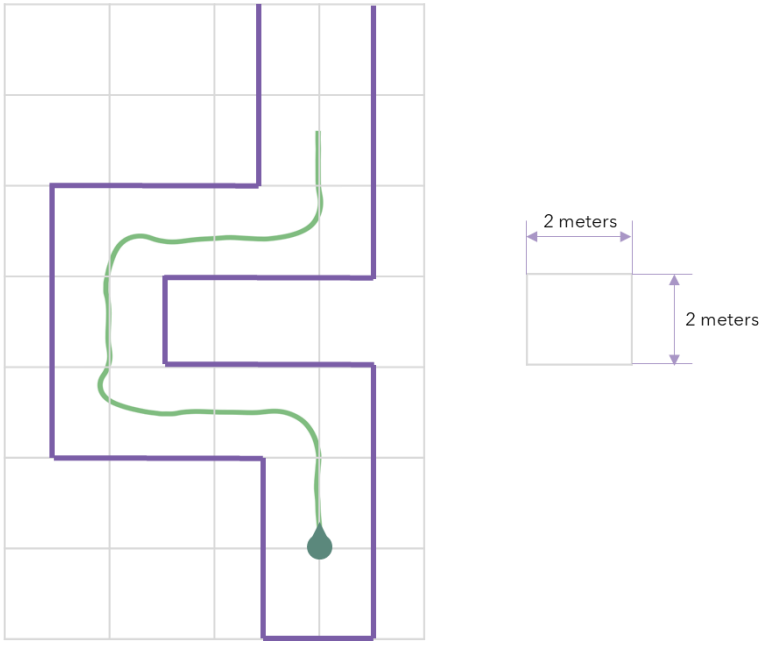


Figure 5.21: Travelled path. Circuit without obstacles, safety + legal + comfort req. The robot collects more cumulative reward when it travels along the midline. The pitching of the heading can be improved considering more learning episodes and loading more scenarios during the learning process, but it does not compromise the comfort because the angular acceleration values are below the limit, as seen in Figure 5.22

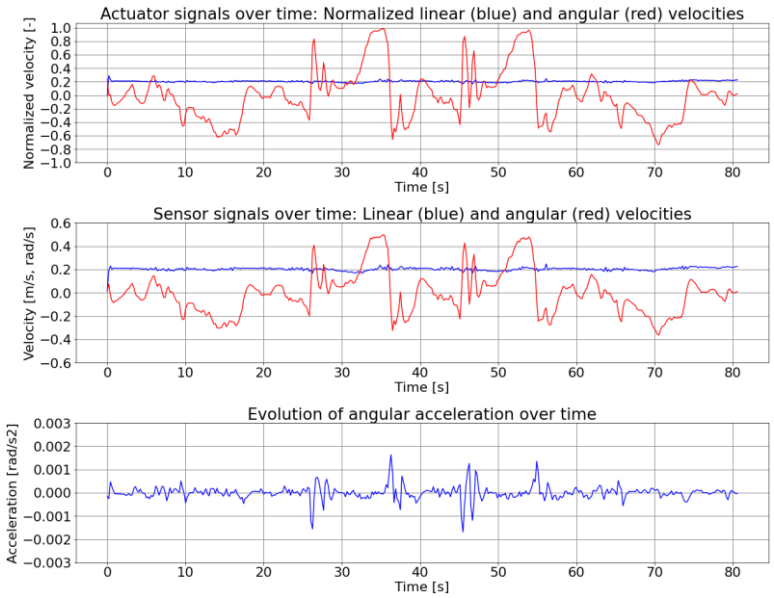


Figure 5.22: Angular speed, angular acceleration, and linear speed. Circuit without obstacles, safety + legal + comfort req. The angular speed, angular acceleration and linear speed correspond to the path of Figure 5.21

Now the robot reacts much smoother than in the previous experiments. The commanded signals sent to the actuators are smooth to avoid actuator damage and keep a good comfort feeling. The maximum acceleration limit where the robot starts to get penalized is 0.001 rad/s^2. Below this value, the robot does not get any penalization, as the comfort feeling is still ok for those low acceleration values. Additionally, the robot follows the midline perfectly. There is however some pitching in the robot's heading after the 90-degree curves. However, this pitching does not compromise the comfort, because the angular acceleration values are below the predefined limit. Anyway, more learning episodes will improve the pitching and reduce even further the acceleration values.

Regarding the state variables, the normalized angular acceleration has been included.

4. Multiple actions. Safety + legal + comfort + task-oriented requirements

Finally, a task-oriented requirement has been added to stop the robot exactly at the destination pose. The same reward function as in the second experiment has been used. However, there is a small modification to bring the robot velocity to 0 at the destination pose. The distance from the robot to the destination pose is measured. For example, this variable can be measured by a camera. If the destination pose is indicated by an object with a color that the camera can recognize, the perception module can calculate the relative distance from the robot to the destination pose. If the camera does not detect the target color, the distance is set to the maximum value (equal to 1 if this distance has been normalized). The robot target speed can be then ramped down to 0, as the robot approaches the destination pose.

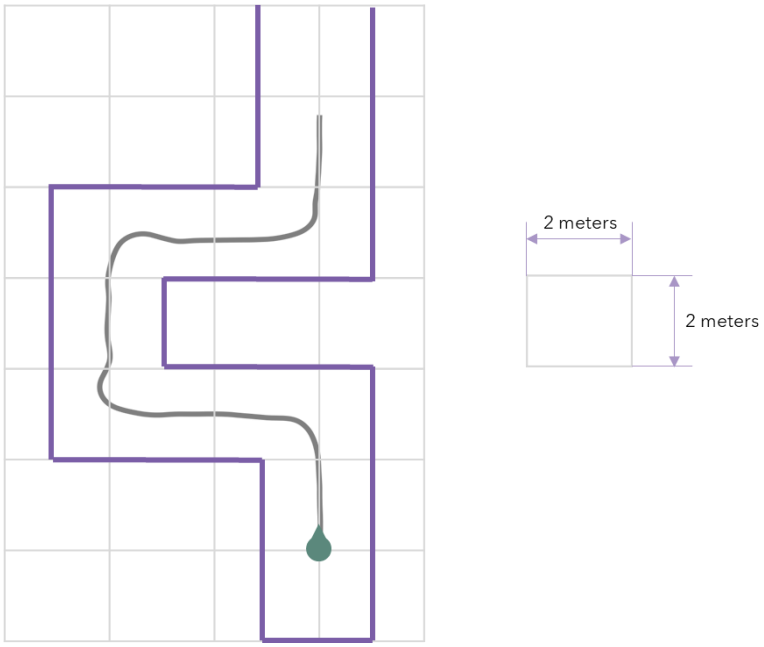


Figure 5.23: Travelled path. Circuit without obstacles, safety + legal + comfort + task-oriented req. The robot collects more cumulative reward when it travels along the midline. The pitching of the heading can be improved considering more learning episodes and loading more scenarios during the learning process, but it does not compromise the comfort because the angular acceleration values are below the limit, as seen in Figure 5.24

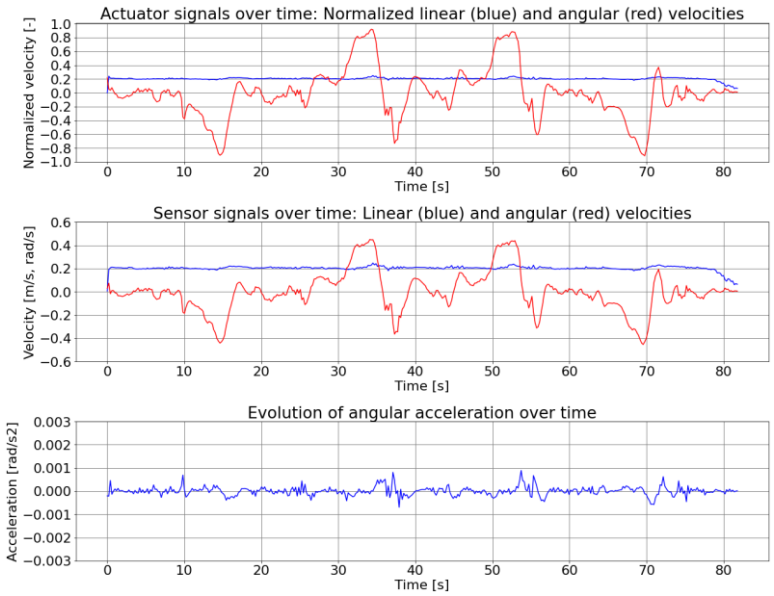


Figure 5.24: Angular speed, angular acceleration, and linear speed. Circuit without obstacles, safety + legal + comfort + task-oriented req. The angular speed, angular acceleration and linear speed correspond to the path of Figure 5.23

As we can check from Figure 5.24, the robot linear speed (blue line) gets down to 0 when the robot reaches the destination pose. Additionally, all other requirements are also fulfilled: the robot has not crashed into the circuit boundaries (safety requirements), the robot linear speed does not exceed the maximum predefined limit (legal requirements) and the angular acceleration does not exceed the maximum limit, above which there would be discomfort feeling (comfort requirements). The robot also follows the midline and the pitching of the robot's heading after the 90-degree curve does not affect the comfort, since the angular acceleration values are below the predefined limit.

The state variables also include the distance to the destination pose as an additional variable.

All 4 experiments have run until the episodic reward has been maximized. Since all reward functions have been defined as penalization, the maximum cumulative reward that the robot can obtained after one episode is 0. Figure 5.25 shows the episodic reward. As learning progresses, the episodic reward tends to 0.

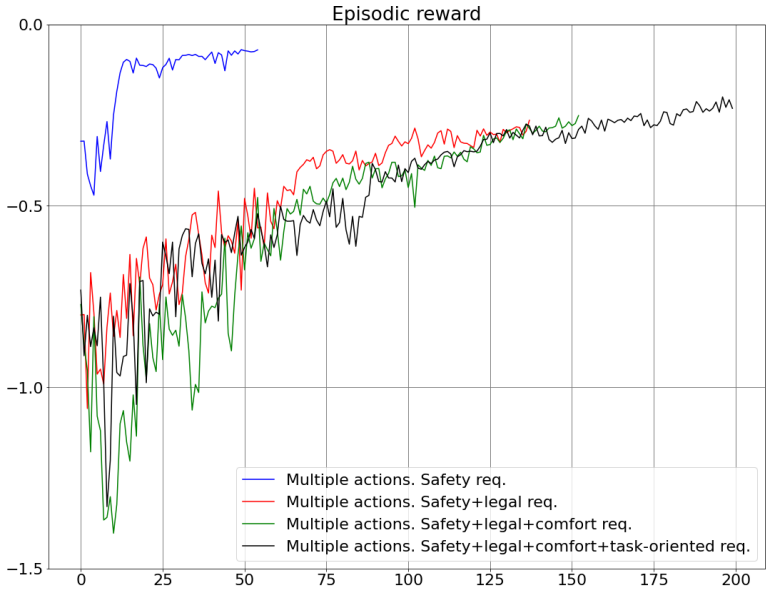


Figure 5.25: Episodic reward. Circuit without obstacles. The cumulative episodic reward keeps on increasing at the end of the plot, so there is still room for improvement if more learning episodes are considered

Another interesting variable is the time needed until the learning process is completed. Table 5.4 summarizes the time needed for the previous 4 experiments.

Experiment	Total time
Single action. Safety req.	2.0 hours
Multiple actions. Safety + legal req.	3.3 hours
Multiple actions. Safety + legal + comfort req.	3.7 hours
Multiple actions. Safety + legal + comfort + task-oriented req.	5.1 hours

Table 5.4: Learning total time

5.2.3. Circuit with dynamic obstacles

This scenario is like the previous ones, but in this case the obstacles are dynamic. They move around the circuit with a fixed speed and direction, so the level of unpredictability of the obstacles is low. Here we can see how reinforcement learning techniques can successfully solve complex driving scenarios, generating automatically a high mature policy by trial and error that can be generalized to new unseen situations, as will be proved later.

Figure 5.26 shows the configuration of the circuit, as well as the distribution of the static and dynamic obstacles. The dynamic obstacles are marked with green color. The position marked in the Figure 5.26 is the initial position. The magnitude and direction of the dynamic obstacles are also shown. In contrast with the previous scenarios, the distance of the safety requirements is set to 0.5 meters instead of 1 meter. In this way, the robot just gets penalized if it gets too close to the walls, in contrast with the two previous chapters that the robot was intended to travel along the middle lane between walls.

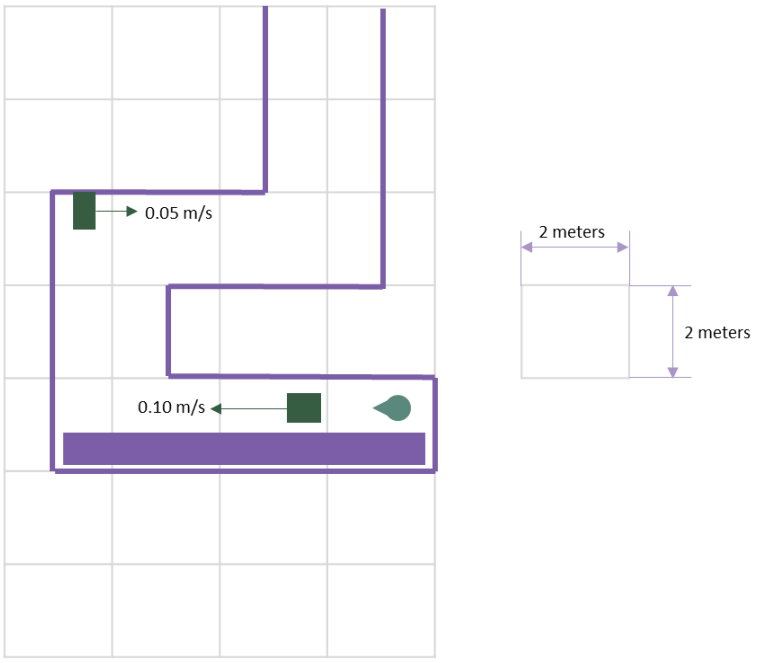


Figure 5.26: Circuit with dynamic obstacles

Figure 5.27 shows the path travelled by the robot inside the circuit. Intermediate positions of the dynamic obstacles are also shown in the picture.

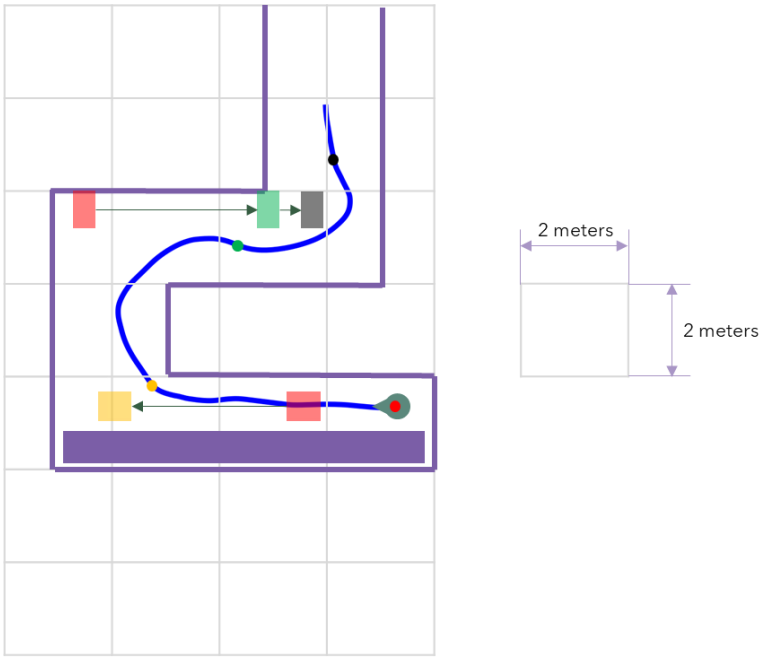


Figure 5.27: Travelled path. Circuit with dynamic obstacles. The color of the obstacles is assigned with a position in the robot's travelled path with a spot of the same color. In this experiment the robot is not intended to travel along the midline, like in the previous experiments. The robot just gets penalized if it gets too close to the obstacles. Therefore, there is no pitching anymore in the trajectory of the robot

The robot can adapt both the linear and angular speeds to reach safely the destination while avoiding dynamic and static obstacles. Figure 5.28 shows the linear and angular speeds of the robot.

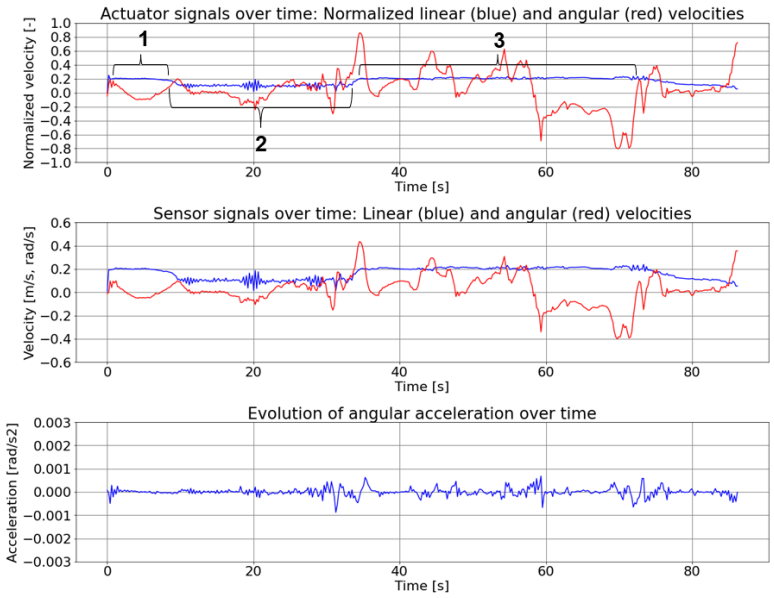


Figure 5.28: Linear and angular speeds and angular acceleration of the path shown in Figure 5.27. Circuit with dynamic obstacles. As seen in the acceleration plot, the robot travels under comfortable conditions. The 3 marks in the actuator plot are explained in detail in the text

There are 3 interesting areas marked in the linear speed plot in Figure 5.28. In the area number 1, the robot travels at maximum legal speed (0.2 m/s) until it gets trapped between the static and dynamic obstacles. Then, in area number 2, the robot reduces the linear speed to match the speed of the dynamic obstacle (0.1 m/s). In this way, the robot follows the dynamic obstacle at a fixed distance, therefore avoiding crashing. This situation ends when the robot sees enough free space to get through between the dynamic obstacle and the right wall. In this point (area number 3), the robot increases its linear speed again up to the legal speed, as there is no danger to collide with any obstacle. The second dynamic obstacle, which is travelling at 0.05 m/s, is overtaken by the robot travelling at maximum speed (0.2 m/s), as there is no necessity to decrease the linear speed to safely overtake the dynamic obstacle.

From Figure 5.28 we can also check that the comfort requirements are also fulfilled, and the angular speed looks smooth (the angular acceleration never goes higher than $0.001\ rad/s^2$, which is how the comfort requirement has been defined).

These successful results are obtained after the learning process. The episodic reward from Figure 5.29 shows how the episodic reward increases at the end of each episode after updating the policy. In this case, the total time needed to automatically learn the policy is 6.57 hours (the system info of the computer used during this learning process is mentioned in the introduction of section 5).

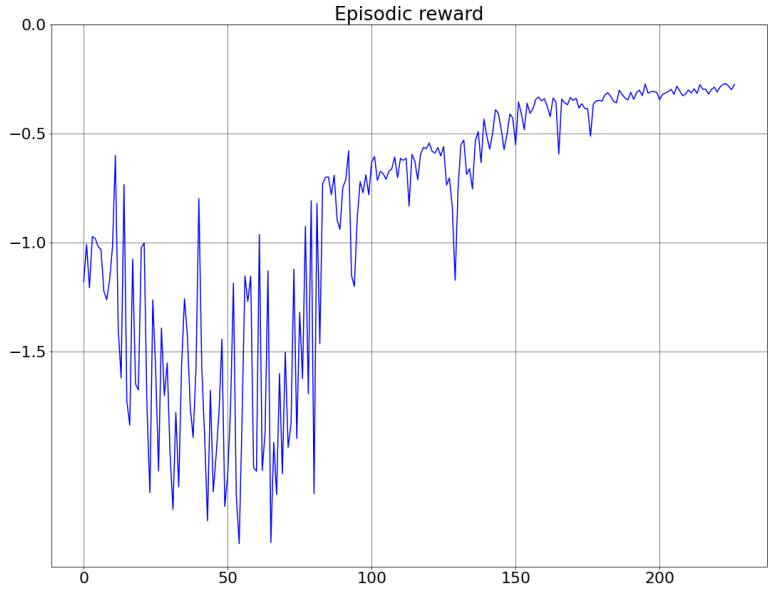


Figure 5.29: Episodic reward. Circuit with dynamic obstacles. The cumulative episodic reward seems very flat at the end of the episodes, but there is still room for improvement

So far, the robot seems to perform very good under the scenario used for the learning process. But it is very important to validate the policy under new unseen scenarios, to ensure that the policy can properly generalize and solve new unseen situations. For this purpose, a new and more complicated dynamic scenario has been set up. Figure 5.30 shows the configuration of this scenario.

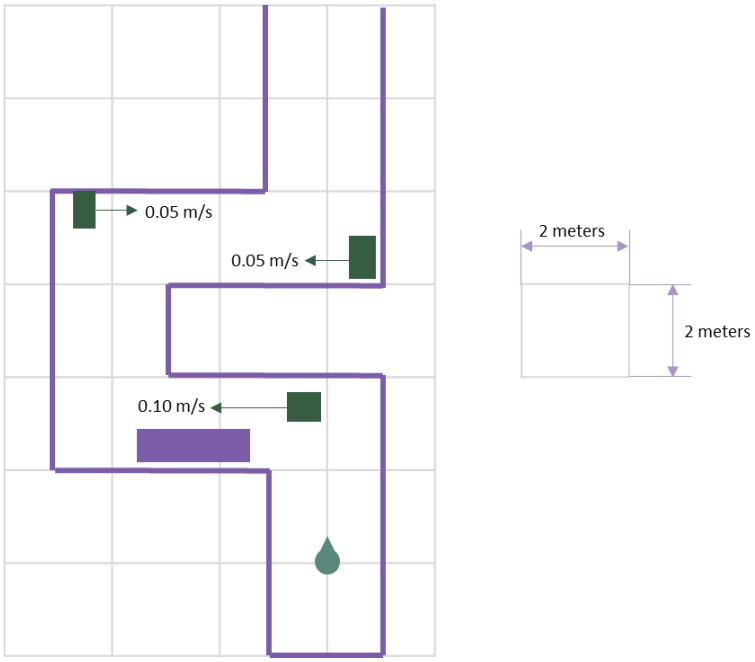


Figure 5.30: Circuit with dynamic obstacles 2

The path travelled by the robot and the intermediate positions of the dynamic obstacles are shown in Figure 5.31.

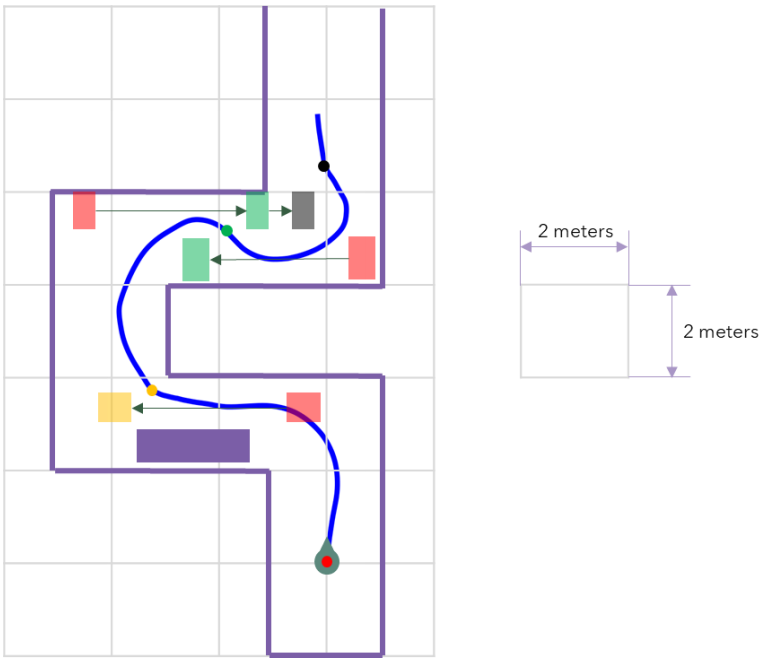


Figure 5.31: Travelled path. Circuit with dynamic obstacles 2. The color of the obstacles is assigned with a position in the robot's travelled path with a spot of the same color. In this experiment the robot is not intended to travel along the midline, like in the previous experiments. The robot just gets penalized if it gets too close to the obstacles. Therefore, there is no pitching anymore in the trajectory of the robot

The linear and angular speed profiles of the robot are plot in Figure 5.32.

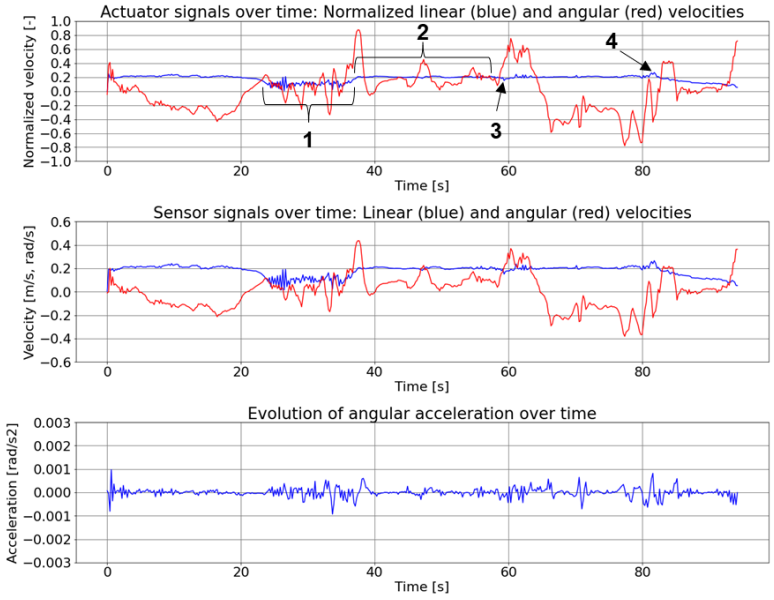


Figure 5.32: Linear and angular speeds and angular acceleration of the path shown in Figure 5.31. Circuit with dynamic obstacles 2. As seen in the acceleration plot, the robot travels under comfortable conditions. The 4 marks in the actuator plot are explained in detail in the text

As in the previous scenario, the robot adapts its linear speed to follow the first dynamic obstacle (area number 1), as the robot gets trapped between the walls, the static obstacle, and the dynamic obstacle. After the robot detects enough space to get though the wall and the dynamic obstacle, the robot increases the linear speed to the legal maximum speed (area number 2). In the point number 3, the robot gets shortly trapped again between the wall and the two dynamic

obstacles travelling at opposite speeds. Because of that, the robot shortly decreases the linear speed, therefore fulfilling the safety requirements. In point number 4, the robot gets squeezed between the dynamic obstacle and the right wall after turning in the corner. The robot accelerates shortly to move away from the dynamic obstacle, which was approaching the robot.

This chapter proves the effectiveness of the reinforcement learning techniques under dynamic scenarios, which are usually the most complex to solve. The overtaking and the follow up maneuvers have been successfully tested under the previous scenarios.

5.2.4. Room with obstacles

The fourth scenario considered is a room with obstacles. The goal of this scenario is to test and validate the online learning methodology and prove how the robot can drive in open spaces from a starting pose A to a destination pose B. The policy, which is learnt online as the robot drives, will eventually fulfil the safety, legal, comfort and task-oriented requirements. As the robot learns online, it is even possible to overcome problems such as actuator malfunction or sensor aging. For example, let's assume that the left wheel of the differential robot collects some dirt or dust, increasing the friction of the shaft. This increment in the friction can impact the behavior of the robot. If we think of the state-of-the-art solution provided by the lattice planner, the motion planning algorithm can calculate a trajectory that is no longer feasible and cannot be realized by the motion control software module. This is because the cost function is a fixed function, which does not have any input or information about any actuator malfunction or sensor aging. On the other hand, the online learning can adapt in an online way the policy, considering the new conditions of the environment (including actuator malfunction or sensor aging) to fulfil again the requirements. In case of a robot with redundant actuators (like a spider robot with multiple legs), the policy can still be adapted to fulfil the requirements even in case of completely actuator malfunction (completely lost of one leg due to electrical or mechanical damage). This characteristic is of special interest in applications where the robot cannot be driven home to get repaired, like a robot sent to a mission to another planet.

The scenario considered is limited by walls which indicates the open space where the robot can drive. Driving outside these limits is not possible. Inside this open space, the robot is initially located in a starting pose. The destination pose that the robot must reach can be either provided in form of a waypoint in the memory of the robot or a camera installed in the robot can detect the direction where the destination is located. In this work, the second variant has been used. The destination is marked with a special green color that the camera can recognize, therefore being able to calculate the target direction of the destination (see section 4.4.4). A schematic of the scene is depicted in Figure 5.33:

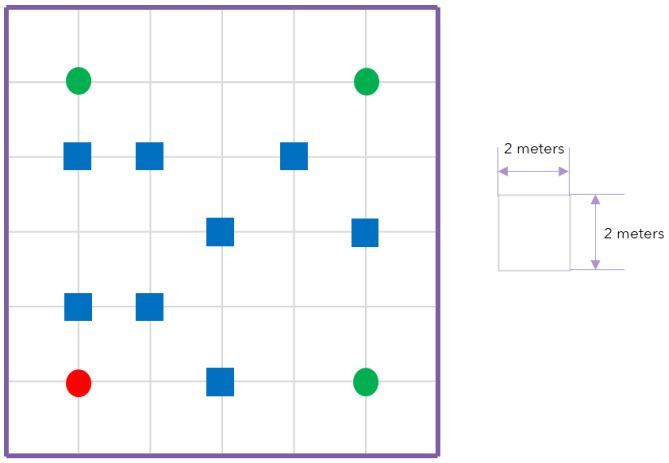


Figure 5.33: Room with obstacles 1

There are 2 test cases proposed in this section:

1. Safety + task-oriented requirements: the goal is to reach the destination (task-oriented requirement) avoiding all obstacles (safety requirements).
2. Safety + legal + comfort + task-oriented requirements: the goal is to reach the destination (task-oriented requirement) avoiding all obstacles (safety requirements), limiting the angular acceleration of the robot (comfort requirements) and not exceeding the maximum room speed (legal requirements) but travelling whenever is possible to that maximum speed to minimize travelling time (task-oriented requirement).

The results of these 2 test cases are explained below:

1. Safety + task-oriented requirements:

The robot starts the learning process at the starting pose. However, the robot's position does not get reset back to the initial pose if the learning episode gets terminated. Instead, if the learning episode gets terminated, the robot stops, backs up a predefined distance and turns around a predefined angle (see section 4.6). After this sequence, a new learning episode starts from the current robot's position. The learning process needs roundabout 200 episodes to get a policy mature enough to fulfil the requirements.

Once the policy has been determined, it gets validated under the same scenario. The robot travels from the starting pose A to the 3 possible destinations inside the room. The path travelled by the robot is shown on the left part of Figure 5.34, Figure 5.35 and Figure 5.36 and the linear and angular speeds commanded by the agent and measured are shown on the right part.

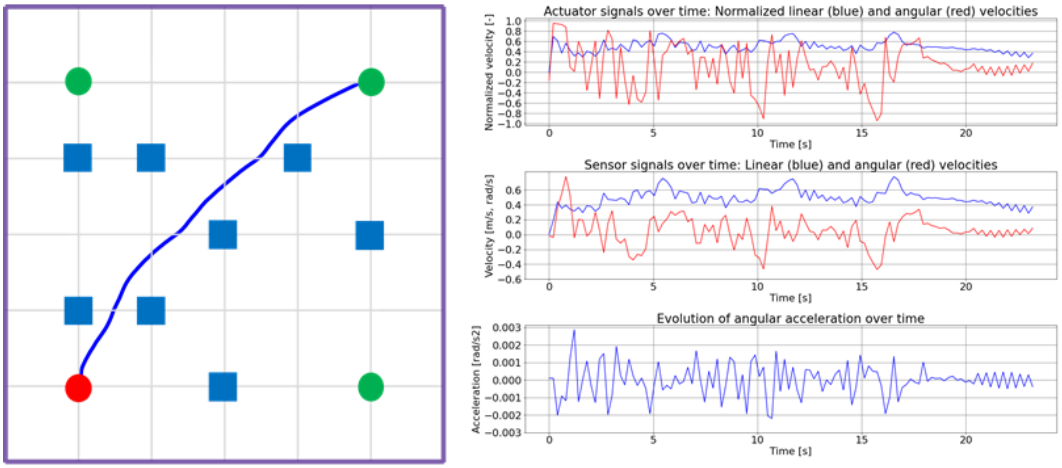


Figure 5.34: Path travelled. Room with obstacles 1, destination 1. Safety + task-oriented req. The destination, marked in green, is safely reached, but under uncomfortable conditions

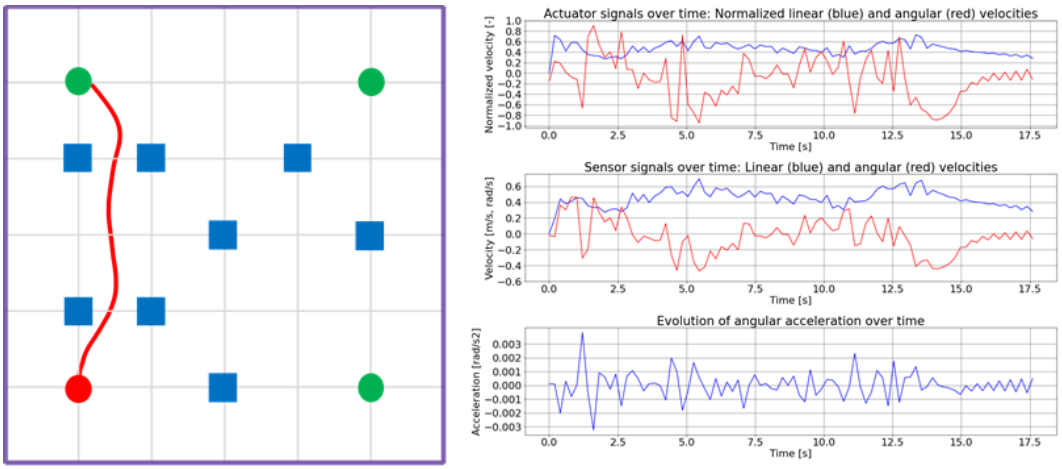


Figure 5.35: Path travelled. Room with obstacles 1, destination 2. Safety + task-oriented req. The destination, marked in green, is safely reached, but under uncomfortable conditions

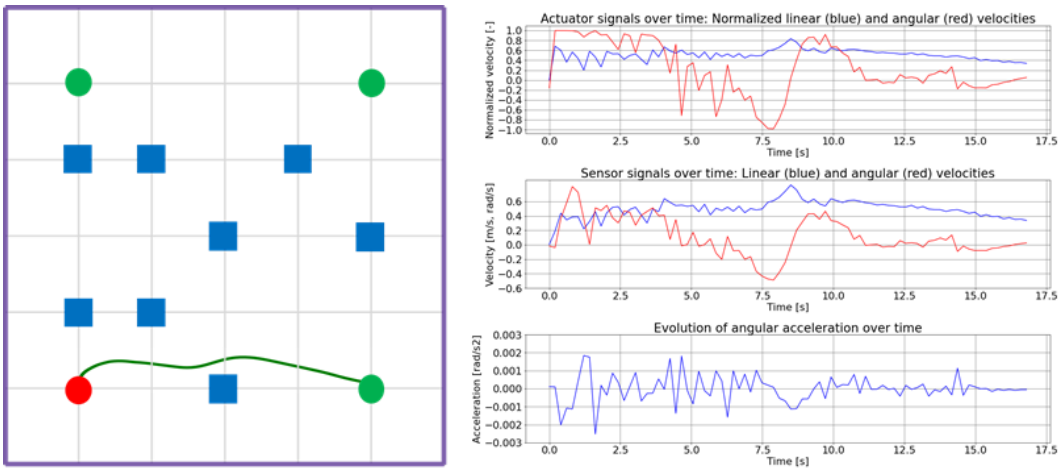


Figure 5.36: Path travelled. Room with obstacles 1, destination 3. Safety + task-oriented req. The destination, marked in green, is safely reached, but under uncomfortable conditions

The robot reaches all 3 destinations avoiding any contact with the obstacles. However, the commanded angular speed of the robot is very aggressive. The linear speed is also not limited to any value. The next test case will enable the comfort and legal requirements to solve these issues.

2. Safety + legal + comfort + task-oriented requirements:

This second test case adds a penalization for angular acceleration higher than 0.001 rad/s^2. In addition, the maximum room linear speed is set to 0.2 m/s. The resulted policy is tested under the same scenario of the learning process. Results are shown in Figure 5.37, Figure 5.38 and Figure 5.39 for the different destination poses:

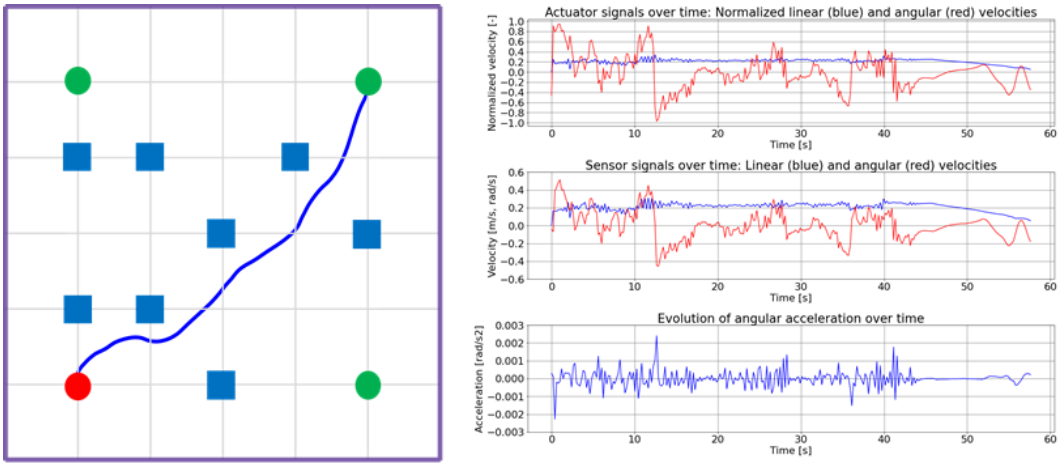


Figure 5.37: Path travelled. Room with obstacles 1, destination 1. Safety + legal + comfort + task-oriented req. Although the acceleration level has decreased with respect to the previous test case (safety + task-oriented req.), there is still room from improvement, as the episodic reward can be further decreased (red line from Figure 5.44)

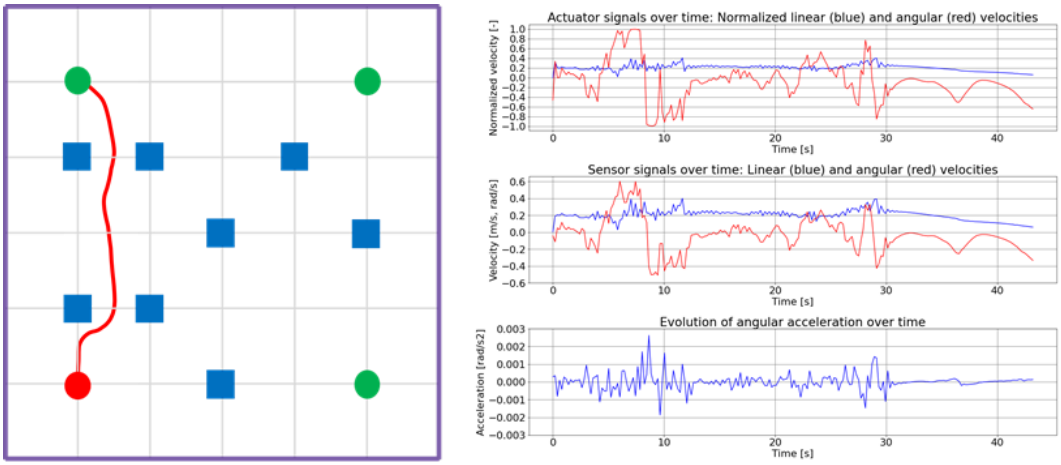


Figure 5.38: Path travelled. Room with obstacles 1, destination 2. Safety + legal + comfort + task-oriented req. Although the acceleration level has decreased with respect to the previous test case (safety + task-oriented req.), there is still room from improvement, as the episodic reward can be further decreased (red line from Figure 5.44)

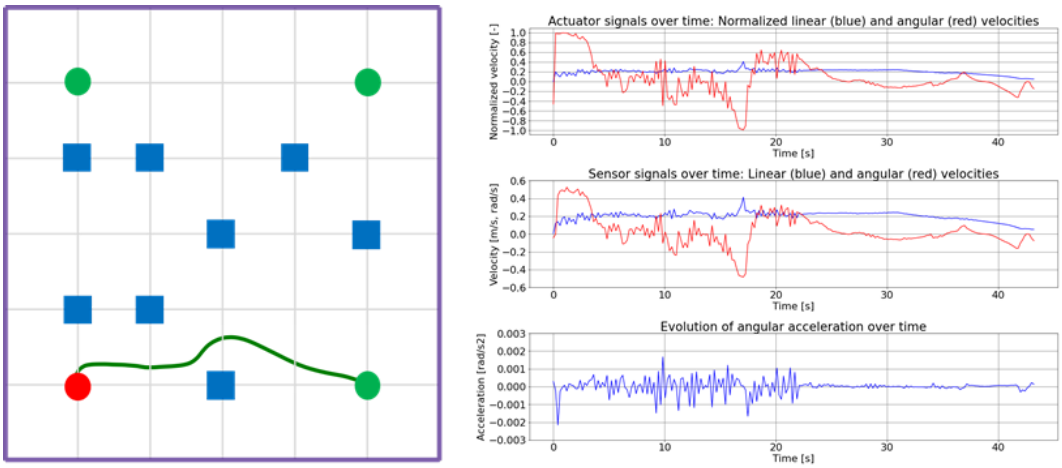


Figure 5.39: Path travelled. Room with obstacles 1, destination 3. Safety + legal + comfort + task-oriented req. Although the acceleration level has decreased with respect to the previous test case (safety + task-oriented req.), there is still room from improvement, as the episodic reward can be further decreased (red line from Figure 5.44)

The level of angular acceleration has significantly decreased in comparison with the previous case. However, there is still room for improvement and get a smother response by considering more learning episodes. This can be checked from the episodic reward plot, which is shown later in Figure 5.44.

The previous results confirm a good policy behavior under the same scenario as the learning process. But it is important to obtain a policy that can generalize to unseen scenarios. For that reason, a new scenario, which was not used during the learning process, has been used now for the validation of the resulted policy. The schematic of this new room distribution is in the following picture:

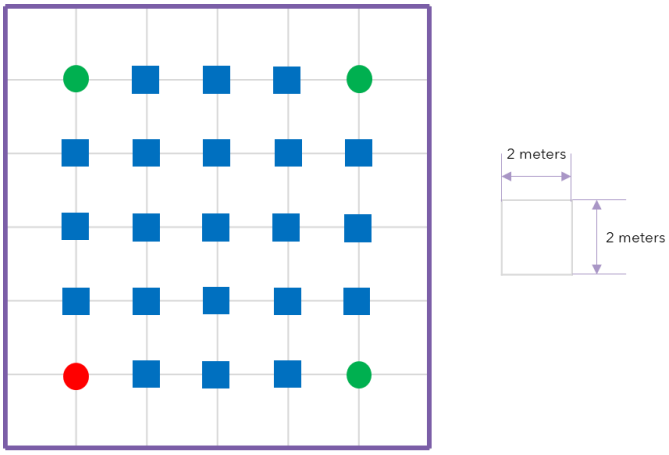


Figure 5.40: Room with obstacles 2

As before, the left part of Figure 5.41, Figure 5.42 and Figure 5.43 shows the path travelled by the robot and the right part the robot's speeds and angular acceleration:

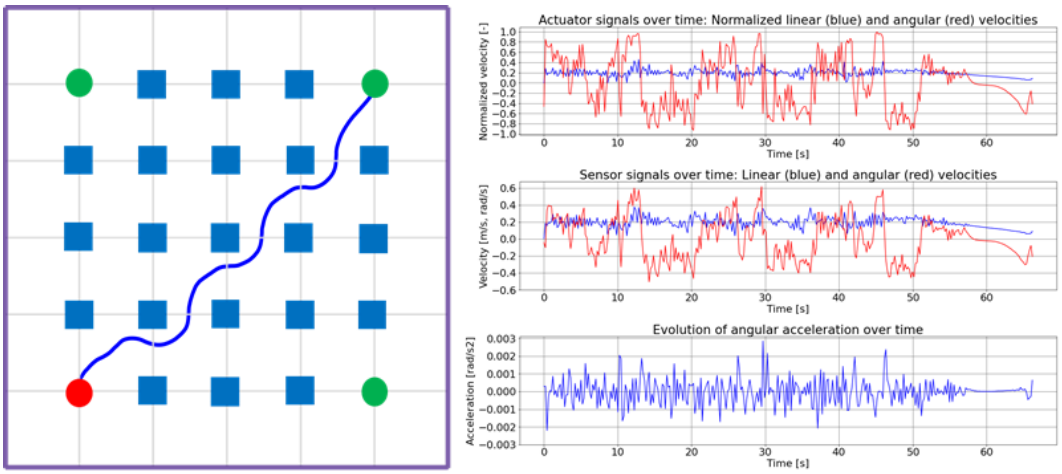


Figure 5.41: Path travelled. Room with obstacles 2, destination 1. Safety + legal + comfort + task-oriented req. Here the acceleration level has increasing in comparison with the scenario used for the learning process. One method to solve this problem is to include more scenarios during the learning process and therefore try to cover the complete range of measurements of the sensors

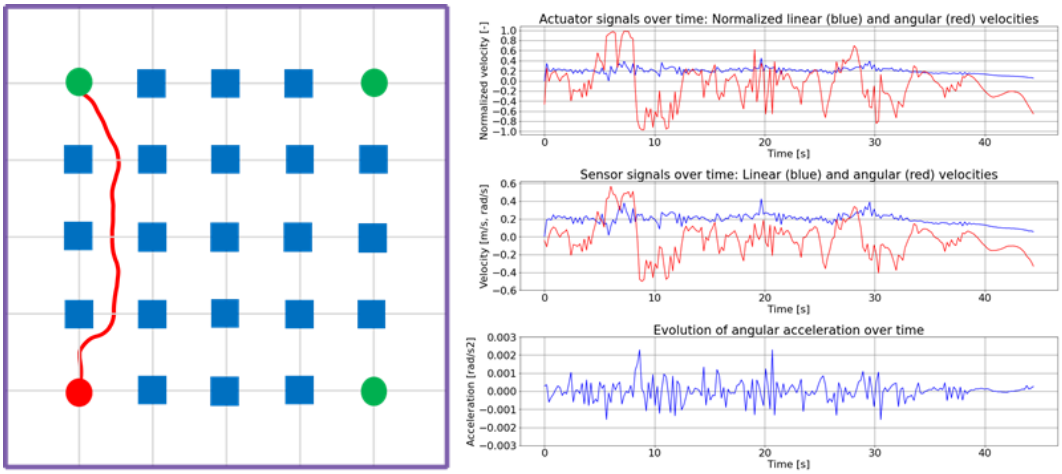


Figure 5.42: Path travelled. Room with obstacles 2, destination 2. Safety + legal + comfort + task-oriented req. Here the robot gets too close to the first obstacle but manages to avoid it without contact. It is very likely that this combination of observations has not been seen during the learning process and therefore the result seems to be not very optimized

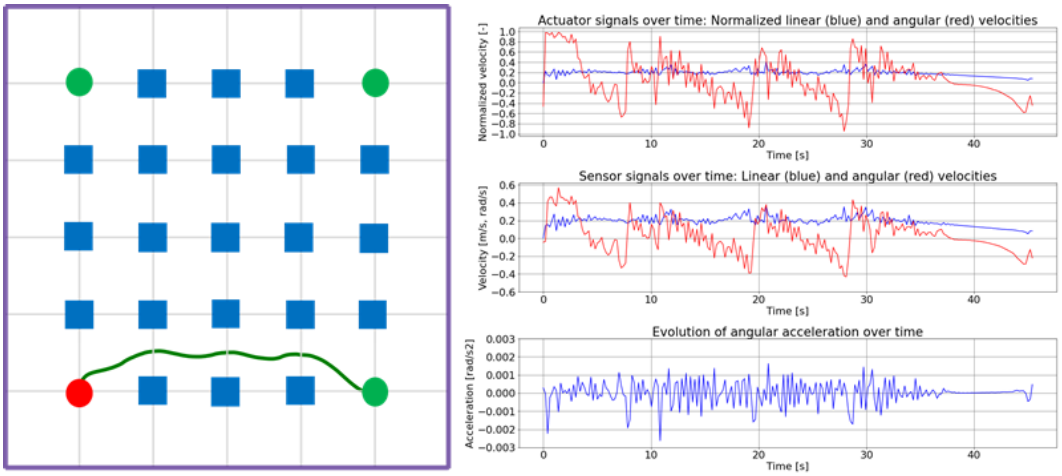


Figure 5.43: Path travelled. Room with obstacles 2, destination 3. Safety + legal + comfort + task-oriented req.

The generalization of the policy has been successfully accomplished and proved with the previous results. However, the commanded angular speed seems to be a little bit more aggressive than in the scenario used for the learning process. Unseen scenarios can lead to areas in the actor (policy) and critic neural networks which have not been adapted due to lack of data during the learning process. For example, the validation scenario used previously reports readings in many laser rays. On the other hand, the learning scenario has a few obstacles and only a few laser rays report a reading lower than the maximum range. To improve the quality of the generalization, two main points can be considered:

- Include the limitation of jerk values (derivative of acceleration) in the comfort requirements
- Consider not only one but different scenarios during the learning process
- Try to reduce the size and hidden layers of the neural networks used in the reinforcement learning algorithm (or prune the resulted neural networks)

Regarding the episodic reward, Figure 5.44 contains the episodic reward of the first (safety + task-oriented req.) and second (safety + legal + comfort + task-oriented req.) test cases. In both test cases, the episodic reward increases towards 0 after roundabout 200 learning episodes but there is still room for improvement, as the episodic reward is a little bit away from 0, which is the maximum possible episodic reward that can be obtained (i.e., the robot does not get any penalization along the way). Therefore, more learning episodes could have been considered to improve the maturity and quality of the policy.

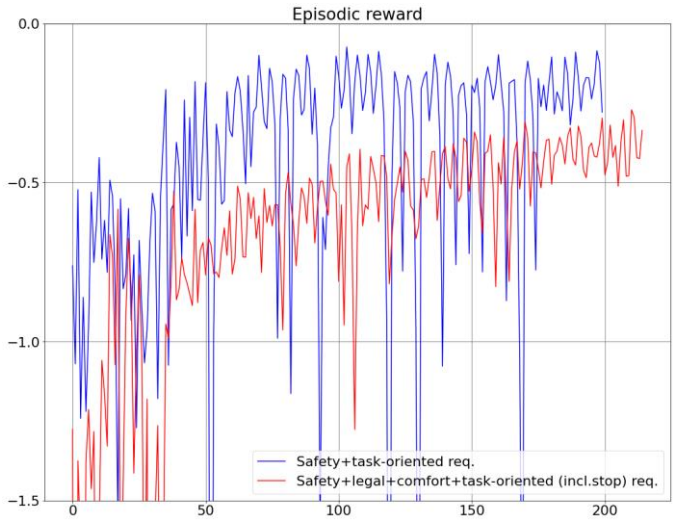


Figure 5.44: Episodic reward. Room with obstacles 1

The learning total time of these two experiments performed are summarized in Table 5.5:

Experiment	Total time
Room with obstacles 1. Safety + task-oriented req.	2.4 hours
Room with obstacles 1. Safety + legal + comfort + task-oriented req.	6.1 hours

Table 5.5: Learning total time room with obstacles

6. Conclusions

If we think of autonomous driving for passenger cars, many requirements need to be considered, making the development of software a challenging task. Not only safety requirements need to be considered (lane keeping or avoid static and dynamic obstacles) but also comfort, legal (maximum road speed limit) and task-oriented requirements are essential to be fulfilled. The state-of-the-art solutions produce thousands of possible trajectories to finally select the best one based on the score of a well-design cost function, such as the lattice planner introduced in [6]. However, many disadvantages come up with these state-of-the-art solutions. They are not only very computationally demanding, but they are also very challenging to implement to fulfil all predefined requirements. The new self-learning navigation algorithm introduced in this work comes up automatically with a policy and a cost function that fulfils all predefined requirements at a reasonable computational cost, since the algorithm proposed does not count with a motion planning module, like in the reactive paradigm. These requirements are defined by multiple and simple reward functions. Therefore, a high mature autonomous driving logic or policy is obtained just by trial and error. This has been proved by setting up some classical scenarios for the autonomous driving domain. Section 5.2.1 sets up a typical scenario for the validation of lane keeping algorithms. The reinforcement learning algorithm comes up automatically with a policy that perfectly follows the driving direction marked by means of walls (or with lanes painted on the ground in case of a front steering vehicle). Section 5.2.2 introduces not only the safety requirements from Section 5.2.1, but also legal, comfort and task-oriented requirements. Unlike some previous works like [24] that only include comfort and task-oriented (minimize travelling time) requirements, the methodology proposed also includes safety and legal requirements, apart from comfort and task-oriented requirements. Thus, the algorithm proposed can deal with continuous and multiple actions (linear and angular speeds) to fulfil the predefined requirements. On the other hand, section 5.2.3 introduces dynamic obstacles to the driving conditions. The state-of-the-art solutions like the lattice planner, based on the hierarchical/deliberative paradigm, struggle a lot under dynamic scenarios, making them difficult to apply under these conditions. The results achieved under section 5.2.3 prove that the reinforcement learning techniques are very powerful for complex scenarios with dynamic obstacles. Not only all the requirements are fulfilled, but also the policy is learnt automatically by trial and error. This learning process, which can be very time consuming, can be automated by simulation, loading different and complex driving scenarios. Additionally, the reinforcement learning algorithm produces a policy considering the dynamic constrains of the target vehicle or robot. This is because the agent interacts and learns directly from the environment where the robot belongs to, making it unnecessary to implement embedded models of the robot dynamics like in the model predictive control (such as [11], [12]). The observations and rewards delivered from the environment to the agent adapt the policy in a way so that the dynamic restrictions that the vehicle might have (limitation in radius of

curvature and its derivatives, friction tire-road, activation of dynamic control system under driving conditions close to the limit, etc.) are considered for every driving situation. Therefore, since the policy is generated automatically during the learning process, many engineering time and costs can be saved. Moreover, the neural networks produced by the algorithm are not very time consuming, making it even possible to run these already trained neural networks in a microcontroller, avoiding therefore high expensive computational platforms. Finally, section 5.2.4 proves the advantages of the proposed methodology under open scenarios of unknown dynamics, where the robot must learn from itself how to drive without having any autonomous driving logic programmed in advance. This is possible thanks to the online learning methodology, where the robot learns while it explores the surroundings. Besides, in case of online learning, the robot itself can recover from critical situations such as sensor aging or actuator malfunction. The reinforcement learning algorithm can modify the policy in an online way if the performance is influenced by sensor aging or if suddenly an actuator gets damaged, in case of a robot with redundant actuators. However, this case would need a high-performance computational platform since a microcontroller does not have enough resources.

7. Future work

There are 3 main branches of research and further investigation within the framework of this work. First, a more sophisticated and advanced setup for the learning process of the reinforcement learning algorithm can be created. The ideal setup would be the commissioning of a server with a dedicated GPU. The best configuration is to use multiple scenarios for the learning process, in contrast with the approach followed in this work, where a single scenario has been used for the learning process. Although the scenarios have been carefully designed to cover all the possible range of sensor measurements, it is challenging to accomplish this task with a single scenario. Second, a study considering hyper-parameter variations can be performed to find out the best set of parameters in terms of the robot performance. It is important to investigate different types of neural network architectures to select the simplest one which is still able to successfully learn the non-linearities of the problem. Additionally, it is interesting to adapt the different parameters of the reward functions depending on the goal to be accomplished. And finally, bring the logic into a microcontroller and connect it with real hardware. A differential robot can be used in a first step to prove the algorithm under real conditions. It is also interesting to investigate an efficient way to bring the neural network models into the microcontroller (prune neurons with low weights, implement fixed point variables to save memory or split the neural network into sequential functions to improve CPU load).

8. Bibliography

[1] C. W. Warren, "Multiple robot path coordination using artificial potential fields," in *Proceedings., IEEE International Conference on Robotics and Automation*, 1990.

[2] J. Álvarez-Sánchez, F. De la Paz López, J. Troncoso and D. Sierra, "Reactive navigation in real environments using partial center of area method," *Robotics and Autonomous Systems,* vol. 58, pp. 1231-1237, December 2010.

[3] S. E. Dreyfus, "An Appraisal of Some Shortest Path Algorithm," *Oper. Res.,* vol. 17, pp. 395-412, June 1969.

[4] B. Moses, L. Jain, R. Finn and S. Drake, "Multiple UAVs path planning algorithms: A comparative study," *Fuzzy Optimization and Decision Making,* vol. 7, pp. 257-267, April 2008.

[5] D. Dolgov, S. Thrun, M. Montemerlo and J. Diebel, "Practical Search Techniques in Path Planning for Autonomous Driving," *AAAI Workshop - Technical Report,* January 2008.

[6] A. Labbé and Y. Bhatt, "Autonomous vehicle navigation and parking," 2018. [Online]. Available: https://hdl.handle.net/20.500.12380/257268. [Accessed 01 08 2022].

[7] M. McNaughton, C. Urmson, J. M. Dolan and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *2011 IEEE International Conference on Robotics and Automation*, 2011.

[8] B. Paden, M. Čáp, S. Z. Yong, D. Yershov and E. Frazzoli, "A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles," *IEEE Transactions on Intelligent Vehicles,* vol. 1, pp. 33-55, 2016.

[9] G. C. Pereira, "Lateral model predictive control for autonomous heavy-duty vehicles: Sensor, Actuator, and Reference Uncertainties," Licentiate dissertation, KTH Royal Institute of Technology, Stockholm, Sweden, 2020.

[10] C. Zhou, B. Huang and P. Fränti, "A review of motion planning algorithms for intelligent robots," *J Intell Manuf 33,* p. 387–424, 2022.

[11] C. Liu, S. Lee, S. Varnhagen and H. E. Tseng, "Path planning for autonomous vehicles using model predictive control," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017.

[12] L. Svensson, M. Bujarbaruah, A. Karsolia, C. Berger and M. Törngren, *Traction Adaptive Motion Planning at the Limits of Handling,* 2020.

[13] I. Askari, B. Badnava, T. Woodruff, S. Zeng and H. Fang, "Sampling-Based Nonlinear MPC of Neural Network Dynamics with Application to Autonomous Vehicle Motion Planning," in *2022 American Control Conference (ACC)*, 2022.

[14] F. Hegedüs, T. Bécsi, S. Aradi and P. Gáspár, "Motion Planning for Highly Automated Road Vehicles with a Hybrid Approach Using Nonlinear Optimization and Artificial Neural Networks," *Strojniski Vestnik,* vol. 65, pp. 148-160, March 2019.

[15] F. Dellaert, "Machines That Sense, Think, And Act," Youngzine, [Online]. Available: https://youngzine.org/article-expert/technology/machines-sense-think-and-act. [Accessed 01 08 2022].

[16] R. S. Nair and P. Supriya, "Robotic Path Planning Using Recurrent Neural Networks," in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2020.

[17] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction (Second Edition), MIT Press, 2018.

[18] H. Dong, Z. Ding, S. Zhang, H. Yuan, H. Zhang, J. Zhang, Y. Huang, T. Yu, H. Zhang and R. Huang, Deep Reinforcement Learning: Fundamentals, Research, and Applications, H. Dong, Z. Ding and S. Zhang, Eds., Springer Nature, 2020.

[19] Microsoft Project Bonsai, "Writing Great Reward Functions - Bonsai," YouTube, 2017.

[20] Y. Hu, L. Yang and Y. Lou, "Path Planning with Q-Learning," *Journal of Physics: Conference Series,* vol. 1948, p. 012038, June 2021.

[21] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR,* September 2015.

[22] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley and A. Shah, "Learning to Drive in a Day," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019.

[23] M. Vitelli and A. Nayebi, "CARMA : A Deep Reinforcement Learning Approach to Autonomous Driving," 2016.

[24] P. Wang, C.-Y. Chan and A. de La Fortelle, "A Reinforcement Learning Based Approach for Automated Lane Change Maneuvers," 2018.