RAM ROBOTICS AND MECHATRONICS

Towards continuous control for mobile robot navigation: A reinforcement learning and SLAM based approach

K. (Khaled) Mustafa

MSc assignment

Committee:
dr.ir. T.J.A. de Vries
N. Botteghi, MSc
dr. B. Sirmaçek
dr.ir. M. Abayazid
dr. M. Poel

August 2019

036RaM2019
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE | DIGITAL SOCIETY INSTITUTE

Summary

Mobile robots have attracted intense research interest in recent years due to their numerous potential applications. The capability of autonomous navigation is critical when performing a wide range of tasks including search and rescue missions, ballast tanks maintenance, pipe inspection and activities that are potentially hazardous for humans. Therefore, it is such an exciting and varied topic in the robotics research community.
In literature, various traditional motion planner approaches do exist to solve the autonomous navigation problem for mobile robots. However, most of these approaches assume that a highly precise map of the navigation environment is provided a priori which is not the case, in real life applications, where the exact locations of the obstacles can be hardly obtained. Thus, the motivation for this work is to formulate the mobile robot navigation task in unknown environments as a reinforcement learning problem where (sub)optimal trajectories to desired targets can be realized through trial and error interaction with the environment. Henceforth, a model-free deep deterministic policy gradient approach within an off-policy actor-critic framework is sought that aims at training a motion planner end-to-end to navigate to any random target within the workspace. The motion planner is designed by taking 10-dimensional sparse laser range finder readings, the target position with respect to the robot's coordinate frame and the previously executed actions as inputs and the continuous linear and angular velocity commands as outputs where the robot has to rely on its on-board sensors to perform the navigation task.
The main novelty of this work is to shape the reward function based on the online-acquired knowledge about the environment that the robot gains during training. This knowledge is obtained through a grid mapping with Rao-Blackwellized particle filter approach in such a way that the robot can learn a (sub)optimal policy in less number of iteration steps by increasing its awareness about the locations of the surrounding obstacles. To the best of the author's knowledge, this is the first time grid mapping is combined with reinforcement learning to shape the reward function for robot navigation. Additionally, the learned planner can generalize to unseen virtual environments as well as to a real non-holonomic differential robot platform without any fine-tuning to real-world samples.
In order to validate the effectiveness of the proposed approach, two different virtual simulation environments are explored. The evaluation indicates that the proposed approach decreases the number of iteration steps significantly by 35.1% and 23.8% on the first and second environments respectively. Other performance evaluation metrics are also introduced that demonstrate that the proposed approach significantly outperforms the standard reinforcement learning approach. Furthermore, the generalization capabilities of the learned policy was tested. It is proven that the proposed algorithm substantially decreases the required learning time after the first task instance has been solved, which makes it easily adaptable to changing environments. Finally, it is validated that the learned planner can generalize to a real non-holonomic differential drive robot platform without any further training.

Contents

1 Introduction

As the robotics field progresses, robots are being employed in increasingly complicated and demanding tasks. To accomplish a given task, a robot receives sensory information representing its external environment and takes actions accordingly based on the collected data. The goal for mobile robots is to enhance their behaviour over time by empowering them with high autonomous ability based on their incoming experience about the environment. In case a complete knowledge about the environment is known a priori, a feasible path to a goal location in the environment can be determined using techniques such as potential field methods [1], graph search [2] and rapidly exploring random trees [3]. Although a lot of progress has been made by those path planning algorithms, they so often require human assistance during set-up time for sensory data acquisition [4]. Additionally, a highly precise map of the environment is required a priori for path planning approaches to work. For this reason, the challenge of autonomous navigation in environments with unknown terrain can be formulated as a reinforcement learning problem where the agent learns the optimal path through a straightforward trial and error process by interacting with the environment.

In recent years, reinforcement learning has achieved impressive results on a wide variety of challenging tasks like learning to play Atari video games directly from pixels [5] and learning policies for complex continuous control problems that involve locomotion and manipulation [6; 7], etc. In robotics, in particular, deep learning and deep reinforcement learning have similarly achieved impressive results. Thanks to it, robots are now capable of learning complex manipulation tasks like opening a bottle [8], putting cloths on a hanger [9], and precisely fitting small pieces into a larger structures [10]. Because reinforcement learning agents can learn without expert supervision, the type of problems that are best suited to reinforcement learning are complex problems where no obvious or easily programmable solution does appear. Reinforcement learning enables a robot to autonomously discover an optimal behavior through trial-and-error interactions with its environment. During this interaction, the robot perceives the environment through its sensors $\mathcal{S}$ and affects the environment through a set of actions $\mathcal{A}$ performed by its actuators. Instead of explicitly detailing the solution to a problem, in reinforcement learning, the designer of a control task provides feedback in terms of a scalar objective function that measures the one-step performance of the robot. In other words, by applying an action $a_t \in \mathcal{A}$, the agent is able to change its own state $s_t \in \mathcal{S}$ and the state of the environment and consequently it receives a reward $r_t \in \mathbb{R}$ for being in the new state. The reward is the way of teaching the agent whether the action taken in that state is good or bad. Accordingly, the optimal mapping from states to actions, called optimal policy $\pi^*(a|s)$, can be discovered by maximizing a predefined accumulated reward that reflects the quality of the trajectory taken by the robot.

In reinforcement learning context, the reward function is one of the most essential and effective parts since it is the only way by which the agent can evaluate its performance. Hence, a good reward often makes a difference between a tractable learning problem and an intractable one as will be discussed in section 5.2. For this reason, a novel idea to condition the reward signal on the knowledge of the environment is presented in this thesis.

The aim of this thesis is twofold. Firstly, it is aimed to design and apply a deep deterministic policy gradient learning based approach through an off-policy actor-critic framework. This approach is aimed at training a motion planner to accomplish autonomous navigation tasks where the planner can output continuous linear and angular velocities. Secondly, in an attempt to improve the learning rate, the reward function is shaped based on the knowledge gained about the environment that the robot acquires online during training to increase its awareness about the surrounding obstacles. The learned navigation policy is validated on both virtual and real-world environments

1.1 Related Work

In literature, lots of proposed learning approaches do exist that enable a robot to learn navigation actions using on-board sensory information in environments with known and unknown flat terrain. Benefiting from the improvement of high-performance computational hardware, these methods are mainly based on deep reinforcement learning. In [11], a successor feature deep Q-Network (DQN) based reinforcement learning is proposed to solve the navigation problem when a map of the environment is known a priori. The main focus is to transfer the knowledge from one environment to another where the input is depth images obtained through a kinetic sensor and the output is discrete actions including standing still, turning 90° right or left or going straight for 1m. The trained controller was transferred to a new environment on a physical mobile robot, and additional training was conducted. Experiments in the same environment showed that the number of additional training is less than having to train a new network from scratch. In [12], Asynchronous Advantage Actor-Critic (A3C) approach was proposed to help a robot moving out of a random maze for which a map is given. The input to the system includes 2D map of the environment, the robot's heading and the previous estimated pose whereas the output is the navigation actions such as move (forward, backward, left, right) and rotate (left, right) to a given target location. Furthermore, in [13], an external memory acting as an internal representation of the environment for the agent is fed as an input to a deep reinforcement learning algorithm. In this way, the agent is guided to make informative planning decisions to effectively explore new environments.
Although the aforementioned literature shows the feasibility of using deep reinforcement learning to learn navigation policies using high dimensional sensory inputs from the environment, they suffer from two main problems. The first one is that their navigation actions are simply discrete like move forward, turn left and turn right which may lead to rough navigation behaviors. Moreover, in all these methods, the map of the environment was provided to the robot a priori where the robot is trained to navigate through this map without human assistance. However, there are two issues regarding providing a map a priori for a training method. The first problem is regarding the time-consumption needed for building the obstacle map, whereas the second one is the high dependency of these approaches on the preciseness of the built map. Hence, in this thesis, it is aimed to design a learned motion planner that produces continuous control actions to navigate a mobile robot to a desired target. The contribution of this work is that no map is provided to the robot beforehand. However, instead, the robot builds a probabilistic map of the environment online during training and uses this map to shape its reward function. In this way, the robot gets more awareness about the surrounding obstacles which enables it to figure out a navigation policy in less iteration steps.

1.2 Problem Formulation and Challenges

Although deep reinforcement learning has achieved tremendous results where a key representative of this advancement is the application of reinforcement learning to the Go game [14] that was once considered the most challenging problem in the artificial intelligence community [15], robotics hold several unique challenges for learning algorithms compared to other fields. Accordingly, a naïve application of reinforcement learning techniques in robotics is likely to be doomed to failure. Hence, it is instructive to emphasize on some challenges faced in robotics learning. The following points give more insight regarding the main challenges that can be encountered while applying reinforcement learning algorithms in robotics.

(i) Many tasks of interest in robotics have continuous and high dimensional action spaces which limits the application of deep Q-Networks [1] to such problems since they rely on finding the action that maximizes the action-value function which would be computa-

[1] Discussed in detail in section 3.2

tionally exhaustive in continuous control tasks. A direct solution to this problem is to discretize the action space, however this has many limitations among them is the curse of dimensionality. In 1957, Bellman coined the term *Curse of Dimensionality* when he faced an exponential explosion of the number of actions with the number of degrees of freedom in discrete high dimensional spaces. For instance, a 7 degree-of-freedom robotic arm with a coarsest discretization $a_i \in \{-k, 0, k\}$ for each joint leads to an action space with dimensionality: $3^7 = 2187$. The problem becomes even worse when a finer discretization is required. This exponential growth in the number of states leads to very slow convergence rates of the reinforcement learning algorithm.
Hence, to tackle this problem, in this thesis, it is focused on the navigation problem of non-holonomic mobile robots with continuous control through deep reinforcement learning, which is an essential ability for the most widely used robots. This is achieved by adapting a policy search based learning approach that is discussed in section 3.4.

(ii) Another problem for learning in robotics is that it is often unrealistic to implement the training procedure in the real world. The reason is that the trial-and-error process may lead to a serious damage to real systems. Furthermore, obtaining real data from robotic systems can be extremely difficult and time-consuming. Thus, applying reinforcement learning in robotics demands safe exploration which becomes a key issue of the learning process, a problem often neglected in the general reinforcement learning community, due to the use of simulated environments. Additionally, the dynamics of the robot can change, during learning, due to several external factors including wear and tear of physical components and thereby the learning process may never converge.
Fortunately, simulation with accurate models could potentially be used to offset the cost of real-world interaction. One of the most powerful simulation platforms in robotics community is Gazebo in the sense that it is more realistic with respect to the underlying physics and the sensor noise compared to other simulated environments [16]. In an ideal setting, this approach would allow to learn the behavior in simulation and subsequently transfer it to the real robot.

(iii) The reward function can be considered the most important component within the reinforcement learning framework since it is the only way by which the agent can evaluate its performance while training. In contrast to traditional reinforcement learning algorithms, it is challenging to define a good reward function for the robot task since sparse reward can barely succeed in robotics applications. Therefore, it is necessary to design a proper reward function that takes into account the features of the space in which the agent robot operates and the available knowledge of the environment. Adjusting the reward function based on the task at hand is called, in literature, reward shaping which is considered as the main contribution of this thesis.

Based on the aforementioned reasons, in this thesis, it is aimed to train a motion planner to continuously navigate a mobile robot to any desired target within the environment by using only low-dimensional laser range findings. Accordingly, an actor-critic deep deterministic policy gradient algorithm is adopted for learning a navigation policy that can generate continuous control actions. Furthermore, the reward function is adjusted by incorporating the knowledge gained about the environment during training to improve the learning rate. This knowledge is provided in the form of a probabilistic occupancy grid map. To get this map, a *simultaneous localization and mapping*, also known as SLAM, based approach is introduced that makes use of Rao-Blackwellized particle filters. Moreover, the simulated experiments within the framework of this research are conducted on Gazebo simulation platform since simulations are frequently faster than real-time and safer for both the robot and its environment. Afterwards, the learned policy on the simulation is deployed on a real-world differential drive mobile robot to continuously control its navigation actions.

1.3 Proposed Method

As mentioned beforehand, the main purpose of this research is to design a deep reinforcement learning algorithm that uses raw sensory data from the robot's on-board sensors to determine a series of primitive navigation actions for the robot to execute in order to traverse to a goal location in an environment with unknown flat terrain. The advantage of deep reinforcement learning is that it can directly use raw sensory data to determine robot navigation actions without the need for pre-labeled data. In section 1.2, it was discussed that policy search methods are more suitable for physical control tasks. For that reason, a model-free off-policy[1] actor-critic algorithm based on deep deterministic policy gradient is presented within the framework of this thesis to solve the navigation problem for non-holonomic mobile robots. To achieve this goal, two neural networks are introduced where one of them works as the actor whereas the other is the critic. The aim of the actor network is to determine the optimal deterministic policy that maps the states, which are given as raw laser range findings data and the target's position in the robot's frame to navigation actions in the form of continuous linear and angular velocities. Here, it should be pointed out that the deep-RL algorithm represents the *high-level controller* of the robot. Once the robot's navigation actions are determined, the robot's low-level controller executes each action by sending the appropriate commands to the actuators. The critic network then evaluates the quality of taking a particular action from a certain state by computing an action-value function. Afterwards, the parameters of the actor network are then adjusted in the direction of the gradient of the action-value function which is discussed in section 3.4. This process continues iteratively until an optimal policy is realized.

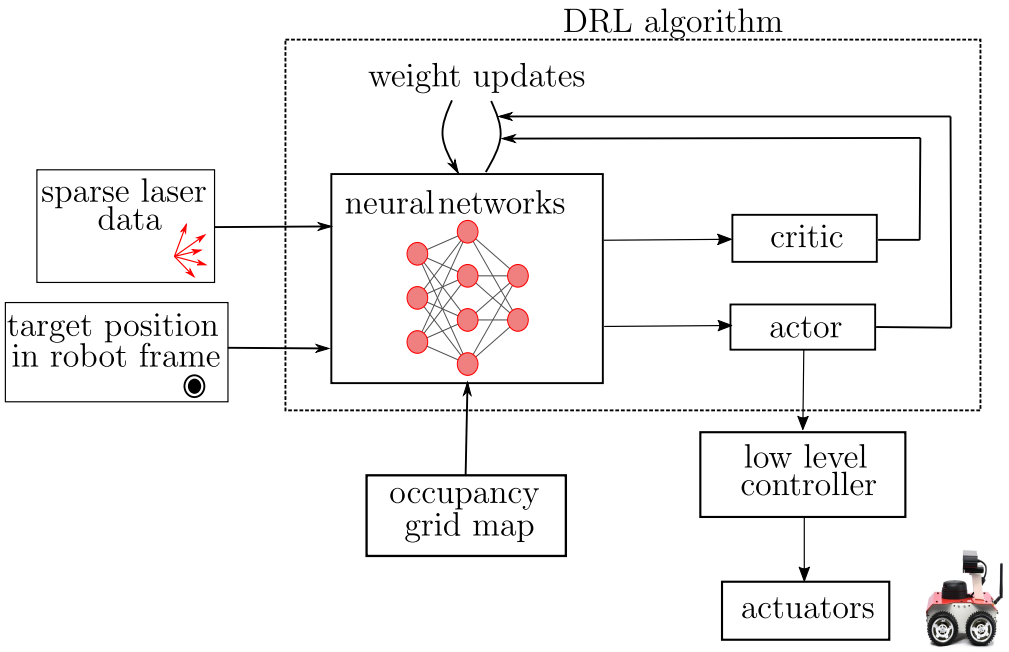


Figure 1.1: Proposed architecture for continuous control navigation in unknown environment. The motion planner is trained through a deep-RL within an off-policy actor-critic framework that represents the high level controller of the robot using only sparse laser data. The robot's low level controller executes the navigation actions determined by the motion planner. The occupancy grid map is built online by the robot and used to shape the reward function.

As a matter of fact, since the reward function has a direct impact on the learning rate of the reinforcement learning algorithm, the available knowledge of the environment is incorporated

[1] These concepts are discussed in chapter 2

in the reward function. This knowledge is represented by the online-acquired occupancy-grid map that the robot gets while learning through adopting a grid-mapping with Rao-Blackwellized particle filters. This SLAM technique is used due to its computational efficiency which is a key factor in deep reinforcement learning algorithms along with the fact that it can be simply integrated in robot operating system "ROS"; the main middleware framework on which the proposed algorithm is implemented. A comparison is then made to evaluate the difference in performance between the standard reinforcement learning technique and the one combined with SLAM. Here, it should be pointed out that SLAM is not the main concern of this research, however instead, it is just used as a tool to improve the learning algorithm. The architecture of the proposed framework is depicted in Figure 1.1 where the main modules are discussed in detail throughout this thesis.
Moreover, a major challenge of learning in continuous action space is exploration especially since the algorithm adopted learns a deterministic policy. However, an advantage of the off-policy algorithm is that the exploration problem can be treated independently from the problem of learning. In other words, it is possible to learn a deterministic policy while following a stochastic behavioral policy for exploration purposes. Therefore, different exploration policies are introduced including action and parameter space policies.

1.4 Research Questions

To fulfill the required tasks discussed in the previous section, the following two main research questions are formulated:
RQ1. How the navigation problem of non-holonomic mobile robots can be formulated as a reinforcement learning problem that could be solved by using deep deterministic policy gradient (DDPG) actor-critic algorithm?
RQ2. To what extent does the incorporation of the partial map obtained about the environment via the SLAM algorithm improve the learning algorithm?
From these main research questions, further sub-questions arise which can be articulated as follows:
RQ3. How to determine a proper reward function that can reflect the quality of the learned trajectory?
RQ4. How applicable is it to generalize the learned policy on one environment to another environment through transfer learning?
RQ5. What is the effect of different exploration noise on the quality of the learned trajectory and the learning rate?
RQ6. Is it possible to transfer the learned policy on the virtual environment directly to the real robot?

The rest of this thesis is dedicated to answer these research questions.

1.5 Report Layout

The remainder of this thesis is structured as follows. In **chapter** 2, a formal introduction to reinforcement learning problem is provided by explaining the mathematical preliminaries associated with it. This chapter also discusses traditional reinforcement learning approaches; model-free and model-based RL which forms the basis of most deep reinforcement learning methods. **Chapter** 3 introduces deep neural networks and explains its different components. The chapter also discusses the modifications that should be done to neural networks in order to be used as function approximators in learning algorithms. Additionally, few deep reinforcement learning algorithms are introduced and a motivation is provided for choosing a particular method for this thesis. A brief discussion of grid-mapping Rao-Blackwellized particle filter based SLAM approach is provided in **chapter** 4. Furthermore, **chapter** 5 focuses on the design

of the motion planner and how DDPG algorithm is applied to the autonomous mobile navigation in environments with unknown flat terrain problem. The reward shaping and the way in which the acquired map is incorporated in the reward function are discussed in this chapter as well. Moreover, a comparison between action and parameter space exploration noise is made. In **chapter** 6, the experimental setup settings along with the environments on which the learning algorithms take place to answer the research questions are described. In addition to that, a proposed difference model learning algorithm that captures the mismatch between the real system and the simulated model is explained. In **chapter** 7 the performance of the proposed algorithm on different experimental setups is evaluated and the results are discussed. Eventually, **chapter** 8 draws a conclusion about this research and provides possible recommendations for future work.

2 The Reinforcement Learning Problem

This chapter provides an overview of the reinforcement learning problem, presenting the mathematical background behind the solution of the decision making problem, and explaining the methods upon which this research is built.
In general, machine learning techniques can be typically classified into three broad categories: supervised learning, unsupervised learning and reinforcement learning. These three approaches differ by the type of feedback they receive to learn. In supervised learning, the learning agent is provided with a data-set of labeled examples, each contains a description of a situation as well as the correct classification of this situation or action to take when confronted by it. The objective of the learning is then to extrapolate this training data-set and be able to determine the correct classification or action to take for unseen situations. Unsupervised learning algorithms, in contrast to supervised learning, have no access to output values and therefore try to find hidden parameters and structures within the data by creating clusters that can group the given data. On the other hand, reinforcement learning is different from both previously described categories.
Reinforcement learning is a branch of machine learning that deals with sequential decision making. RL is a problem in which the agent, also called a decision maker, interacts with the environment. In the context of robotics, in this interaction, the agent senses the environment through its on-board sensors and responds to the environment through actions performed by its actuators. Based on these actions, the agent receives a scalar reward which is a way of letting the agent know how good or bad it was to take that action from this particular state. The fundamental idea of reinforcement learning is to learn an *optimal policy*, which is a mapping from states to a probability distribution over actions, that maximizes the expected sum of rewards in an attempt to achieve a desired goal. RL is different from supervised learning in the sense that in RL the agent does not know a priori what the right action is at the particular instant. However, instead, it must figure that out based on a trial and error interaction with the environment [17]. In episodic settings, where the task is restarted after each time the episode is over, the goal of the agent is to maximize the total reward per episode. However, if the task is on-going, there is no clear beginning and end, the aim is either to maximize the average reward of the whole life-time or a weighted average reward where the distant rewards have less influence.

The interaction model between the agent and the environment can be modelled as a Markov Decision Process (MDP) [18] which is described in detail in the next section.

2.1 Markov Decision Process

In the robotics context, the agent can be considered as the high level controller of the robot which is responsible for decision making [19], i.e. the required velocity that should be requested from the motors. The aim of reinforcement learning is to control this agent by finding an optimal policy by which the agent's can determine its optimal actions. The thing that the agent interacts with, comprising everything else outside the agent, is called the environment. This includes even the robot's sensors and actuators. The agent and the environment interact continually where the agent selects actions and the environment responds to these actions and presents new situation to the agent. This interaction process between the agent and the environment is modelled as a Markov decision process.
A Markov decision process is a discrete mathematical framework for *sequential decision making* which consists of five components that can be formally represented by a tuple [18]

$$\mathcal{M} = \left(\mathcal{S}, \mathcal{A}, \mathcal{P}(s_{t+1}|s_t, a_t), \mathcal{R}(s_t, a_t), \gamma\right) \tag{2.1}$$

where: $\mathcal{S}$ is a finite set of states which contains the information about the environment that is available to the agent at a discrete time-step t; an example would be the current position of a robot in a navigation task. $\mathcal{A} = \{a_1, ..., a_k\}$ is a set of actions the agent can perform on the environment; an example could be the linear and angular velocities of a robot. Not to mention that both the states and actions can be either discrete or continuous sets. The probability of ending up in state s_{t+1} when performing an action a_t in the state s_t is restricted to satisfy the Markov property given by

$$\mathcal{P}(s_{t+1}|s_t, a_t) = \mathcal{P}(s_{t+1}|s_1, a_1, ..., s_t, a_t) \tag{2.2}$$

In other words, the Markov property states that the transition from s_t to s_{t+1} depends exclusively on the previous state s_t and action a_t and not on additional information about the past states or actions. Moreover, the transition probability captures the dynamics of the environment. $\mathcal{R}(s_t, a_t)$ is the reward function that gives the agent a feedback from the environment. This feedback is given in terms of a scalar signal $\mathcal{R}(s_t, a_t) \in \mathbb{R}$ after the agent performs an action a_t from the state s_t and is assumed to be a function of the state. This gives a rise to a sequence $(s_0, a_0, r_1, s_1, a_1, r_2, ...)$ that is rolled out by the agent in the environment. As a matter of fact, the reward function specifies the goal of the reinforcement learning problem. The last component of the Markov Decision Process is the discount factor $\gamma \in [0, 1]$ which is used to determine how much the future rewards should influence the accumulated reward, also called as the return. If γ is close to 0, the return evaluation will be myopic and may result in poor performance, whereas, it will be "far-sighted" when γ is close to 1, i.e., the closer γ is to 1, the more effect future rewards would have on the return. Here it should be pointed out that in some reference, the discount factor γ is considered as a part of the MDP [19; 20; 21] while in other definitions it is regarded as an additional parameter where both conventions are widely used in literature. The Markov Decision Process in the context of robotics is depicted in Figure 2.1:

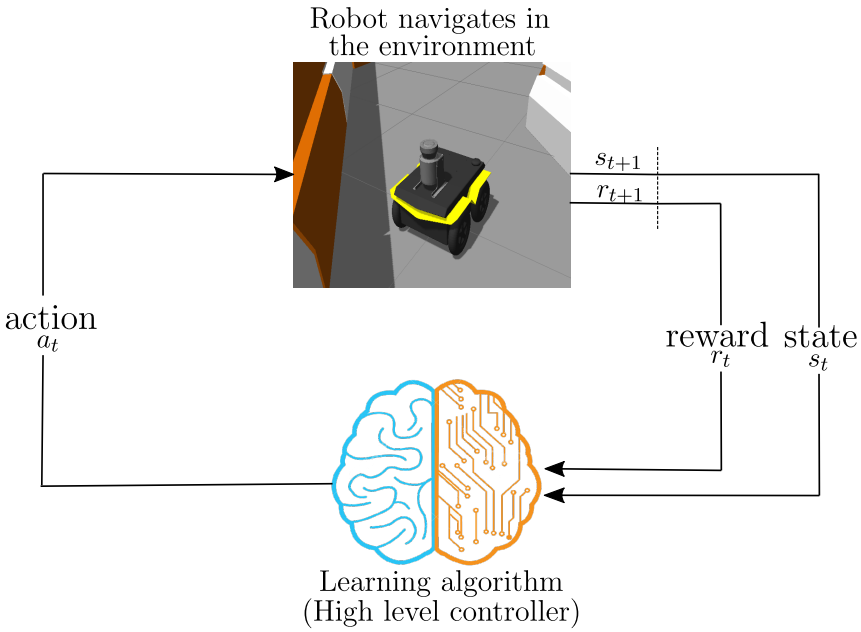


Figure 2.1: The reinforcement learning loop in the context of robotics. At time t, the agent receives a state s_t from the environment. The agent uses its policy to choose an action a_t. Once the action is executed, the environment transitions a step, providing the next state s_{t+1} as well as feedback in the form of a reward r_{t+1}. The agent uses knowledge of state transitions, of the form $(s_t, a_t, s_{t+1}, r_{t+1})$ in order to learn and improve its policy.

Figure 2.1 can be related to Figure 1.1 in the sense that the brain of the robot, the agent, can be resembled by the DRL algorithm depicted in Figure 1.1 while everything else represents the environment that the agent interacts with.

2.2 Reinforcement Learning Bricks

In this section, the fundamental background for reinforcement learning is discussed that is essential for understanding this thesis.

Definition 2.2.1. Let R_t be the immediate reward at time step t, and $\gamma \in [0,1]$ be the discount factor. The return G_t, is then defined as

$$G_t = \sum_{k=0}^{T} \gamma^t R_{t+k+1} \tag{2.3}$$

where T is the last time step in the interaction between the agent and the environment. The planning horizon can be finite, as the case in episodic tasks [19], or infinite, as in continuing tasks where $T = \infty$. $\gamma < 1$ prevents an infinite sum of rewards from being accumulated. The reward defined in 2.3 represents the reward over a single sample across the environment, and thus, no expectation is required at this stage.
In Markov decision process, the goal is to find a policy, denoted by π, that maximizes the cumulative return. The policy describes the agent's way of behaving in the environment since it determines the next action the agent should take from any given state. The policy can be either deterministic $\pi(s)$ or stochastic $\pi(a|s)$. A deterministic policy always returns the exact same action from a given state in the form of $a = \pi(s)$, whereas a stochastic policy models a conditional probability distribution over actions and then draws an action according to this distribution $a \sim \pi(a,s) = p(\mathcal{A} = a|\mathcal{S})$.

Definition 2.2.2. A deterministic policy is a function $\pi(s)$ that maps states into actions $\mathcal{S} \rightarrow \mathcal{A}$, whereas a stochastic policy $\pi(a,s) : \mathcal{S} \rightarrow p(\mathcal{A} = a|\mathcal{S})$ is a mapping from a state to a probability of taking a specific action.

Another important concept in RL is the state-value function $V^\pi(s)$. This function represents the assessment of how good it is for the agent to be in a given state in terms of how much future reward can be accumulated from this state [18]. The state-value function can be calculated by the expected amount of reward the agent can expect to get from state s when following policy π.

Definition 2.2.3. The state-value function $V^\pi(s)$ of a state s, under the policy π can be defined as:

$$V^\pi(s) = \mathbb{E}_\pi [G_t|s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|s_t = s \right], \forall s \in \mathcal{S} \tag{2.4}$$

where $\mathbb{E}_\pi [.]$ is the expectation operator for policy π that the agent follows..

The value function of state s_t can be expressed in terms of the immediate reward and a discounted value of the successor state s_{t+1}. This recursive relationship between $V^\pi(s_t)$ and $V^\pi(s_{t+1})$ is known as *Bellman's expectation equation* [22] that can be derived starting from the definition of the state-value function given in (2.4).

$$\begin{aligned}
V^\pi(s_t) &= \mathbb{E}[G_t|s_t = s] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|s_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + ...\right)|s_t = s\right] = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1}|s_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s\right] = \sum_a \pi(a|s)\left(R_{t+1} + \gamma \sum_{s_{t+1}} p(s_{t+1}|s,a)V^\pi(s_{t+1})\right)
\end{aligned}$$

The reason why an expectation does exist in the state-value function is due to the fact that the underlying policy is stochastic. Thus, it is required to average over all possible actions that can be taken from this state. Furthermore, since the transition from one state to another after taking a certain action is not deterministic as it is conditioned by the transition model imposed by the dynamics of the environment, it is also obliged to average over the state-value function of all successive states. Here it is worth mentioning that a state might have a high value despite having a low immediate reward, because it regularly leads to other states that yield high rewards.
Similarly, it is possible to define the value of taking an action a from a state s and following a policy π thereafter. This is called the action-value function or quality function and is denoted as $Q^{\pi}(s,a)$. The action-value function is closely related to the state-value function but it is also conditioned by the action a. In other words, instead of measuring the value of being in a particular state, it measures the quality of taking an action a from a state s. As will be shown later, the advantage of using action-value function over state-value function is that no model of the environment is needed to figure out the optimal policy which makes it suitable for model-free approaches.

Definition 2.2.4. The action-value function $Q^{\pi}(s,a)$ for a given state-action pair (s,a) under policy π is defined as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t | s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a\right] \tag{2.5}$$

where $\mathbb{E}_{\pi}[.]$ is the expectation operator for policy π that the agent follows.

As mentioned previously, the goal of an agent is to learn a policy π (find a sequence of actions through an MDP) that maximizes the expected return of a state $V^{\pi}(s)$. A policy π is said to be better than or equal to a policy π' if the expected return it generates is greater than or equal to that of π' for all states, such as:

$$\pi \geq \pi' \Longleftrightarrow V^{\pi}(s) \geq V^{\pi'}(s), \forall s \in \mathcal{S}$$

For all MDPs, there exists at least one policy that is better than or equal to all other policies. This is called the optimal policy π^*. Accordingly, the optimal state-value function is the function that corresponds to the optimal policy and can be defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \quad \forall s \in \mathcal{S} \tag{2.6}$$

Similarly, the optimal action-value function is defined as

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{2.7}$$

The Bellman equation for the optimal state-value function $V^*(s)$ results in the *Bellman optimality equation*. The interpretation of the Bellman optimality equation is that the value of a state evaluated for an optimal policy must be equal to the expected return when in state s and picking the best action from this state.

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q_{\pi_*}(s,a) \\ &= \max_{a \in A(s)} \mathbb{E}_{\pi_*}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a\right) = \max_{a \in A(s)} \left(R_{t+1} + \gamma \sum_{s_{t+1}} p(s_{t+1}|s_t,a) V(s_{t+1})\right) \end{aligned}$$

This means that for reinforcement learning methods that use a value function to find optimal policies, such an optimal policy can be derived from an optimal value function, by picking

the best action from each state, where best means the action that maximizes the value of the next state in each state. This also applies to the action-value function resulting in Bellman optimality equation for Q^*

$$Q^*(s,a) = \sum_{s_{t+1}} p(s_{t+1}|s,a)\left(R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})\right) \tag{2.8}$$

In case the transition probabilities and the reward functions are known, the Bellman optimality equation can be solved in an iterative fashion. This approach is known as Dynamic programming-based optimal control approach such as policy iteration or value iteration. The algorithms which assume these probabilities to be known or estimate them online are collectively known as *model-based* algorithms. But for most other algorithms, they assume the probabilities are not known and they estimate the policy and value functions by performing rollouts on the system. These methods are known as *model-free* algorithms. Monte Carlo, temporal difference and policy search methods are the most common model-free algorithms used. In most of practical scenarios, an explicit model of the environment is not given a priori and it also requires high computational time to build this model. Thus, in the robotics context, it is preferred to use model-free learning approaches [19]. For this reason, model free algorithms are discussed in the next section. The reader who is interested in model-based algorithms can refer to Appendix A where they are discussed briefly.

2.3 Model-free Methods

Model-free methods can be applied to any reinforcement learning problem since they do not require an explicit model of the environment. Model-free methods can be generally classified into two categories, mainly value function based approaches and policy search methods. In value function based approaches, the agent tries to learn a value function and infer an optimal policy from it. On the other hand, in policy search methods, the agent directly searches in the space of the policy parameters in an attempt to find an optimal policy. There is also a hybrid, *actor-critic* approach, which employs both value functions and policy search [19].
Model-free approaches can also be classified as being either on-policy or off-policy. On-policy methods use the same policy for both generating actions and updating the current policy. However, on the other side, off-policy methods use a different exploratory policy to generate actions as compared to the policy which is being updated. The following subsections look at various model-free algorithms used as well as both value function and policy search based methods.

2.3.1 Value function based Methods

2.3.1.1 Monte Carlo Methods

Unlike dynamic programming discussed in Appendix A, Monte Carlo methods do not assume a complete knowledge of the environment's dynamics. However, instead, they require sample sequences of states, actions, and rewards obtained through interactions with the environment [18]. Monte Carlo methods estimate action-value functions $Q_\pi(s,a)$ by averaging the returns observed after visiting these states in the previous episodes. Thus, in order to ensure that well-defined returns are available, learning is only possible in episodic tasks. Furthermore, the problem is non-stationary; since the return of a state upon taking an action depends on the sequence of actions taken in post-states and the selection of the actions is undergoing learning. To handle this issue, Monte Carlo methods adopt the same idea of general policy iteration discussed in Appendix A, however, they differ from dynamic programming in the sense that they are based on a sample of experienced sequences rather than on a complete distribution of all possible scenarios. The value functions and corresponding policies interact to obtain an optimal policy:

$$\pi_0 \xrightarrow{PE} Q^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{PE} Q^{\pi_1} \xrightarrow{I} \pi_2 ... \xrightarrow{I} \pi^* \xrightarrow{PE} Q^{\pi^*}$$

Alternating between evaluation and improvement happens on an episode-by-episode basis. The observed returns of an episode are used for policy evaluation, and then the policy is improved at all the states visited in the episode. These methods converge to the optimal policy Q^{π^*} as the number of visits to each action-state pair approaches infinity. The corresponding greedy policy for any action-value function Q, is the one that for each $s \in \mathcal{S}$, chooses an action deterministically with maximal action-value:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s,a)$$

There are two main Monte Carlo methods that differ in the way in which the average return is calculated:

1. *First Visit MC method*: estimates the action-value function as the average of the returns following the first visit to the state-action pair in every episode.

2. *Every Visit MC method*: estimates the action-value function as the average of the returns after every visit of the state-action pair in every episode.

To summarize, Monte Carlo methods differ from dynamic programming methods in two major ways. Firstly, they can be used for direct learning without a model of the environment. Secondly, they do not bootstrap, which means they do not update their value estimates based on the value estimates of the successive state. Though Monte Carlo methods are straightforward in their implementation, they require a large number of iterations for their convergence and suffer from a large variance in their value function estimation since they use the actual return from every visited state till the end of the episode where this return suffers from noise. For this reason, temporal difference learning is discussed in the next section.

2.3.1.2 Temporal Difference (TD) Learning

Temporal difference methods combine ideas from both dynamic programming and Monte Carlo methods [23]. Like dynamic programming methods, TD methods execute bootstrapping which means that they update their estimates partly based on previous estimates, however, they use samples as Monte Carlo methods. While Monte Carlo methods need to wait until the end of the episode to determine the increment in the value function, one step TD method waits only until the next time step to execute the updates. Accordingly, instead of using the total accumulated reward, TD methods calculate a temporal error, which is the difference between the new and old estimates of the value function, by considering the reward received at the current time step and use it to update the value function. This kind of update reduces the variance but increases the bias in the estimate of the value function since it doesn't use the actual return in updating the value function estimates. The update equation for the value function is given by:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha \left(\underbrace{r_{t+1} + \gamma Q(s_{t+1},a_{t+1})}_{\text{TD-target}} - Q(s_t,a_t) \right) \tag{2.9}$$

where $\alpha \in [0,1]$ is the step-size parameter that determines how much the Q-value is updated, r_{t+1} is the reward received at the current time step, s_{t+1} is the new state and s_t is the old state. The algorithm keep repeating until the loss function (TD error) reaches a small value. Two TD algorithms which have been widely used to solve reinforcement learning problems are SARSA (acronym for State- Action-Reward-State-Action) and Q-Learning.
SARSA is an on-policy temporal difference algorithm that tries to learn an action-value function instead of a state-value function. In SARSA, the temporal difference error is used for the update of the action-value function. SARSA algorithm is summarized below:

Algorithm 1 SARSA

Initialize $Q(s,a) \in \mathbb{R}$ randomly, $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$.
repeat
  Initialize s_1
  Select an action a_1 using a policy derived from $Q(s,a)$, (e.g. ϵ-greedy)
  for $t = 1 : T$ **do**
    Take action a_t, observe reward r_{t+1} and new state s_{t+1}.
    Choose next action a_{t+1} using policy derived from $Q(s,a)$, (e.g. ϵ-greedy)
    Update Q using

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right)$$

  end for
until terminated

It is clear that SARSA is an on-policy algorithm since the behavioral policy for exploring the environment is the same as the update policy. Hence, this method is not preferred in case a deterministic policy is used for updating the Q-values since in this case it is not guaranteed to explore the entire workspace sufficiently. Therefore, in [24], an off-policy temporal difference algorithm known as Q-learning is introduced. In Q-learning, the post-action a_{t+1} is selected by maximizing the Q-value of the next state $Q(s_{t+1},a_{t+1})$ instead of following the current policy. Thus, Q-learning belongs to the off-policy category. The Q-learning algorithm is summarized below:

Algorithm 2 Q-learning

Initialize $Q(s,a) \in \mathbb{R}$ randomly, $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$.
for all episode **do**
  Initialize s_1
  Choose action a_1 using policy $\pi(s)$ derived from $Q(s,a)$, (e.g., ϵ-greedy)
  repeat
    for each step t in an episode **do**
      Take action a_t, observe reward r_{t+1} and new state s_{t+1}.
      Choose next action a_{t+1} using policy $\pi(s_{t+1})$.
      Update Q using

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right)$$

    end for
  until terminated
end for

In contrast to dynamic programming where the solution methods sweep through the whole state space for each iteration, in Q-learning, each Q-value is only updated when the agent discovers the state. This means that some of the states might be updated more than the others. Thus, extracting a greedy policy could then lead to too much exploitation of the knowledge the agent has learned. To tackle this problem, a popular choice for a policy that can be used to encourage exploration during sampling is the ϵ-greedy policy. The parameter $\epsilon \in [0,1]$ specifies the probability of selecting a completely random action. In all other cases, the assumed best

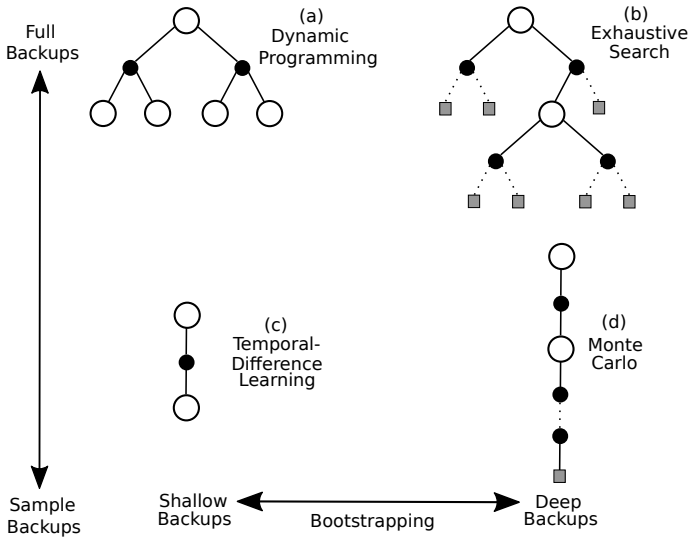


Figure 2.2: Two dimensions of RL algorithms, based on the backups used to learn or construct a policy. At the extremes of these dimensions are (a) dynamic programming, (b) exhaustive search, (c) one-step TD learning and (d) pure Monte Carlo approaches. Bootstrapping extends from (c) 1-step TD learning, with (d) pure Monte Carlo approaches not relying on bootstrapping at all. Another possible dimension of variation is choosing to (c, d) sample actions versus (a, b) taking the expectation over all choices. Recreated from [18].

action will be executed:

$$a_t = \begin{cases} \text{random action}, & \text{with } \epsilon\text{- probability} \\ \underset{a}{\text{argmax}}\, Q(s_t, a_t), & \text{otherwise} \end{cases} \tag{2.10}$$

This is used to let the agent discover states that might not be visited a lot of times in order to update the Q-value for these states, which could eventually lead to better policies. In most of the cases, it is preferred to start by a large number of ϵ to encourage the robot to explore most of the environment and then decrease this value by a decay rate to exploit the best obtained trajectories. This is known as *Exploration-Exploitation trade off*.
As a recap for value-function based approaches, Figure 2.2 depicts the main differences between dynamic programming, Monte Carlo methods and temporal difference learning.

2.3.1.3 Function Approximators

Function approximation is a family of mathematical and statistical techniques used to represent a function of interest when it is computationally intractable to represent the function exactly or explicitly. The easiest way to save the values of a value function for different states is in a tabular form. However, if the state space of the problem is large, it becomes impossible to store all the values in a tabular format. The reason is not only due to the fact that storing this data would require extremely huge amount of memory but also looking up some value for a particular state will require an entire sweep of the table which is computationally expensive. In addition, if the state space is continuous, a tabular format will become impossible. Here it is worth mentioning that the problem is not limited only to the large amount of memory required to store the table, but also to the large number of data and time required to estimate each state-action pair accurately. In other words, it is required to generalize the experience gained on a subset of state-action pairs to approximate a broader set.
To overcome this problem, function approximators are used to store a value function. Instead of a table, the value function can be parameterzied by a vector $\theta = [\theta_1, \theta_2, ..., \theta_n]^T$ that approximates the value function for a given input and is denoted as $V(s, \theta)$. The function approxima-

tor can be thought of as a mapping from a vector θ in $\mathbb{R}^n$ to the space of the value function. Nowadays, artificial neural networks (ANN) are widely used as function approximators. The advantage of using neural networks is due to their capability to represent complex value functions with lesser number of parameters. This reduces training time for reinforcement learning algorithms for high dimensional systems and is less memory extensive. However, despite the advantages of neural networks as function approximatros, applying them directly to reinforcement learning problem results in unstable performance. Thus, additional modifications are required to enable them to be applied effectively as discussed in section 3.2. Before describing the techniques in deep reinforcement learning, some theory of deep learning is needed which is introduced in section 3.1.

2.3.2 Policy Search Methods

In previous sections, it was shown that it is possible to derive reasonable performing policies from good estimates of value functions. However, because policies derived from value functions search over a discrete number of Q-values, it is not possible to directly obtain policies that output continuous actions using one of these methods described before. For this reason, policy search methods are analyzed in this section.
Policy search methods are another class of reinforcement learning algorithms that use parameterized policies π_{θ^π} that can be completely described by the parameter θ^π and thus provide maximal freedom to learn any action-generating function. To evaluate different policies, the expected return following π over all trajectories conditioned by the policy, formally $\tau \sim p_\pi(\tau) = p(\tau|\theta^\pi)$, is used where $p(\tau|\theta^\pi)$ is the probability distribution over sampled trajectories. The return over a single trajectory $r(\tau)$ is given by

$$R(\tau) = \sum_{t=1}^{T-1} \gamma^{t-1} r_{t+1} \tag{2.11}$$

The term r_{t+1} is the reward given to action a_t executed in state s_t of the respective trajectory. The probability distribution over trajectories $p(\tau|\theta)$ is decomposed as follows:

$$p(\tau|\theta) = p(s_1) \prod_{t=1}^{T-1} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2.12}$$

where $p(s_{t+1}|s_t, a_t)$ is given by the system dynamics of the robot and its environment. Since, the ultimate goal of the agent is to find an optimal policy π^*_θ that maximizes the expected accumulated reward that is defined as

$$\begin{aligned} J_\pi &= \int R(\tau)\, p(\tau|\theta) \mathrm{d}\tau = \mathbb{E}_\pi \left(R(\tau)|\pi\right) \\ &= \mathbb{E}_\pi \left(\sum_{t=1}^{T-1} \gamma^{t-1} r_{t+1} \right) \end{aligned} \tag{2.13}$$

This can be achieved by updating the parameters of the policy in the direction of increasing the expected return using gradient ascent. The update rule for the parameters of the policy can be written in terms of the expected return, J_π, as

$$\theta^\pi_{t+1} = \theta^\pi_t + \alpha \nabla_{\theta^\pi} J_\pi, \quad J_\pi = \mathbb{E}_\pi \left(\sum_{t=1}^{T-1} \gamma^{t-1} r_{t+1} \right) \tag{2.14}$$

According to [25], following the policy gradient to solve reinforcement learning tasks only slightly modifies the parameters of the policy in contrast to value based methods, where large jumps between two estimated policies are possible. This property arguably improves training stability and convergence towards an optimal policy. As shown in equation (2.14), in order to

update the parameters of the policy, it is required to derive an appropriate estimate of the gradient of the expected return $\nabla_{\theta^\pi} J_\pi$ with respect to the parameters of the policy. Analytically calculating the gradient is impossible, as it would be necessary to sum over possibly infinitely many trajectories. In addition, the dynamics of the environment $p(s_{t+1}|s_t, a_t)$ might also be unknown and not differentiable anyway. For this purpose a likelihood-ratio trick is introduced by [26] to evaluate the gradient of the expected return which is discussed in the next section.

2.3.2.1 Likelihood-Ratio Policy Gradient

Likelihood-ratio methods make use of the so called 'likelihood ratio trick' that is given by the identity $\nabla_{\theta^\pi} p(\tau|\theta) = p(\tau|\theta)\nabla_{\theta^\pi} \log p(\tau|\theta)$. This identity can be easily confirmed by using the chain rule to calculate the derivative of $\log p(\tau|\theta)$ which is given by

$$\nabla_{\theta^\pi} \log p(\tau|\theta) = \frac{\nabla_{\theta^\pi} p(\tau|\theta)}{p(\tau|\theta)} \tag{2.15}$$

Thus, the gradient of the expected return defined in equation (2.13) can be written as

$$\nabla_{\theta^\pi} J_\pi = \int R(\tau) \nabla_{\theta^\pi} p(\tau|\theta) \mathrm{d}\tau = \int R(\tau) p(\tau|\theta) \nabla_{\theta^\pi} \log p(\tau|\theta) \mathrm{d}\tau \tag{2.16}$$

$$= \mathbb{E}_{p(\tau|\theta)} \left(R(\tau) \nabla_{\theta^\pi} \log p(\tau|\theta) \right) \tag{2.17}$$

The expectation in equation (2.17) is useful for estimating the gradient of J_π while avoiding integrating over all trajectories which is intractable. However, the inner term $R(\tau)\nabla_{\theta^\pi} \log p(\tau|\theta)$ still depends on the possibly unknown or not differentiable system's dynamics. This problem can be solved by making use of equation (2.12)

$$\nabla^\pi_\theta \log p(\tau|\theta) = \nabla_{\theta^\pi} \log p(s_1) + \sum_{t=1}^{T-1} \nabla_{\theta^\pi} \log \pi_\theta(a_t|s_t) + \sum_{t=1}^{T-1} \nabla_{\theta^\pi} \log p(s_{t+1}|s_t, a_t) \tag{2.18}$$

$$= \sum_{t=1}^{T-1} \nabla_{\theta^\pi} \log \pi_\theta(a_t|s_t) \tag{2.19}$$

As shown in equation (2.19), the system dynamics can be excluded since they do not depend on the parameter θ^π. This means that all the knowledge about the dynamics of the environment can be easily discarded to form a model-free estimate of the parameter gradient. Thus, finally, the gradient of the expected return can be formulated as

$$\nabla_{\theta^\pi} J_\pi = \mathbb{E}_{\tau \sim p(\tau|\theta)} \left(R(\tau) \sum_{t=1}^{T-1} \nabla_{\theta^\pi} \log \pi_\theta(a_t|s_t) \right) \tag{2.20}$$

2.3.2.2 Exploration Strategies in Policy Search Methods

The exploration strategy is essential for efficient model-free policy search, since variability in the generated trajectories to determine the policy update is required. The main purpose of exploration strategies is to ensure that the agent's behavior does not converge prematurely to a local optimum. Most model-free policy search methods use a stochastic policy for exploration. Exploration strategies can be categorized into step-based and episode-based exploration strategies. While step-based exploration uses an exploratory action in each time step, episode-based exploration directly changes the parameter vector θ^π of the policy only at the start of the episode. Step-based exploration strategies can be problematic as they might produce action sequences which are not reproducible by the noise-free control law, and, hence, might affect the quality of the policy updates. Additionally, a step-based exploration strategy

causes a large parameter variance which grows with the number of time steps. Such exploration strategies may even damage the robot as random exploration in every time step leads to large jumps in the controls of the robot.
The exploration strategy is used to generate new trajectory samples τ^i which are subsequently evaluated by policy evaluation strategies and used for policy update. Thus, an efficient exploration strategy is crucial for improving the performance of policy search algorithms. The exploration strategy can also be categorized based on whether the exploration noise is applied in the action space or the parameter space. Exploration in action space is implemented by adding an exploration noise ϵ_u directly to the executed actions; $a_t = \mu(x,t) + \epsilon_u$ where the exploration noise is sampled independently for each time step. As a matter of fact, the action noise can be either uncorrelated as Gaussian noise $\epsilon_u \sim \mathcal{N}(0, \sigma^2 I)$ or correlated as the Ornstein-Uhlenbeck process $\epsilon_u \sim \text{OU}(0, \sigma^2)$. On the other hand, parameter space noise injects randomness directly into the parameters θ of the policy, altering the types of decisions it makes such that they always fully depend on what the agent currently senses. This exploration noise can either only be added at the beginning of an episode, or, a different perturbation of the parameter vector can be used at each time step.

2.4 Summary

This chapter introduced the basics of reinforcement learning by explaining many important concepts associated with it that will be used throughout this thesis. Various popular approaches within traditional RL were also discussed. In general, model-free methods are preferred for simulated environments since they do not require any information about the environment. Furthermore, for robotics applications, policy search methods are preferred over value function methods since they are suitable for continuous state and action spaces.

3 Deep Reinforcement Learning

In this chapter, deep learning and reinforcement learning are combined into what is called Deep Reinforcement Learning (DRL).The value-based methods discussed previously can be represented in a tabular format which is only feasible for state and action spaces with a limited number of states and actions. In cases where both $\mathcal{S}$ and $\mathcal{A}$ are big sets, these methods are infeasible not only due to memory requirements for storing the big value function tables, but also due to the data needed to fill out these tables accurately and the time needed for acquiring that amount of data. Instead of a table representation of the value function, one can use a function that approximates the desired value function. This function approximator is then trained using interactions between agent and environment. Deep reinforcement learning has received a lot of interest among the AI community in the last couple of years where it refers to the use of deep neural networks as function approximators for value functions or policy in an RL framework [19]. The fact that deep reinforcement learning can handle high dimensional state and action space makes it extremely suitable for the purpose of controlling the motion of a mobile robot, the problem under study.

3.1 Artificial Neural Networks

In order to understand neural networks, one first needs to be familiar with the computational units they consist of which are artificial neurons. An artificial neuron is a simple mathematical model that mimics the way biological neurons in the human brain process information [27]. A neuron is a computational unit that takes as an input a number of n signals, $x_0, x_1, ..., x_n$ and combine them together into a scalar output, thus it can be thought of as a mapping function. Each of the input signals are multiplied by their own weights $w_0, w_1, ..., w_n$ that determine how much the individual input signal affects the output of the neuron. These weighted inputs are summed together and a bias b is added before it passes through an activation function, ϕ that has the effect of applying non-linearity to v. Formally, the output is expressed by the two equations:

$$v = \sum_{j=0}^{n} w_j x_j + b$$
$$y = \phi(v) = h(x_0, x_1, ..., x_n) \tag{3.1}$$

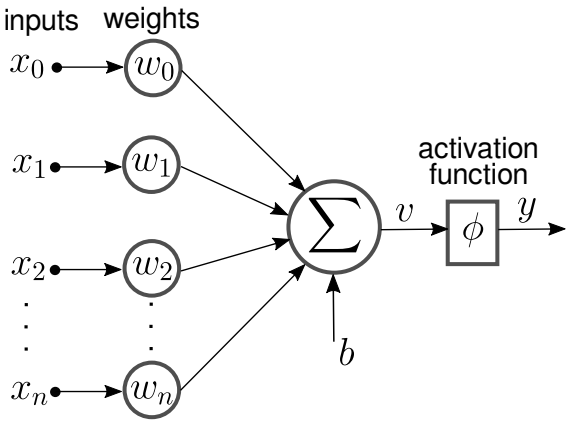


Figure 3.1: Illustration of how an artificial neuron transforms its inputs $x_0, x_1, ..., x_n$ to its output y.

There are different activation functions that can be used, where the sigmoid function, hyperbolic tangent (tanh) function, and rectified linear unit function, often called ReLU, are the most popular choices. Activation functions are sometimes also called squashing functions as they limit the output values of the neuron [27]. The sigmoid function, given in equation (3.2), is an example of a squashing function that limits the output of the neuron in the range of $y = [0, 1]$. The ReLU function, expressed in equation (3.3), does not set an upper bound of the output and does not allow negative outputs as

well.

$$\phi(v) = \frac{1}{1+e^{-v}} \tag{3.2}$$

$$\phi(v) = \max(0, v) \tag{3.3}$$

An important property of activation functions is that they introduce non-linearity to their inputs allowing the network to learn any arbitrary functional mappings. In addition, activation functions are differentiable which allow for gradient-based learning methods. Neurons are used to construct networks of connected neurons also known as neural networks. The neurons are organized in different layers and these layers are then stacked to build larger networks. There are different types of neural networks for which fully-connected and convolutional neural network are two common types. However, since throughout this thesis, only fully-connected neural networks are discussed, convolutional neural networks are discarded.

3.1.1 Feed-Forward Neural Networks

Feed forward neural networks, also called multilayer perceptrons, are the typical deep learning models. A feed forward network is a function that maps an input x to an output y using parameters θ, such as $y = f(x;\theta)$. The architecture of a feed-forward neural network usually has three kinds of layers: an input layer, a few hidden layers and an output layer. The information flows through the network from the input layer to the output layer which computes the final output through the hidden layers. Unlike recurrent neural networks, there are no feedback connections in which outputs of the model are fed back into itself. A typical fully connected neural network is shown in Figure 3.2.

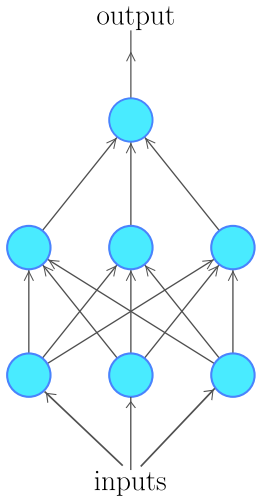
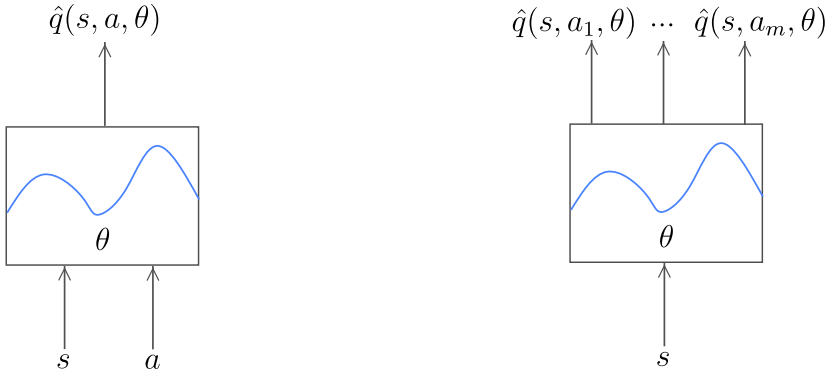


Figure 3.2: A fully connected feed-forward neural network with two hidden layers. The information flows from bottom to top.

Training algorithms for deep neural networks are usually iterative, therefore an initial point should be specified to start the training. This starting point affects the number of iterations required for the learning process to be done and the ability of the network to generalize at the end of the training. To update the parameters θ_i of the network, a loss function $L(\theta)$ is defined that represents the error between the desired output y^* of the input x and the actual output y obtained by performing a forward path through the network using the current parameters. The gradient of the loss function $\nabla_\theta L(\theta)$ is computed and the parameters are readjusted in the opposite direction of the gradient by propagating backward in the network. This algorithm is refereed to as *backpropagation* algorithm that applies recursively the chain rule to compute the gradients, starting from the output layer all the way back to the input layer. In the recent years, stochastic gradient descent has been a popular choice for training the weights of a neural network. A few improved variants of stochastic gradient descent has also been proposed, like ADAM [28]. The advantage of using ADAM method over standard stochastic gradient descent is that it can vary the learning rates based on the distribution of the training data. This reduces the need for careful choice of a learning rate which in turn allows the algorithm to converge faster.

The deployment of neural networks as a function approximator in a value-function based reinforcement learning problem can be done in two ways as shown in Figure 3.3; Firstly, it is possible to pass the current state and the intended action to the network and then the network

evaluates the quality of taking this certain action from that state in the form of an action value function. The second way is to pass only the state as the input to the network and then evaluate the quality of taking any of the all feasible actions from this state. In this case, the output of the network is composed of as many neurons as the number of actions. Based on the estimated action value-functions, the algorithm selects one of the actions based on the deployed policy, i.e, in case of an ϵ-greedy policy, the algorithm is going to select the action with highest value function with a probability of $(1-\epsilon)\%$. Here it is worth mentioning that the second way is only possible in case there is a discrete number of actions since the Q-value is estimated for every action.



(a) The action and state are used as inputs to the network and the quality of taking this action from that state is determined in form of a Q-value.

(b) The state represents the input of the network and the Q-values are estimated for all feasible actions from this state.

Figure 3.3: A graphical representation of neural networks as function approximators in reinforcement learning context [29].

As a matter of fact, neural networks can also serve as a function approximator for a parametric policy in case of a policy search based reinforcement learning as will be discussed later in this chapter where the input of the network is the current state of the robot and the output is the mapped action.
In the next section, it is shown how to use neural networks as function approximators for Q-values in reinforcement learning context.

3.2 Deep Q-Networks

Q-Learning, discussed in section 2.3.1.2, has been a widely used algorithm for model-free reinforcement learning. However, it was shown that utilizing neural networks as function approximators, to approximate the optimal action value function $Q(s,a;\theta^Q) \approx Q^*(s,a)$, directly to Q-learning algorithms without further modifications leads to an unstable behavior and the convergence is no longer guaranteed [30] and thus, most applications of Q-learning were limited to tasks with small state spaces. The main cause of this issue is that when using neural networks for reinforcement learning, it is assumed that the samples are independently distributed. However, this is not the case when the samples are generated sequentially since they are temporally correlated which results in high variance in the estimation. To tackle this problem an *experience replay* is introduced to break the temporal correlation between the consecutive transitions where the agent's experience at each time step $e_t = \langle s_t, a_t, r_t, s_{t+1} \rangle$ is stored in a replay buffer $\mathcal{D} = \{e_1, e_2, ..., e_N\}$ with a finite size N. At each time step, a fixed number of samples, a mini-batch, is extracted randomly from the replay buffer and used to train the network. When the replay buffer is full, the oldest samples were discarded meaning that the replay buffer $\mathcal{D}$ does not have to start with e_1 since it is shifted every time a new sample is added after getting full. This way, gradient descent methods from the supervised learning literature can be safely

used to minimize the TD-error squared. The learning of the value-function in deep reinforcement learning is based on the adjustment of the neural network weights by minimizing the loss function, which corresponds to the mean squared error between the TD target and the current value function

$$L_i\left(\theta_i^Q\right) = \mathbb{E}_{s\sim\rho_{\pi(.)},a\sim\pi(.)}\left[\left(\underbrace{Q\left(s_t,a_t;\theta_i^Q\right)-y_i}_{\text{TD error}}\right)^2\right] \tag{3.4}$$

where $y_i = r(s_t,a_t)+\gamma\max_{a_{t+1}} Q\left(s_{t+1},a_{t+1};\theta_i^Q\right)$ is the target at iteration i, $\pi(a|s)$ is the behaviour policy and $\rho_{\pi(.)}$ is the distribution of states under policy $\pi(a|s)$. To minimize the loss function, the gradient of the loss function is computed with respect to the weights and is given by

$$\nabla_{\theta_i^Q} L\left(\theta_i^Q\right) = \mathbb{E}_{s\sim\rho_{\pi(.)},a\sim\pi(.)}\left[\left(Q\left(s,a;\theta_i^Q\right)-r(s_t,a_t)-\gamma\max_{a_{t+1}} Q\left(s_{t+1},a_{t+1};\theta^Q\right)\right)\nabla_{\theta_i^Q} Q\left(s_t,a_t;\theta_i^Q\right)\right] \tag{3.5}$$

The parameters θ are updated using the stochastic gradient descent of the loss function such that

$$\theta_{i+1} = \theta_i - \alpha\nabla_{\theta_i^Q} L\left(\theta_i^Q\right) \tag{3.6}$$

As a matter of fact, DQN is is an off-policy model-free algorithm where the agent learns the Q-value function, while following a different behavior policy that provides sufficient exploration of the domain space. In practice, the behaviour policy is generally selected by an ϵ-greedy strategy. In [5], it is shown that implementing equation (3.4) directly results in divergence in many cases. The reason is that the updated $Q(s,a|\theta^Q)$ is also used with the same weights in calculating the TD target y_t which makes the optimization appears to be chasing its own tail resulting in instability. One possible solution is to introduce a second network called *target neural network* that is proposed in [31] to calculate target Q-values $Q'(s,a|\theta^{Q'})$ where the target network parameters θ' are only updated with the Q-network parameters θ every certain number of steps. The target-network and the Q-network share the same network architecture, but only the weights of the Q-network are learned and updated. A graphical representation of the DQN approach is shown in Figure 3.4

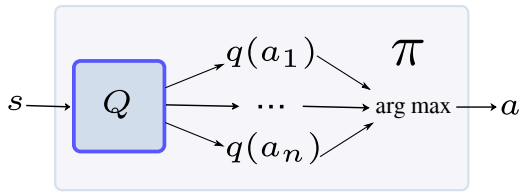


Figure 3.4: Architecture of DQN where the Q-Network outputs an estimate of the action-value function for each action. Subsequently, the policy π chooses the action with the highest value.

The complete algorithm of deep Q-Network (DQN) is given below:

Algorithm 3 Deep Q-Network

Initialize action value function $Q(s,a;\theta^Q)$ with random weights.
Initialize a replay buffer $\mathcal{D}$ with size N.
Initialize target action value function $Q'(s,a;\theta^{Q'})$ with $\theta^{Q'}=\theta^Q$.
for all episode **do**
  Initialize s_1
  repeat
    for each step t in an episode **do**
      Choose an action $a_t \in \mathcal{A}$ using ϵ-greedy strategy.
      Execute action a_t, observe reward r_{t+1} and new state s_{t+1}.
      Store transition $\langle s_t, a_t, r_t, s_{t+1}\rangle$ in replay buffer $\mathcal{D}$.
      Sample random mini-batch of transitions $\langle s_i, a_i, r_i, s_{i+1}\rangle$ from $\mathcal{D}$.
      if s_{i+1} is terminal **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a_{t+1}} Q'(s_{t+1}, a_{t+1};\theta^{Q'})$
      end if
      Train the Q-network on $(y_i - Q(s_i,a_i;\theta^Q))^2$ using (3.5).
    end for
  until terminated
end for

The one major drawback of the above algorithm is the need to calculate the maximum over actions and this prohibits the use of of the above algorithm for tasks with continuous actions spaces. To deal with continuous action spaces, an actor critic algorithm was developed which uses the Q-function as the critic and updates the policy using the Deterministic Policy Gradient discussed in the following sections.

3.3 Actor-Critic Algorithms

It is possible to combine value functions with an explicit representation of the policy, resulting in *actor-critic* methods. Actor-critic methods are TD methods which store the policy explicitly. The policy is known as the actor since it predicts the action in a given state. The value function acts as the critic since it evaluates the policy based on the temporal difference error. The policy is updated based on this critic. The actor critic method is mostly on-policy, but off-policy actor critic have been introduced in the literature [32]. Actor critic algorithms can either store the actor and critic in a tabular form or can use function approximators, which is the case with most robotic applications. When using function approximators, the policy is updated similar to policy search methods, except that the critic decides the direction of gradient ascent instead of the expected return. The stochastic policy gradient theorem [33] defines the gradient of the expected return, $\nabla_{\theta^\pi} J(\pi_\theta)$, using the likelihood-ratio trick explained beforehand in section 2.3.2.1, as

$$\begin{aligned}\nabla_{\theta^\pi} J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_{\theta^\pi} \pi_\theta\left(a|s;\theta^\pi\right) Q^\pi(s,a;\theta^Q)\mathrm{d}a\mathrm{d}s \\ &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi_\theta\left(a|s;\theta^\pi\right) \nabla_{\theta^\pi} \log\pi_\theta\left(a|s;\theta^\pi\right) Q^\pi(s,a;\theta^Q)\mathrm{d}a\mathrm{d}s \\ &= E_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log\pi_\theta\left(a|s;\theta^\pi\right) Q^\pi(s,a;\theta^Q)\right]\end{aligned} \tag{3.7}$$

where $\rho^\pi(s)$ is the state distribution under policy π, θ^π is the parameter vector for the policy π and θ^Q is the parameter vector for Q-function. The parameters of the action value function, $Q(s,a;\theta^Q)$, are updated using temporal difference learning. The types of algorithms that use

the definition of gradient given in (3.7) are known as stochastic actor-critic algorithms.
Equation (3.7) shows that the gradient is an expectation of both states and actions. Therefore, a large number of samples from both action and state space is required, in principle, in order to evaluate a good estimate of the gradient. However, by utilizing a deterministic policy instead, the mapping from state space to action space becomes fixed and accordingly there is no need to integrate over the whole action space and the thus the objective function can be rewritten as

$$J(\pi_\theta) = \int_{\mathcal{S}} \rho^\pi(s) Q(s, \pi(s;\theta^\pi)) \mathrm{d}s \tag{3.8}$$

where $\pi(s;\theta^\pi) = a$ represents the deterministic policy. In [34], a deterministic policy gradient algorithm that defines the gradient of the expected return subjected to deterministic policy is introduced. This is achieved by applying the chain rule to the gradient of the action-value function in 3.8 which results in its decomposition into its gradient with respect to actions, and the gradient of the policy with respect to the policy parameters

$$\begin{aligned} \nabla_{\theta^\pi} J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \nabla_{\theta^\pi} \pi\left(s;\theta^\pi\right) \nabla_a Q\left(s, a;\theta^Q\right) \mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho_\pi}\left[\nabla_{\theta^\pi} \pi\left(s;\theta^\pi\right) \nabla_a Q\left(s, a;\theta^Q\right)|_{a=\pi(s;\theta^\pi)}\right] \end{aligned} \tag{3.9}$$

In this way, the critic network uses Q-learning in updating its parameters in the direction of the gradient of the loss function with respect to the critic network's parameters

$$\begin{aligned} \delta_t &= r_t + \gamma Q(s_{t+1}, \pi(s_{t+1};\theta^\pi);\theta^{Q'}) - Q(s_t, a_t;\theta^Q) \\ \theta^Q_{t+1} &= \theta^Q_t + \alpha_{\theta^Q} \delta_t \nabla_{\theta^Q} Q(s_t, a_t;\theta^Q) \end{aligned} \tag{3.10}$$

where δ_t is the temporal difference error and α_{θ^Q} is the learning rate for the critic network. On the other hand, the actor network updates its parameters in the direction of the gradient of the Q-value with respect to the policy parameters

$$\theta^\pi_{t+1} = \theta^\pi_t + \alpha_{\theta^\pi} \nabla_{\theta^\pi} \pi(s_t|\theta^\pi) \nabla_a Q(s_t, a_t|\theta^Q)|_{a=\pi(s)} \tag{3.11}$$

where α_{θ^π} is the learning rate for the actor network.
The deterministic policy gradients can be computed more efficiently than the stochastic case and these algorithms show significantly better performance than their stochastic counterpart [34] since they require fewer data samples to converge. A variant of the off-policy algorithm which uses neural networks as function approximators for the actor and critic, known as Deep Deterministic Policy Gradient (DDPG), the framework of this study, is discussed in the next section.

3.4 Deep Deterministic Policy Gradient

Deep deterministic policy gradient is an actor-critic algorithm that provides an improvement to DQN discussed before by making it more tractable to continuous actions where both the actor and critic functions can be approximated by two separate neural networks with parameter vectors θ^π and θ^Q respectively. As the name suggests, the algorithm updates the parameters of the actor network using deterministic policy gradient theorem [34]. Each training step modifies the policy in such a way that its outputs are pushed in the direction of the positive gradient of the action-value function. Especially for continuous actions, this strategy is very effective, as it directly pushes the generated actions towards the assumed best action with respect to the action-value estimations.

$$\nabla_{\theta^\pi} J(\pi_\theta) = \mathbb{E}_{s\sim\rho_\pi}\left[\nabla_{\theta^\pi} \pi\left(s|\theta^\pi\right) \nabla_a Q\left(s, a|\theta^Q\right)|_{a=\pi(s|\theta^\pi)}\right] \tag{3.12}$$

Like DQN, it was observed that directly updating the parameters of the actor and critic networks using temporal difference as in equation (3.5), leads to divergence of the learning algorithm. Thus, same concepts of target networks with parameter vectors $\theta^{\pi'}$ and $\theta^{Q'}$ for both

the actor and the critic respectively and experience replay are used in DDPG algorithms as well. However, in contrast to DQN, the target networks are updated after each gradient step to slowly replicate the changes made to the trained networks.

$$\begin{aligned} \theta^{Q'} &= \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\pi'} &= \tau\theta^{\pi} + (1-\tau)\theta^{\pi'} \end{aligned} \tag{3.13}$$

This improves the stability of the learning algorithm [6]. The parameter τ determines how quickly $\theta^{Q'}$ and $\theta^{\pi'}$ track θ^{Q} and θ^{π}. Values of τ close to one result in fast yet unstable learning, whereas small values of τ result in slow yet stable learning. To encourage exploration, a stochastic policy is still used as behavioural policy to generate the training samples, which yields an off-policy training algorithm.
The complete DDPG algorithm is shown below

Algorithm 4 Deep Deterministic Policy Gradient

Initialize a replay buffer $\mathcal{D}$ with size N.
Initialize critic network $Q(s,a;\theta^{Q})$ and actor network $\pi(s,a;\theta^{\pi})$ with random weights θ^{Q} and θ^{π} respectively.
Initialize target networks $Q'(s,a;\theta^{Q'})$ and $\pi'(s,a;\theta^{\pi'})$ with $\theta^{Q'} = \theta^{Q}$, $\theta^{\pi'} = \theta^{\pi}$
for all episode **do**
  Initialize s_1
  Initialize random process $\mathcal{N}$
  repeat
    for each step t in an episode **do**
      Select an action $a_t \in \mathcal{A}$ through $a_t = \pi(s_t;\theta^{\pi}) + \mathcal{N}$.
      Execute action a_t, observe reward r_{t+1} and new state s_{t+1}.
      Store transition $\langle s_t, a_t, r_t, s_{t+1}\rangle$ in replay buffer $\mathcal{D}$.
      Sample random mini-batch of transitions $\langle s_i, a_i, r_i, s_{i+1}\rangle$ from $\mathcal{D}$.
      Update the critic network by performing a gradient descent on $(y_i - Q(s_i, a_i|\theta^{Q}))^2$.
      Update actor network using (3.12).
      Update the target networks using

$$\begin{aligned} \theta^{Q'} &= \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\pi'} &= \tau\theta^{\pi} + (1-\tau)\theta^{\pi'} \end{aligned}$$

    end for
  until terminated
end for

Here it should be pointed out that the noise used in the previous algorithm is action space noise since the noise is added directly to the output of the actor network.

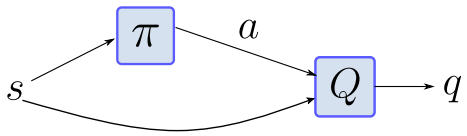


Figure 3.5: Architecture of DDPG where the policy π is trained by backpropagating the q-gradient with respect to action a

3.5 Summary

In this chapter, it was discussed how to apply deep learning to reinforcement learning problems while guaranteeing training stability. This is done by introducing experience replay and target network concepts. Additionally, the deep deterministic policy gradient algorithm within the off-policy actor-critic framework is discussed upon which the proposed approach in this thesis is built.

4 Grid Mapping with Rao-Blackwellized Particle Filters

In this chapter, a grid-based SLAM with Rao-Blackwellized particle filters introduced in [35] is discussed in order to provide an accurate estimate of the robot's pose and a partial map of the environment that can be utilized later in the reward shaping of the reinforcement learning in an attempt to speed-up the learning rate.

4.1 A Brief Introduction to SLAM Problem

Building maps is one of the fundamental tasks of mobile robots. In literature, the mobile robot mapping problem is often referred to as the simultaneous localization and mapping (SLAM) problem [36]. SLAM depicts the process of a robot creating a map of an unknown environment while simultaneously estimating its own position within the self-created map. The main challenge of the SLAM problem is the cyclic dependency between both tasks, namely, localization and mapping, i.e., for localization, a consistent map of the environment is required and for acquiring the map, a robot needs a good estimate of its location. SLAM is considered to be one of the most important functionalities a mobile robot must posses in order to become truly autonomous [37]. In order to introduce the basic concepts and notations, the classical probabilistic framework will be used.
The pose of the robot which is to be estimated at a certain time instance is represented by the state vector x_t where the dimension of x_t is problem dependent. In case of a planar mobile robot, the robot under study, $x_t \in \mathbb{R}^3$ since it consists of the 2D coordinates and the heading of the robot $x_t = (x, y, \theta)^T$. The second variable to be estimated is the map of the environment m that can take different forms depending on the chosen map representation which can be feature-based, grid-based, volumetric or topological. It is often assumed that the map is static and, therefore, does not evolve with time. Additionally, the robot is equipped with multiple information sources which provide the required perception inputs. The information that is provided to the robot can be classified into two classes, namely idiothetic and allothetic [38]. Idiothetic information is related to internal cues and is regarded as the control input u_t. This internal information can be retrieved from velocity commands, wheel encoders, inertial measurement unit (IMU), etc. On the other hand, the allothetic information is related to the external cues which can be represented as a landmark in the environment that the robot can observe and deduce its pose relative to it. These measurements are denoted by vector z_t.

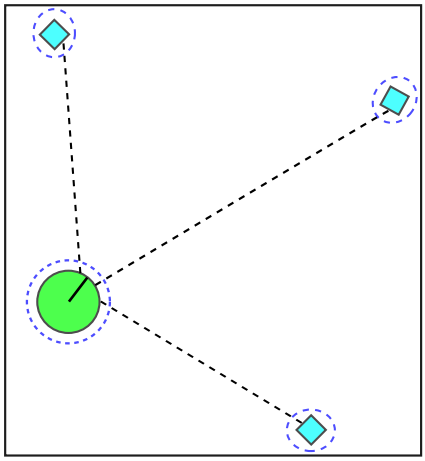
The robot then seeks to calculate the conditional probability given in equation (4.1) that estimates the trajectory of the robot $x_{1:t}$ and the map m based on both idiothetic and allothetic information given by $u_{1:t-1}$ and $z_{1:t}$ respectively.

$$p(x_{1:t}, m | z_{1:t}, u_{1:t-1}) \tag{4.1}$$

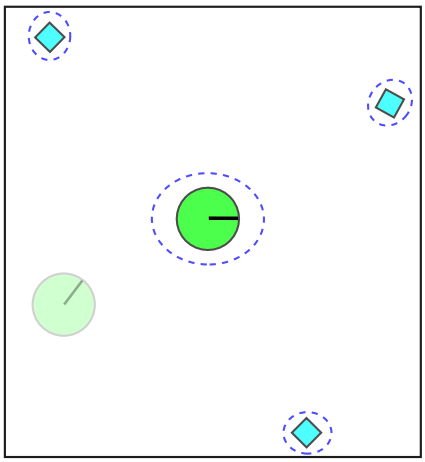
There are different methods to solve the conditional probability which are, in general, based on Bayes filters that is considered the classical and popular technique used for SLAM.

4.2 Rao-Blackwellized particle filter

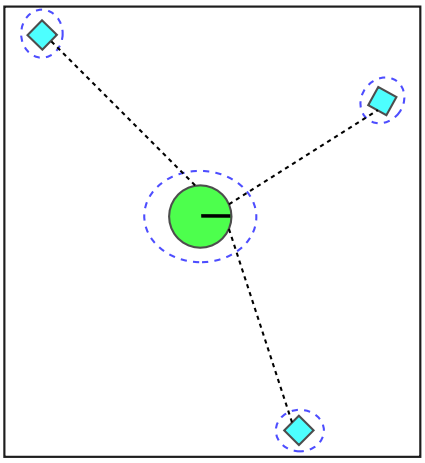
The main principle of Rao-Blackwellized particle filter (RBPF) is to estimate the joint posterior $p(x_{1:t}, m | z_{1:t}, u_{1:t-1})$ about the trajectory of the robot, that is the sequence of its poses, $x_{1:t} = x_1, ..., x_t$ and the map m given the odometry measurements $u_{1:t-1} = u_1, ..., u_{t-1}$ and a set of observations $z_{1:t} = z_1, ..., z_t$ obtained by the mobile robot. Thus, it incrementally processes the odometry readings and sensor observations as they are available. The key idea of a Rao-Blackwellized particle filter for SLAM is to separate the estimate of the trajectory $x_{1:t}$ of the



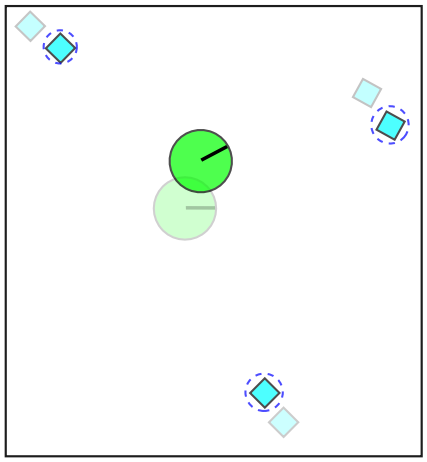
(a) The robot initializes its pose and landmarks along with the corresponding uncertainties



(b) The robot executes a motion resulting in an increase in its uncertainty



(c) The robot re-observes the landmark



(d) The robot corrects its pose along with the belief of the landmark

Figure 4.1: A graphical representation of the SLAM process using the classical filtering approach.

robot from the map m of the environment. This is done by the following factorization:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|x_{1:t}, z_{1:t})p(x_{1:t}|z_{1:t}, u_{t-1}), \tag{4.2}$$

The advantage of this factorization is the fact that the estimation of the joint posterior can be divided into two separate steps. It is possible to estimate firstly the trajectory of the robot using a particle filter where every particle represents a potential trajectory of the robot as will be shown later in the next subsection. Then, it uses this trajectory in order to estimate the posterior of the map $p(m|x_{1:t}, z_{1:t})$ using "mapping with known poses" [39] since $z_{1:t}$ and $x_{1:t}$ are known from the previous step as discussed in section 4.2.2.

4.2.1 Particle Filter Estimator

The purpose of particle filters is to estimate the posterior of the robot given the control actions and measurements. Particle filters are one of the tractable implementations of the Bayes filter for continuous space. Thus, it is worth giving a very brief introduction to Bayes filters firstly. By applying Bayes rule to the posterior of the robot, $p(x_{1:t}|z_{1:t}, u_{1:t-1})$, the conditional probability can be broken down into the following product

$$p(x_{1:t}|z_{1:t}, u_{1:t}) = \eta p(x_{1:t}|z_{1:t-1}, u_{1:t})p(z_t|x_{1:t}, z_{1:t-1}, u_{1:t}) \tag{4.3}$$

where $\eta = 1/p(z_t|z_{1:t-1}, u_{1:t})$ is a normalization factor.
By applying Markov assumption and the law of total probability, equation (4.3) can be formulated in a recursive form

$$p(x_{1:t} \mid z_{1:t}, u_{1:t}) = \eta \underbrace{p(z_t \mid x_{1:t})}_{\text{correction}} \int_{x_{1:t-1}} \underbrace{p(x_{1:t} \mid x_{1:t-1}, u_t)}_{\text{prediction}} \underbrace{p(x_{1:t-1} \mid z_{1:t-1}, u_{1:t})}_{\text{recursive term}} \mathrm{d}x_{1:t-1} \tag{4.4}$$

Equation (4.4) is considered to be the general expression of the recursive Bayes filter where the prediction estimate encodes the motion and propagate the probability density function through a motion model. In addition, the correction step takes into account the allothetic information and correct the prediction estimate based on the measurement model. A graphical representation of the SLAM process using the classical filtering approach is shown in Figure 4.1.
The complete Bayes filter algorithm is shown below

Algorithm 5 Bayes filter

for all x_t **do**
$\overline{bel}(x_t) = \int p(x_t|u_t, x_{t-1}) bel(x_{t-1}) \mathrm{d}x$
$bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t)$
end for

where $bel(x_t)$ is the belief which reflects the robot's internal knowledge about its state and is given by the posterior described in equation (4.4) and $\overline{bel}(x_t)$ represents the belief before the correction step.
Particle filters are a type of non-parametric filters meaning that they do not rely on a fixed function for the posterior, such as a Gaussian function used by Extended Kalman Filter (EKF) parametric filter, for example. Instead, particle filters approximate the belief of the robot's base $bel(x_t)$ by a set of finite particles $\mathcal{X}_t$.

$$\mathcal{X}_t = x_t^{(1)}, x_t^{(2)}, ..., x_t^{(N)} \tag{4.5}$$

where each particle $x_t^{(n)}$ with $(1 \leq n \leq N)$ represents a hypothesis of what the true state may be at time t and N denotes the number of particles in the particle set $\mathcal{X}_t$. Accordingly, the likelihood of a state hypothesis $x_t^{(n)}$ to be included in the particle set $\mathcal{X}_t$ is proportional to its belief $bel(x_t^{(n)})$; $x_t^{(n)} \sim p(x_t|z_{1:t}, u_{1:t})$. This means that the denser a sub-region in the state-space is populated with samples, the more likely the true state lies into this region [36]. Here it should also be pointed out that an individual map is associated with each particle.
Since a particle filter is an implementation of the Bayes filter, it constructs its current belief $bel(x_t)$ recursively from the previous belief $bel(x_{t-1})$ as discussed before. Moreover, since the belief of the robot is represented by a particle set $\mathcal{X}_t$, this means that the next generation of particles $\mathcal{X}_t$ is obtained recursively from the previous generation $\mathcal{X}_{t-1}$ by sampling from a proposal distribution, i.e. the probabilistic odometry motion model $p(x_t^{(n)}|x_{t-1}^{(n)}, u_{t-1})$. Then, by incorporating the probabilistic observation model $p(z_t|x_t^{(n)})$, an individual importance weight $w_t^{(n)}$ is assigned to each particle. After that, particles are drawn with a replacement proportional to their assigned importance weight. This step is called a resampling step. As a matter of fact, after resampling, all particles have the same weight. By incorporating the importance weights into the resampling process, the distribution of the particles changes; whereas before the resampling step, they were distributed according to $\overline{bel}(x_t)$, after resampling, they are distributed according to the posterior $bel(x_t) = \eta p(z_t|x_t^{(n)}) \overline{bel}(x_t)$ where the particles with lower importance weights are depleted. The complete particle filter algorithm is given below

Algorithm 6 Particle filter [36]

$\overline{\mathcal{X}_t} = \mathcal{X}_t = \phi$
for $n = 1$ to N **do**
  sample $x_t^{(n)} \sim p(x_t | x_{t-1}^{(n)}, u_t)$
  $w_t^{(n)} = p(z_t | x_t^{(n)})$
  $\overline{\mathcal{X}_t} = \overline{\mathcal{X}_t} + \langle x_t^{(n)}, w_t^{(n)} \rangle$
end for
for $n = 1$ to N **do**
  draw n with probability $\propto w_t^{(n)}$
  add $x_t^{(n)}$ to $\mathcal{X}_t$
end for

where $\overline{\mathcal{X}}_t$ is a temporary particle set before the resampling step. One drawback of this algorithm is that whenever a new observation is available, it is required to evaluate the weights of the trajectories again from scratch which is not computationally efficient. In [40], a recursive formulation to compute the importance weights is represented under the restriction that the proposal distribution should fulfill the following assumption

$$\pi(x_{1:t} | z_{1:t}, u_{1:t-1}) = \pi(x_t | x_{1:t-1}, z_{1:t}, u_{1:t-1}) \pi(x_{1:t-1} | z_{1:t-1}, u_{1:t-2}) \tag{4.6}$$

Based on that

$$\begin{aligned} w_t^{(n)} &= \frac{p\left(x_{1:t}^{(n)} \,\middle|\, z_{1:t}, u_{1:t-1}\right)}{\pi\left(x_{1:t}^{(n)} \,\middle|\, z_{1:t}, u_{1:t-1}\right)} \\ &= \frac{\eta p\left(z_t \,\middle|\, x_{1:t}^{(n)}, z_{1:t-1}\right) p\left(x_t^{(n)} \,\middle|\, x_{t-1}^{(n)}, u_{t-1}\right)}{\pi(x_t | x_{1:t-1}, z_{1:t}, u_{1:t-1})} \underbrace{\frac{p\left(x_{1:t-1}^{(n)} \,\middle|\, z_{1:t-1}, u_{1:t-2}\right)}{\pi(x_{1:t-1} | z_{1:t-1}, u_{1:t-2})}}_{w_{t-1}^{(n)}} \\ &\propto \frac{p\left(z_t \,\middle|\, x_t^{(n)}, m_{t-1}\right) p\left(x_t^{(n)} \,\middle|\, x_{t-1}^{(n)}, u_{t-1}\right)}{\pi(x_t | x_{1:t-1}, z_{1:t}, u_{1:t-1})} w_{t-1}^{(n)} \end{aligned} \tag{4.7}$$

Thus, using this recursive structure results in a computationally efficient algorithm. Here it is worth mentioning that the general expression of the sample weight is the one given in equation (4.7) where it is the ratio of the target distribution $p\left(x_{1:t}^{(n)} \,\middle|\, z_{1:t}, u_{1:t-1}\right)$ and the proposal distribution $\pi\left(x_{1:t}^{(n)} \,\middle|\, z_{1:t}, u_{1:t-1}\right)$ as defined in [41]. However, if the odometry model is used as the proposal distribution, the particle weight will be given as the observation model as defined in Algorithm 6 which can be easily derived from equation (4.7).
Although the particle filter algorithm described in Algorithm 6 achieved good results as shown in [42; 43], in [41], it is shown that choosing the odometry motion model $p(x_t | x_{t-1}, u_{t-1})$ as the proposal distribution could lead to sub-optimal solutions when the observation model is significantly more precise than the odometry motion model which is the case if the robot is equipped with a laser range finder. The reason is that the meaningful area of the observation likelihood is substantially smaller than that of the motion model. Thus, by using the odometry motion model as the proposal model, the importance weight of the individual particle will differ significantly from one another since only a fraction of the samples cover the meaningful area, the area in which an overlap occurs between the probability likelihood of the odometry motion model and observation model. As a result, a comparably high number of particles is required to sufficiently cover the meaningful area of the distribution in this case. According to [44], to overcome this problem, the most recent observation z_t is incorporated when generating the next generation of particles. Based on that, the optimal proposal distribution can be

given as

$$p\left(x_t^{(n)} \,\middle|\, x_{t-1}^{(n)}, u_{t-1}, z_t, m_{t-1}^{(n)}\right) = \frac{p\left(x_t^{(n)} \,\middle|\, x_{t-1}^{(n)}, u_{t-1}\right) p\left(z_t \,\middle|\, x_t^{(n)}, m_{t-1}^{(n)}\right)}{\int_{x_t} p\left(z_t \,\middle|\, x_t^{(n)}, m_{t-1}^{(n)}\right) p\left(x_t^{(n)} \,\middle|\, x_{t-1}^{(n)}, u_{t-1}\right) \mathrm{d}x_t} \tag{4.8}$$

In [45], it is claimed that the target distribution, in most of the cases, has only a single peak. Thus, it is possible to sample poses in the surrounding of this peak while ignoring the less meaningful areas of distributions which saves a significant amount of computational resources since it requires less number of samples. This leads us to the computation of an improved proposal distribution introduced in [41] that can be approximated by a Gaussian function. The mathematical derivation of this Gaussian distribution is discussed in Appendix B. By following this proposal distribution, the number of particles reduced dramatically as claimed in [41].

4.2.1.1 Selective Resampling

Another aspect that has a major influence on the performance of the particle filter is the resampling step. During the resampling step, particles with low importance weights $w^{(n)}$ are typically replaced by particles with high weights. Although, the resampling step is important due to the fact that only a finite number of samples are used to approximate the target distribution, the resampling step can delete good samples from the sample set resulting in particle depletion. Additionally, although a recursive relation is formulated to calculate the importance weights of the particles after every resampling step, it is still computationally expensive to calculate the importance weights every time-step. For these reasons, resampling makes sense if particle weights differ significantly. In [46], an empirical criterion to specify the necessity of applying the resampling step is introduced based on how well the current particle set represents the true posterior. This is done by introducing the so-called effective number of particles n_{eff} which is given by

$$n_{eff} = \frac{1}{\sum_{n=1}^{N} \left(\tilde{w}^{(n)}\right)^2} \tag{4.9}$$

where $\tilde{w}^{(n)}$ is the normalized importance weight of particle n. Here, to avoid confusion, it should be pointed out that the term "normalized" in this context is different from the one used in reinforcement learning context. Here, the normalized importance weight is given by $\tilde{w}^{(n)} = \frac{w^{(n)}}{\sum_{n=1}^{N} w^{(n)}}$. According to this, the resampling step takes place only when the variance of the importance weights is high which results in a bad approximation of the target distribution. The high variance can be triggered if the effective number of particles n_{eff} drops below a predefined threshold which is, according to [44], given by $N/2$ where N is the total number of particles in the particle set $\mathcal{X}$. By following this strategy, it was proved that it reduces the risk of particle depletion since the resampling step is carried out only when needed [45].

4.2.2 Occupancy Grid Mapping

As stated previously, the advantage of using Rao-Blackwellized particle filter is to separate the estimate of the robot's trajectory $x_{1:t}$ from the estimated of the environment's map m. As shown in the previous section, the trajectory of the robot is estimated using a particle filter. Thus, in this section, the trajectory is used to estimate the posterior of the map using "mapping with known poses". This will be done using occupancy grid mapping as a way to model the environment. Occupancy grid maps are often favored over feature-based maps as they do not require an explicit definition of the landmarks and could offer a more informative representation of the environment [45].

Occupancy grid maps address the problem of generating consistent maps from noisy and uncertain measurement data, under the assumption that the robot pose is known [36]. In contrast to landmark-based mapping where a gaussian distribution to estimate the location of the landmark is required, grid-mapping has no parametric model. Instead, the occupancy grid map

divides the worksapce into evenly spaced cells where a probability distribution is assigned to each cell in the grid indicating whether it is occupied or free. As a matter of fact, the resolution of the grid cells should be compatible with the smallest feature of the environment that should be considered as an obstacle. The posterior over maps given the trajectory of the robot $x_{1:t}$ and all the observations $z_{1:t}$ up to time t is given as $p(m|x_{1:t}, z_{1:t})$ where the controls $u_{1:t}$ play no role since the path of the robot is already known. In occupancy grid maps, it is assumed that the probability of every grid cell, whether it is occupied or not, is independent of each other. By making advantage of this assumption, the certainty of the estimation of the entire map can be broken down into the problem of estimating the posterior of every grid cell m_i in the map and then the posterior over the entire map can be approximately estimated by:

$$p(m|z_{1:t}, x_{1:t}) = \prod_{i=0}^{M} p(m_i|z_{1:t}, x_{1:t}) \tag{4.10}$$

where M represents the number of grid cells in the map. The probability of every grid cell $p(m_i|z_{1:t}, x_{1:t})$ can be easily derived from Bayes rule where the full derivation can be found in [36]. Equation (4.10) plays an important role in this thesis since it is used to reshape the reward function as discussed in chapter 5.

4.3 Summary

This chapter gave a brief introduction to the grid mapping with Rao-Blackwellized particle filter method that will be used for shaping the reward function in the motion planner design as described in the next chapter. The advantage of using this approach is that it depends on the lidar data points and the odometry readings which are already part of the reinforcement learning algorithm, thus no additional hardware is required. Furthermore, the number of particles required to build the map and estimate robot's pose is reduced significantly which makes this approach efficient from the computational complexity point of view.

5 Motion Planner Design

In this chapter, the theoretical background discussed beforehand is applied to the problem of mobile robot navigation in an unknown environment while avoiding obstacle collisions. As discussed previously, the purpose of this study is twofold. Firstly, it is aimed to develop a deep deterministic policy gradient (DDPG) based approach within an off-policy actor-critic framework to navigate a non-holonomic mobile robot continuously to randomly distributed targets. This is done by taking into account only sparse low-dimensional laser ranger finder readings. In that sense, firstly, the structure of both the actor and critic neural networks are discussed in detail in section 5.1. The second purpose is to integrate the "partial" knowledge of the map, obtained by the SLAM algorithm to assess how much it speeds up the convergence rate and improves the optimality of the trajectories. To achieve the second purpose, the way in which the online-acquired map during training is incorporated in the reward function based on a grid-based Rao-Blackwellized particle filter to improve the learning rate is introduced in section 5.2. Finally, since the adopted algorithm is an off-policy learning approach, it is possible to use a behavioural policy which is different from the learned one. For that purpose, different stochastic exploration policy methods are discussed in section 5.3.

5.1 Actor-Critic Networks' Structure

To achieve the first purpose, two neural networks are constructed to represent the the actor and the critic respectively. The actor network represents the policy and thus it is responsible for mapping the states into actions $a_t = \pi(s_t)$. The main framework that represents the processed gathered information is shown in Figure 5.1. The states ($\mathcal{S} \in \mathbb{R}^{14}$) are selected to be the observation from the laser range finder that can be represented as 10-dimensional laser beams with 180° field of view (FOV) x_t, the relative distance between the target and the agent represented in polar coordinates p_t and finally the last action executed by the agent v_{t-1} whereas, the output actions are the continuous angular and linear velocities:

$$v_t = \pi(s_t) = \pi(x_t, p_t, v_{t-1}) \tag{5.1}$$

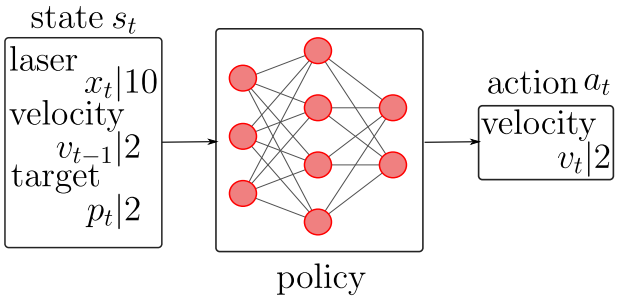


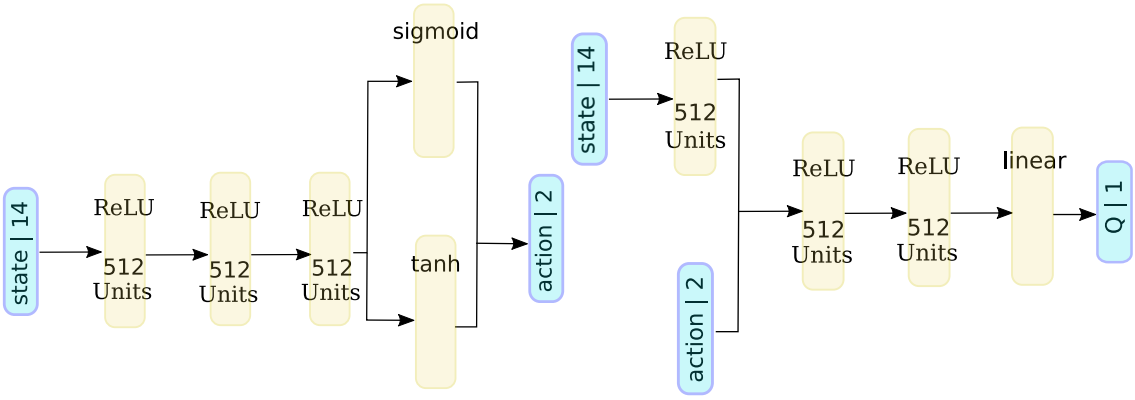
Figure 5.1: The agent is trained based on 10-dimensional sparse laser sensor x_t, last executed action v_{t-1} and the relative distance between the agent and the target in polar coordinates p_{t-1}. The output actions are continuous linear and angular velocities v_t.

The actor's neural network is composed of three fully-connected hidden layers with 512 nodes each which are activated by a rectified linear unit (ReLU) activation function. The output of the actor's network is a 2-dimensional vector representing the linear and angular velocities of the robot respectively ($\mathcal{A} \in \mathbb{R}^2$). For this purpose, a sigmoid activation function is used to constrain the linear motion of the robot in the range between [0,1]. There are two reasons for restricting the backward motion of the robot. Firstly, the chattering behaviour of the robot observed at

the preliminary experiments due to the stochasticity of the behavioral policy. In addition, since the field of view of the laser sensor is selected to be 180°, it does not make much sense to allow backward motion of the robot since that could result in a collision with the obstacles. Furthermore, to constrain the angular velocity of the robot within [-1,1], a hyperbolic tangent function "tanh" is employed. Moreover, the output of the actor network is further multiplied by hyperparameters to limit the maximum linear velocity of the robot to 0.25m/s and the maximum angular velocity to 1rad/s. The actor outputs, thus the actions, are then sent to the low-level controller to control the motion of the robot's actuators. The layout of the actor neural network is depicted in Figure 5.2a.
On the other hand, the critic network estimates the Q-value of a state-action pair and thus it takes both the state and the action as inputs. Similarly to the architecture described by [6], the action are not included until the second layer in the critic network to force the network to learn representations from states alone first. The output of the critic is an estimation of the reward. Based on the output of the critic network, the weights of both the actor and critic are updated accordingly. Like the actor network, the hidden layers of the critic network are activated by a ReLU function. The Q-value is finally activated through a linear activation function:

$$y = wx + b, \tag{5.2}$$

where x is the input of the last layer, y is the predicted Q-value, w and b are the trained weights and bias of the last layer. Figure 5.2b shows the architecture of the critic network.



(a) Layout of the actor network for DDPG.

(b) Layout of the critic network for DDPG.

Figure 5.2: A graphical representation of the actor-critic neural networks in the context of mobile robot navigation problem.

5.1.1 Batch Normalization

The training of deep network is troublesome due to the fact that the inputs of a layer are affected by the parameters of all preceding layers so that small variations to the parameters of the network are amplified as the network becomes deeper. This results in the change in the distribution of the inputs to the inner nodes within the network. This change in the distribution of inputs to layers in the network is referred to as "internal covariate shift" [47]. As mentioned previously in section 3.1.1, in backpropagation, the model is updated layer-by-layer backward from the output to the input assuming that the weights of the layer prior to the current layer are fixed [48]. However, this is not the case since the weights of the entire network are updated simultaneously resulting in a change of the distribution after the weights of the previous layers are updated. For this purpose, batch normalization is proposed [47]. The basic idea behind batch normalization is to normalize the scalar features "neurons" independently over

each mini-batch $\mathcal{B} = \{x_{1,...,m}\}$, by making it have a mean of zero and a unit variance.

$$\hat{x}_k = \frac{x_k - \mu_\mathcal{B}}{\sigma_\mathcal{B}} \tag{5.3}$$

where the mean and variance are computed over the mini-batch $\mathcal{B}$. In that sense, normalizing the activations of the prior layers makes it possible to assume that the distribution of the inputs during the weight updates will not change, at least not dramatically as the case before normalization takes place. However, normalizing every input of a layer may change what the layer represents. Accordingly, the authors of [47] recommend to scale and shift the normalized value,

$$y_k = \gamma \hat{x}_i + \beta \tag{5.4}$$

where the parameters γ and β are learned along with the model weights and biases. An advantage of this scale and shift is that it allows the network to recover its value before normalization if it decides that this is the optimal solution by setting $\gamma = \sigma_\mathcal{B}$ and $\beta = \mu_\mathcal{B}$.

Algorithm 7 Batch Normalization

Inputs: Values of x over a mini-batch: $\mathcal{B} = \{x_{1,...,m}\}$;
$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{k=1}^{m} x_k$ mini-batch mean
$\sigma^2_\mathcal{B} \leftarrow \frac{1}{m}\sum_{k=1}^{m} (x_k - \mu_\mathcal{B})^2$ mini-batch variance

$\hat{x_k} \leftarrow \frac{x_k - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}$ normalize
$y_k \leftarrow \gamma \hat{x}_k + \beta$ scale and shift

In algorithm 7, ϵ is a small constant added to avoid zero division and thus guarantees numerical stability.

5.2 Reward Function Definition

The reward function is the most important aspect in a reinforcement learning problem since the actions are selected in such a way that the cumulative reward is maximized. The reward signal is the mean by which the goal of the learning is specified for the agent. It is a designed task-specific function that, given the action of the agent and the state of the system, returns a single real number indicating how good or bad that action was. The reward signal corresponds to pleasure and pain in biological systems. Designing a good reward signal for a robotic reinforcement learning task can be challenging in different ways. This area of reinforcement learning, known as reward designing or shaping, is considered an art rather than a well-established science [18]. Not to mention that the reward function purpose is not to tell the learning algorithm *how* to achieve a certain goal but rather *what* to achieve. Accordingly, the learning algorithm should then use this reward to discover the necessary actions to execute in order to achieve the ultimate goal without being hard-coded. For the navigation problem, reward functions can be a simple bonus when the agent reaches a target and, consequently, a penalty in case it hits an obstacle. This *sparse reward* is assigned to prioritize actions that make the agent reach the goal and penalize actions that make the agent colliding. On the other hand, the reward can be more sophisticated and depend on the distance between the agent and the target. This is called *dense reward*. In the autonomous navigation problem, the advantage of using a dense reward over a sparse reward can be intuitively formulated as follows. In case of a dense reward, if a sequence of robot's actions results in a positive reward, the parameters of the policy network are updated so that the probability of taking that set of actions is more often in the future. However, in case of a sparse reward, the robot continues taking random actions until, by chance, it gets some non-zero reward. The disadvantage of this is that since

the non-zero rewards are seen so rarely as it happens only when the robot reaches the target, the sequence of actions that resulted in the reward might be very long. More importantly, it is not clear which of the these actions were really useful in getting the reward. In reinforcement learning context, this problem is known as *credit assignment*.

This section first discusses how a reward function is designed for the autonomous navigation problem for mobile robots. It then presents a novel idea to shape reward functions. The idea is based on shaping the reward function based on the online-acquired knowledge about the environment which is provided in a form of an occupancy grid map that the robot builds during training.

5.2.1 Designing the Reward Function

In this section, two different reward functions are defined to assess the quality of the performance of the robot while interacting with its environment.

5.2.1.1 Exponential Euclidean Distance

The goal is to move the robot towards a defined goal position. The agent receives a penalty proportional to the exponent of the euclidean distance between its current position and the goal position. The euclidean distance between the robot and the target is simply evaluated by

$$d = \left\| \mathbf{p}_t^{x,y} - g \right\|_2 = \sqrt{\left(\mathbf{p}_t^{y} - g^{y}\right)^2 + \left(\mathbf{p}_t^{x} - g^{x}\right)^2} \tag{5.5}$$

where $\mathbf{p}_t$ represents the current position of the robot at time t with respect to the inertial frame. Then, the reward based on the exponential function of the euclidean distance is evaluated as follows

$$r_{\text{exp}} = 1 - e^{\gamma d} \tag{5.6}$$

where γ represents the decay rate of the exponent. In this perspective, all the states closed to the goal would receive much higher rewards than the ones far away. In addition, a sparse reward is added if the agent reaches the target position within the interval of a predefined tolerance. On the other hand, the agent would receive a high negative reward "penalty" when it gets too close to an obstacle. Here it should be pointed out that the episode is terminated in three different scenarios; i) the agent reaches the goal with some tolerance d_{min}, ii) the agent gets closer to an obstacle with a minimum threshold, iii) the agent exceeds the maximum number of allowed time-steps T in every episode without either reaching the target or hitting an obstacle. The maximum number of iterations per episode is a hyperparameter that is tuned based on the average number of actions required by the agent to reach the goal observed during the preliminary experiments. The reward $r(s_t, a_t)$ is given after executing every navigation action a_t and can be, mathematically, formulated as:

$$r(s_t, a_t) = \begin{cases} r_{\text{reached}}, & d < d_{min}, \\ r_{\text{crashed}}, & s_{ts}, \\ 1 - e^{\gamma d}, & \text{otherwise.} \end{cases} \tag{5.7}$$

where d is the euclidean distance between the agent and the target, γ is a hyper-parameter that can be tuned and s_{ts} represents an undesirable terminal state including getting too close to an obstacle or exceeding the maximum number of steps allowed in an episode.

5.2.1.2 Difference in Distance in two consecutive time-steps

The second reward function can be given as

$$r_{\text{diff}} = r(s_t, a_t) + \lambda^{\omega} r_t^{\omega} \tag{5.8}$$

where $r(s_t, a_t)$ is the reward based on the difference in the distance between the agent and the target in two consecutive time steps $d_{t-1} - d_t$. This means that the reward would be positive in case the agent is moving towards the target and negative otherwise. To motivate the robot to move towards the target, this term is multiplied by a hyperparameter "scaling factor" λ^g. This distance-based reward can be formulated as

$$r(s_t, a_t) = \begin{cases} r_{\text{reached}}, & d < d_{min}, \\ r_{\text{crashed}}, & s_{ts}, \\ \lambda^g \left(\left\| \mathbf{p}_{t-1}^{x,y} - g \right\|_2 - \left\| \mathbf{p}_t^{x,y} - g \right\|_2 \right), & \text{otherwise.} \end{cases} \tag{5.9}$$

In addition, an orientation-based reward r_t^ω is added to motivate the robot to correct its heading with respect to the target. This term is defined as

$$r_t^\omega = \left\| \text{atan2}\left(\mathbf{p}_t^y - g^y, \mathbf{p}_t^x - g^x \right) - \mathbf{p}_t^\omega \right\|_1 \tag{5.10}$$

5.2.2 Shaping the Reward Function

In this subsection, the reward function is shaped based on the available knowledge about the environment gained throughout the robot's experience. For this purpose, a 2D occupancy grid map of the surrounding built by the SLAM algorithm discussed in chapter 4 is generated while the robot is exploring the unknown environment, using data extracted from laser range finder and the robot's odometry information. Every cell inside the occupancy grid is classified as (occupied, free, unknown) based on a predefined threshold value that determines the occupation probability of each cell. Furthermore, the occupation probability of every cell is being updated while the robot keeps exploring the environment. In that sense, the reward function does not only depend on how far the agent is from the target but on the distance to the multi-obstacles inside the workspace as well. The incorporation of the environment's knowledge should be weighted by the level of certainty of the map's posterior $p(m|z_{1:t}, x_{1:t}) = \prod_{i=0}^{M} p(m_i|z_{1:t}, x_{1:t})$. Moreover, since every obstacle inside the environment is represented by a number of occupied grid cells, this part of the reward is normalized by the total number of occupied grid cells in the field of view (FOV) of the robot. This can be formulated as follows:

$$r(s_t, a_t) = \underbrace{\frac{1}{k} \prod_{i=0}^{M} p(m_i|z_{1:t}, x_{1:t}) \sum_{i=0}^{k} e^{-c_{min}}}_{\text{map-dependent term}}, \tag{5.11}$$

where M is the total number of grid cells in the constructed map, k is the total number of occupied cells in the field of view of the robot and c_{min} is the distance between the robot and the occupied cell. As a matter of fact, since the reward function evolves with time due to the incorporation of the uncertainty, the reward function does not follow the MDP framework anymore. Here it should be pointed out that the map-dependent term defined in equation (5.11) is added to both rewards defined in equations (5.6) and (5.8) and a comparison between these four rewards is made in chapter 7.

5.3 Exploration Noise

The exploration vs. exploitation dilemma is a long standing issue in reinforcement learning. Since the agent has no knowledge about its environment initially, it has to explore. Once the agent finds states of high rewards, it should learn to exploit behaviors that lead towards these high rewards. However, it still remains important to explore since there may still be strategies that are more optimal that have not yet been discovered. In practice, most reinforcement learning agents rely on action space noise where the noise is introduced in additive manner to the output of the actor neural network and thus does not affect its training; $\tilde{\pi}(a|s) = \pi_\theta(a|s) + \mathcal{N}(0, \sigma^2 I)$. As mentioned previously, this additive noise can be represented

by either Gaussian noise or the temporarily correlated Ornstein-Uhlenbeck process. Although these approaches could work properly, one drawback of the action space noise is that the noise is not conditioned on the current state. In other words, whenever a certain state s occurs, the actual action that the policy selects will be vastly different. However, at the same time, a deterministic policy cannot be simply used since it always outputs the same action given the same state. In this way, the agent would not explore at all. For this purpose, parameter space noise introduced in [49] is discussed in this section.
The central idea of parameter space noise is to move the noise process from the actions to the parameters of the policy π_θ where θ is the parameter vector. Thus, instead of adding noise to the actions, parameter space noise perturbs the parameters $\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 I)$. By doing so, the produced action is now fully conditioned on the state s through the perturbed behavioural policy. Figure 5.3 depicts the difference between traditional action space noise and parameter space noise. Although the idea of the parameter space noise is very simple, the naïve implementation of it may cause the learning algorithm to diverge. This, in principle, is due to the following reasons:

(i) Different layers within the neural network may exhibit different sensitivity to the parametric perturbations.

(ii) The effect of the perturbation in parameter space noise cannot be intuitively understood and thus, it is hard to reason about in contrast to the actions space noise. Accordingly, picking the right scale of the noise σ would be difficult.

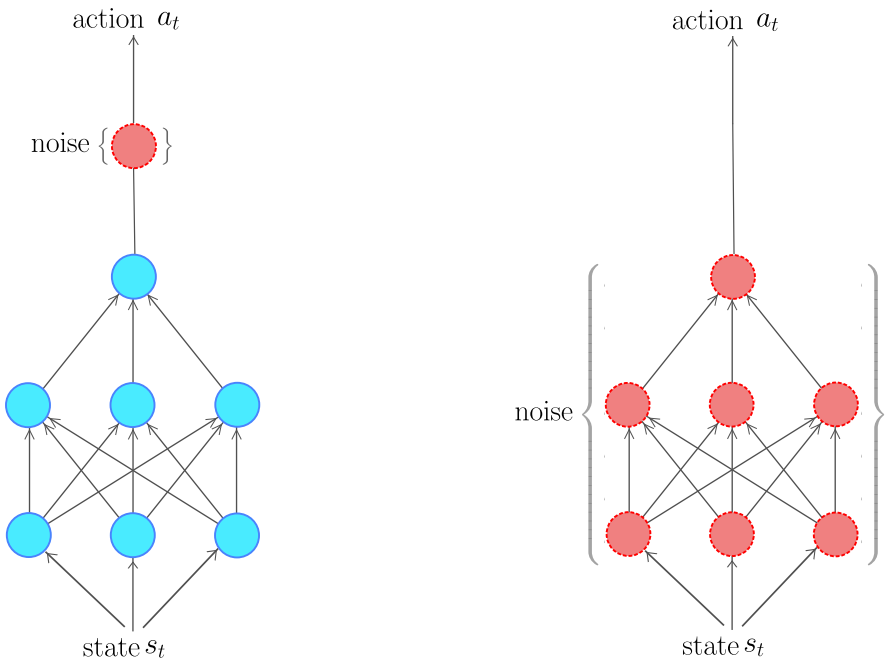
To tackle the first problem, layer normalization [50] is applied to each fully-connected layer to guarantee that the output of all the layers has approximately zero mean and unit variance. Layer normalization differs from batch normalization discussed in section 5.1.1 in the sense that the statistics are computed over all the hidden neurons in the same layer and not across the batches.

$$\mu = \frac{1}{H}\sum_{i=1}^{H} a_i, \qquad \sigma = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i - \mu)^2} \tag{5.12}$$

where H denotes the number of hidden neurons in a layer, a_i is the projection of the input x using the weight w_i. Accordingly the output of every neuron of the hidden layer can be given as

$$h_i = f\left[\frac{1}{\sigma}(a_i - \mu_i) + b\right] \tag{5.13}$$

where $f[.]$ denotes the non-linearity (e.g. a ReLU) and b is the learnable bias. An advantage of layer normalization over batch normalization is that it does not depend on the size of the current batch. Furthermore, it can be applied to recurrent neural networks, however, this is not the scope of the presented work. Thus, it can be concluded that, layer normalization is crucial for parameter space noise since it allows using the same perturbation scale across all layers. In this way, the first issue is resolved. The main difference between layer normalization and batch normalization is better visualized in Figure 5.4.



(a) Action space noise **(b)** Parameter space noise

Figure 5.3: Comparison between action space noise (left) and parameter space noise (right). Depicted in red are the parts subjected to noise. Recreated from [49].

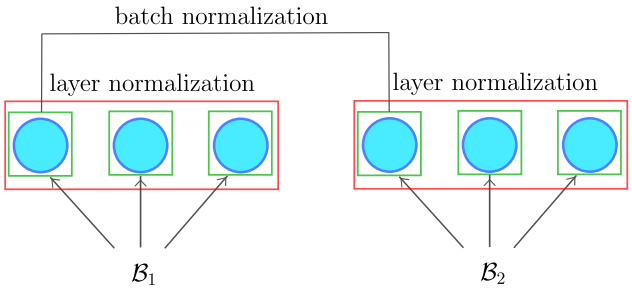


Figure 5.4: Difference between batch normalization and layer normalization. In layer normalization, statistics are computed across the neurons of each layer in contrast to batch normalization where statistics are computed across the mini-batch.

Another issue of parameter space noise is concerned with determining an appropriate value for the variance σ^2 of the additive Gaussian noise. In contrast to action space noise where the effect of the perturbation on the executed actions can be easily understood, this problem is non-trivial in case of parameter space noise. The reason is that, in deep neural networks, there are hundreds of thousands of parameters, and thus it is quite difficult to intuitively understand what effect the additive perturbations can cause. This occurs, particularly, when the sensitivity to the perturbations is going to change as the network is trained since the parameters of the network are moving away from the random initialization around zero to a quite complicated and non-linear mapping from states to (sub)optimal actions. In [49], a trick is applied to deal with this issue by transferring the problem form the parameter space to the action space. This is done by introducing a function $d(.,.)$ that calculates the difference between the predicted action by a perturbed policy $\tilde{\pi}$ and a non-perturbed policy π which is represented by a scalar number. In the context of the navigation problem, this is the difference between the predicted

velocities by both policies. In this way, it becomes possible to measure the effect of the of the parameter perturbation and, subsequently, the scale of the variance of the additive noise can be adjusted. This idea can be mathematically formalized as follows

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k, & d(\pi(a|s), \tilde{\pi}(a|s)) \leq \delta, \\ \frac{1}{\alpha}\sigma_k, & \text{otherwise.} \end{cases} \tag{5.14}$$

where $d(.,.) \in \mathbb{R}$ defines a distance measure between the perturbed policy $\tilde{\pi}$ and the non-perturbed policy π, $\delta \in \mathbb{R}_{>0}$ is a positive threshold value, and $\alpha \in \mathbb{R}_{>0}$ is a positive parameter that is introduced to adjust the current value σ_k. In other words, the scale of the parameter perturbations is increased if the effect of the current scale as measured in the action space is below a desired threshold and decreased otherwise.
The application of parameter space noise to a DDPG algorithm is not troublesome since the DDPG defines an explicit policy network π. Thus, all what is required is only two forward passes for each sample through both networks $(\pi, \tilde{\pi})$.

$$d(\pi, \tilde{\pi}) = \frac{1}{\mathcal{A}} \sqrt{\sum_{i=1}^{\mathcal{A}} \mathbb{E}_s \left[(\pi(a|s)_i - \tilde{\pi}(a|s)_i)^2 \right]} \tag{5.15}$$

where $\mathbb{E}_s[.]$ is estimated using a mini-batch of states from the replay buffer, $\mathcal{A}$ denotes the number of continuous actions, and $\pi(a|s)_i$ denotes the i-th action selected by the policy π. The parameter space noise with adaptive scaling is summarized in algorithm 8.

Algorithm 8 Parameter Space Noise with Adaptive Scaling

Initialize an RL algorithm $\mathbb{A}$ and in particular $\pi = \pi_\theta$,
Initialize an adaptive scale $\alpha \in \mathbb{R}_{>0}$, a threshold $\delta \in \mathbb{R}_{>0}$, and an interval for adaptive scaling $T_{\text{adapt}} \in \mathbb{R}_{>0}$
Initialize $\tilde{\pi} = \pi_{\tilde{\theta}}$ for exploration
Initialize $\bar{\pi} = \pi_{\bar{\theta}}$ for noise adaption
for all episodes **do**
  Perturb $\tilde{\theta} \leftarrow \theta + \mathcal{N}(0, \sigma_k^2 I)$ and obtain $\tilde{\pi} = \pi_{\tilde{\theta}}$
  for each step t in an episode **do**
    Sample an action a_t using the perturbed policy $\tilde{\pi}$
    Execute action a_t and observe a new state s_{t+1}
    Execute a training step of $\mathbb{A}$
    if t mod $T_{\text{adapt}} = 0$ **then**
      Perturb $\bar{\theta} \leftarrow \theta + \mathcal{N}(0, \sigma_k^2 I)$ and obtain $\bar{\pi} = \pi_{\bar{\theta}}$
      Estimate $d_k \leftarrow \frac{1}{\mathcal{A}} \sqrt{\sum_{i=1}^{\mathcal{A}} \mathbb{E}_s \left[(\pi(a|s)_i - \tilde{\pi}(a|s)_i)^2 \right]}$
      Adapt $\sigma_{k+1} = \begin{cases} \alpha\sigma_k, & d(\pi(a|s), \tilde{\pi}(a|s)) \leq \delta, \\ \frac{1}{\alpha}\delta_k, & \text{otherwise.} \end{cases}$
      $k \leftarrow k + 1$
    end if
  end for
end for

6 Experiments

In this chapter, a variety set of experiments that are conducted to validate the effectiveness of the proposed approach are highlighted. The chapter starts by introducing the experimental setup settings in section 6.1 in which the simulation platform, computational hardware and learning parameters are all defined. Afterwards, in section 6.2, the environments on which the learning algorithms take place are described. In this same section, the different experiments that are done to answer the research questions are briefly discussed. Finally, the real-hardware setup is introduced and a proposed difference model learning algorithm that captures the mismatch between the real system and the simulated model is explained in section 6.3.

6.1 Experimental Setup

The main objective of the proposed RL&SLAM approach is to find a (sub)optimal trajectory from the current location of the robot to the target with minimum executed actions. The virtual environment is built on gazebo simulator [1] representing the 3D environment. The experiments were conducted on an Ubuntu 16.04 machine with an Intel Core i7-8550 CPU and an NVIDIA Jetson TX2 GPU. There was no speed gains observed when using a GPU. The algorithm is implemented using OpenAI package provided by the Robot Operating System (ROS) middleware. OpenAI is open source and publicly available [2]. The simulated environment contains cuboid objects representing the obstacles and a target for the agent to reach, rendered as red & white circle, as shown in Figure 6.1. The simulated platform is a Husarion mobile robot with skid-steering model. The actor and critic networks are initialized with two neural networks having three hidden layers with 512 hidden neurons that are activated by ReLU activation function, as described in section 5.1. For training the model, stochastic policy gradient with ADAM [28] optimizer is employed to train both the actor and the critic networks. However, for the actor network, a learning rate of 10^{-4} is used whereas the critic is updated using a learning rate of 10^{-3}. Furthermore, L2 regularization is included with a coefficient of 10^{-2} when training the critic network to avoid overfitting. A discount factor of $\gamma = 0.99$ and target update, $\tau = 0.001$ is used. The initial weights and biases of the hidden neurons are chosen from a uniform distribution $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the number of inputs to the layer. The weights and biases for the output layer are taken from a uniform distribution $[-3\times10^{-3}, 3\times10^{-3}]$ to ensure that the outputs at the start of training are close to zero. The exploration noise is modeled as an Ornstein-Uhlenbeck process with parameters, $\sigma = 0.2$ and $\theta = 0.15$. The outputs of the policy are clipped to lie between the actuator limits after the addition of noise. In all experiments, a standard replay buffer that hold ups to $100,000$ transitions is used, which means that, in the worst case, the buffer can store 100 episodes since the maximum number of iterations in every episode is limited to 10^3 time-steps. The update of the weights of the networks are executed with a mini-batch of dimension 64. It is worth mentioning that a small batch size could lead the algorithm to get stuck into specific portion of the environment (local minima) whereas large batch-size can make the training period much longer since the network would be trained for more data. Thus, a good trade-off of 64 is selected so as not to elongate the training period and to ensure training on larger areas of the environment. The hyperparameters are selected based on the ones used in the original paper for the DDPG [6]. In this work, the robot is trained in a $4\times6\text{m}^2$ area with multiple obstacles. In order to simultaneously map the environment and estimate the robot pose, the ROS gmapping [3] SLAM package is used. The inputs for mapping included wheel odometry, laser range finder data and a 2D occupancy grid map representing the envi-

[1] http://gazebosim.org
[2] http://wiki.ros.org/openai_ros
[3] http://wiki.ros.org/gmapping

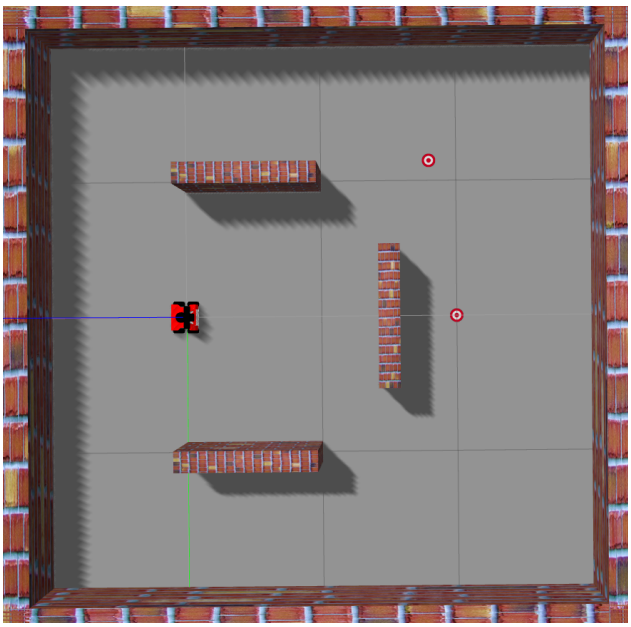


Figure 6.1: The virtual training environment simulated in gazebo. A Husarion robot is used as the platform.

ronment. The grid size of every cell is 1cm × 1cm resulting in 400 × 600 cells. A probability value is assigned to each cell based on whether it is occupied or free according to the laser sensor and odometry readings. An occupancy threshold is assigned a value of 0.65 which means if the probability value of the cell is greater than this value, this cell is occupied and, consequently, free otherwise. Besides that, to avoid higher computational complexity of the calculations, the map is only updated after certain change occurs to the probability of the posterior of the map $p(m|z_{1:t}, x_{1:t})$ within a threshold of 0.25. All hyperparameters are conveniently summarized in Table 6.1. The robot subscribes to laser readings with a scanning range from 0.2m to 2m. The position of the robot is evaluated through Rao-Blackwellized particle filter, instead of using raw odometry data, in order to calculate the polar coordinates from the target position that is fed as an input to the policy network. The agent is free to select any angular and linear velocities from a continuous space as long as they are feasible by the physical constraints of the robot. These velocity commands are directly sent to the low-level controller where the algorithm waits until

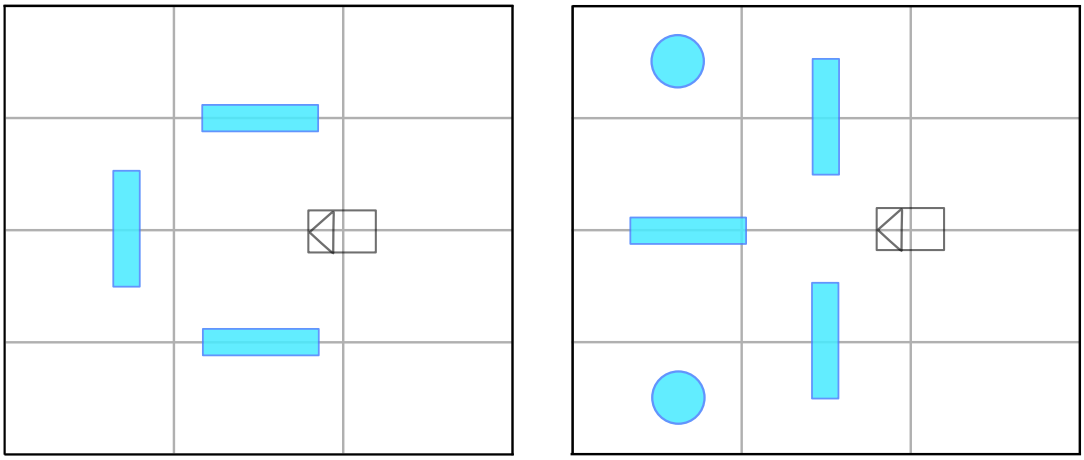
Table 6.1: Hyperparameters for DDPG experiments.

Hyperparameter	**Value**
Target network update parameter τ	$\tau = 10^{-3}$
Batch size	64
Optimizer	Adam [28]
Actor learning rate	10^{-2}
Critic learning rate	10^{-3}
Critic L2 regularization	10^{-2}
Replay buffer size	10^{5}
Discount factor γ	0.99
Training steps	10^{3}
Correlated noise σ	$\sigma = 0.2$
Correlated noise θ	$\theta = 0.15$

the command gets executed. This feedback is provided by estimating the robot's velocity from the odometry encoder's reading.
After the termination of every episode, the environment is reset and the robot returns back to its initial configuration.

6.2 Training Environments

To validate the effectiveness of the proposed approach, two different training environments are introduced on which the learning algorithms are implemented. These two virtual environments are shown in Figure 6.2.



(a) Training environment "*Env-1*" (4m x 6m). **(b)** Training environment "*Env-2*" (4m x 6m).

Figure 6.2: The agent is trained on two different training environments with a variety of cuboid and cylindrical obstacles. *Env-2* is used also for the capability of transferring the learned policy on *Env-1*.

Although, the training environments are not that much complicated, the challenging part lies in the fact that the agent has to figure out a (sub)optimal trajectory to every single target within the workspace. Here it should be pointed out, to answer the research questions, four main experiments on the virtual simulation environments are conducted.

6.2.1 Experiment 1

This first experiment is done in order to get a better intuition about in what way the proposed algorithm enhances the standard reinforcement learning by making use of the available knowledge of the environment provided by the occupancy grid mapping SLAM based technique with Rao-Blackwellized particle filters. By using this knowledge about the map which is built online by the robot while training, the reward function is shaped in an attempt to improve the convergence rate, escape local optima and reduce the number of collision samples with obstacles. This experiment will be conducted only on a single target where the main purpose of the experiment is to train the robot to navigate to this pre-defined single target within the worksapce. In that sense, a comparison is made between a reward function based on the map provided by the SLAM algorithm and a reward function when no knowledge of the map is available, This comparison is made with respect to the number of episodes required for the agent to learn an optimal policy to navigate to this single target as well as the number of collision samples encountered by the robot. The results of this experiment are presented and discussed in section 7.1.

6.2.2 Experiment 2

As explained in section 5.2, there are four different reward functions proposed. Thus, the aim of the second experiment is to make a comparison between these four reward functions and draw a conclusion about the most efficient one for the autonomous navigation problem of mobile robots. The first function is the exponential euclidean distance to the target based reward with no knowledge about the map. The second one is the difference in the distance to the desired target in two consecutive time-steps based reward and also with no knowledge about the map. Finally, the third and fourth experiments are where the map-dependent term obtained by the SLAM algorithm is added to the aforementioned two reward functions. In this experiment, to generalize the learning algorithm to any target inside the workspace, the desired target is randomly sampled from a uniform distribution at the beginning of every episode. In this way, an answer to the second and third research questions "RQ2, RQ3" can be obtained. The results of this experiment are discussed in section 7.2.
To get a better insight in understanding how the map-depended term estimated by the SLAM algorithm affects the reward function, a graphical representation of the reward without the SLAM term is visualized in Figure 6.3.

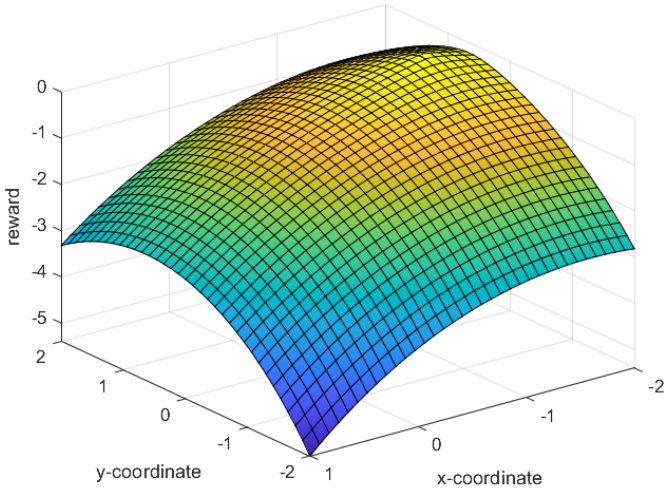
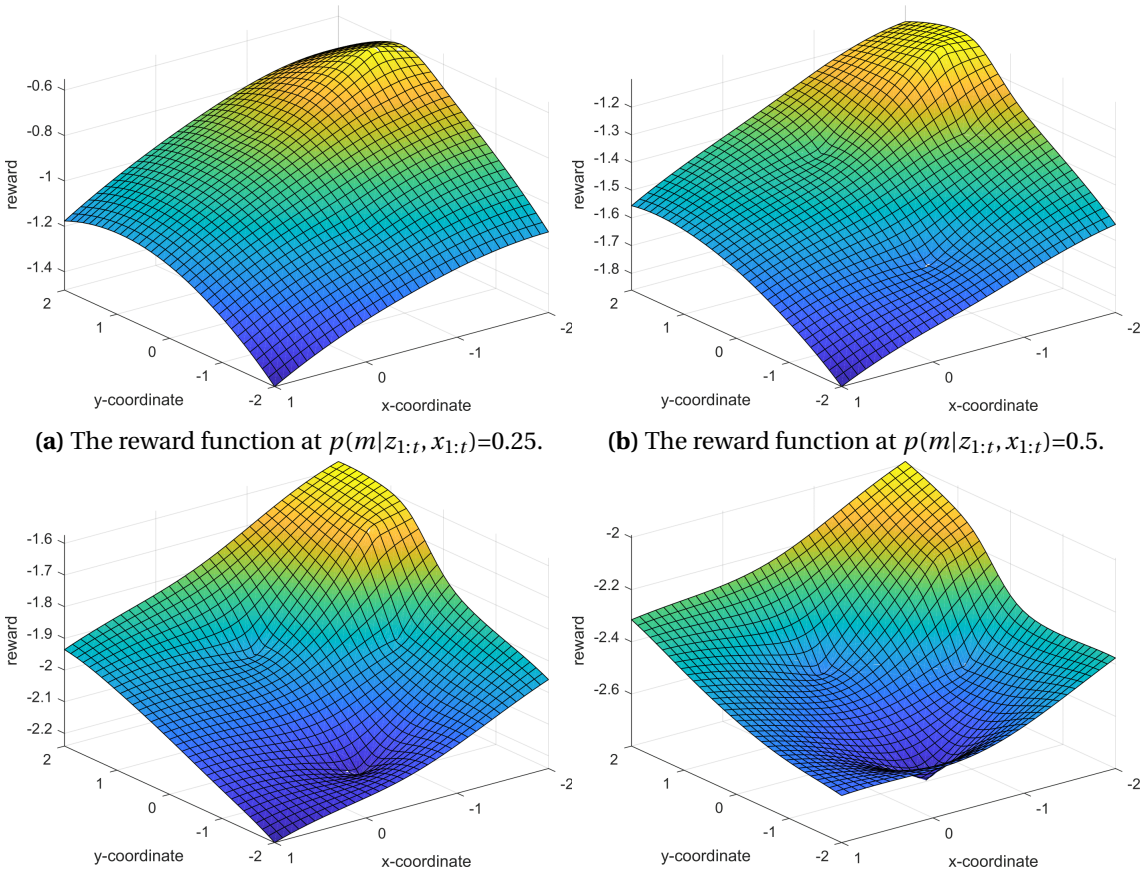


Figure 6.3: A graphical representation for the reward function given in equation 5.6.

From Figure 6.3, it is shown that the reward increases exponentially towards the desired target which is in this case at (-1.6,0.65). Thus, at this point, the algorithm has no awareness of the locations of the obstacles. Not to mention that this manifold is the same for both the first and second training environments shown in Figures 6.2a and 6.2b respectively since it takes into account only the exponential euclidean distance to the desired target.
To visualize how the reward function evolves with the change of the posterior of the map $p(m|z_{1:t}, x_{1:t})$ that increases gradually as the robot becomes more confident about the map, the reward function including the map-dependent term is plotted at four different instants as shown in Figure 6.4. In Figure 6.4, it is shown that there are three obstacles corresponding to the ones shown in the first evaluation environment depicted in Figure 6.2a. Thus, the values of the reward signals decrease exponentially as the robot comes closer to any of these obstacles. Based on that the robot not only learns the locations of the desired targets but also the static positions of the obstacles and try to avoid them by learning a good policy. Here, it should be pointed out that the manifold shown in Figures 6.3 and 6.4 is just depicted for a single target which means that it is going to change as the desired targets are changed randomly according

to a uniform distribution. At this point, it is worth mentioning that for the reward function manifold shown in Figure 6.4d, it is observed that the point at (-2,2) has a higher reward value than the target point at (-1.6,0.65). The reason for this issue is that the reward value for each point within the infinite state-space is a trade-off between how far it is from both the desired target and the surrounding obstacles. Thus, if the target point is in the vicinity of any of the obstacles, there is a high probability that other points on the manifold do have higher reward values. This problem can be easily solved by tuning the hyperparameters in equation 5.6 in such a way it is guaranteed that the desired target point has the highest reward value. However, since randomly distributed targets are considered in this study, this tuning approach is not going to work. Instead, a sparse reward is added when the robot reaches the target which would compensate for this small difference in the reward values.
Here it should be pointed out that although the manifolds described in Figures 6.3 and 6.4 are described for the exponential euclidean distance to the desired target based reward function, the same concept can be applied to the difference in the distance to the desired target in two consecutive time-steps based reward without loss of generality. However, the manifold of this plot cannot be drawn since it does not only depend on the absolute position of the target in the robot's coordinate frame at this instance, but also it takes into account the previous time-step.



(a) The reward function at $p(m|z_{1:t}, x_{1:t})$=0.25.

(b) The reward function at $p(m|z_{1:t}, x_{1:t})$=0.5.

(c) The reward function at $p(m|z_{1:t}, x_{1:t})$=0.75.

(d) The reward function at $p(m|z_{1:t}, x_{1:t})$=1.0.

Figure 6.4: The evolution of the effect of the map-dependent term on the reward function.

To depict how the map-dependent term reshapes the reward manifold of the second environment 6.2b, Figure 6.5 is plotted. As can be seen from there, the value of the reward function decreases as the robot moves towards any of the obstacles shown in Figure 6.2b.
The reason why two different environments are considered is that it is desired to make sure that the proposed algorithm can outperform the standard algorithm regardless off the obsta-

cles' shapes, orientation and the way in which they are distributed throughout the training workspace.

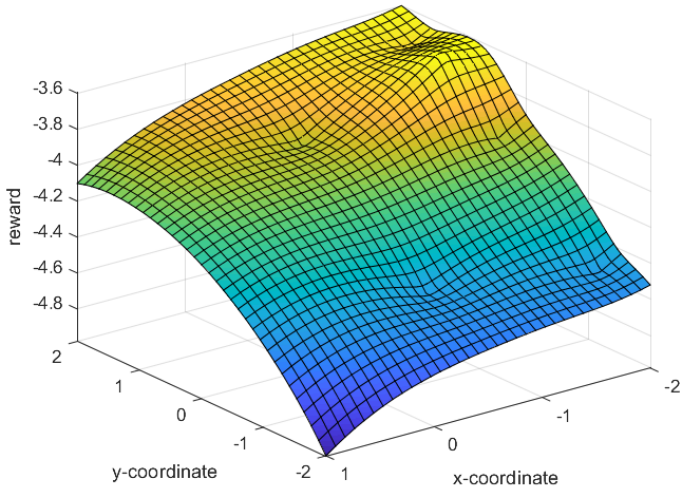


Figure 6.5: A graphical representation for the reward function on Env2.

6.2.3 Experiment 3

In this experiment, the applicability of generalizing the trained policy to other unseen virtual environments through transfer learning is discussed. In this way, the learned policy on the first environment is transferred to the second environment. The results of this experiment are explained in section 7.3.1. In this experiment, it is shown that to get better results, the initial configuration of the robot should be sampled from a uniform distribution probability at the beginning of every episode. In that sense, the learned policy does not get biased to the environment on which it is trained and thus can be generalized successfully to other unseen virtual environment. Since every experiment takes nearly 3 days on average to converge to a (sub)optimal policy due to the limited computational resources available, in this experiment, only the reward function that achieved the best performance from experiment 2 is tested for generalization to the new environment. Additionally, a comparison is made to highlight the significant improvement achieved by transferring the learned policy to new environments rather than starting training from scratch. This gives an answer to the fourth research question "RQ4".

6.2.4 Experiment 4

In the fourth experiment, it is aimed to assess how much different exploration noises affect the performance of the agent while being trained on the first environment. In that sense, two different exploration noise are considered. The first exploration noise is the traditional correlated Ornstein-Uhlenbeck action space noise which is added to the actions predicted by the policy network. On the other hand, the parameter space noise discussed in section 5.3 is assessed where the executed actions by the low-level controller are conditioned by the states. The results of this experiment is clarified in section 7.4.1 which give an answer to the fifth research question "RQ5".

6.3 Real-world Experiments

For the real-world experiments, a Clearpath Jackal differential drive mobile robot available at the Robotics and Mechatronics lab at the University of Twente is used as the mobile ground

platform. The robot subscribes to the laser range findings perceived from a YDLIDAR X4 which has a field of view (FOV) of 360° and an angular resolution of 0.33°. This lidar is selected due to its compatibility with ROS and its simple hardware interface since it only requires a micro-USB cable. The scanning range of the lidar is from 0.12m to 10m, however, to exceed the number of episodes in which a map of the environment can be built online, the maximum range is constrained to 2m when implemented on the real world. Additionally, the field of view is limited to 180° such that the proposed algorithm can be extended to low-cost range sensors with distance information from only 10 directions. Since the model is different from the Husartion robot used in the simulation platform, the experiments will be conducted again on the new robot. However, it should be pointed out that no additional tuning of the hyperparameters is required since the proposed algorithm is model-free.

Figure 6.6: Clearpath Jackal differential drive mobile robot used in the real-world experiments at University of Twente.

Obtaining real-data from robotic systems can be extremely difficult and time-consuming. For instance, for deep reinforcement learning algorithms that require huge number of samples for their convergence, e.g. 10^5 samples as in the navigation problem under study, obtaining those number of samples on a real robot is almost impossible. As mentioned previously, in the introduction section, simulation with accurate models could potentially be used to offset the cost of real-world interactions. Thus , in this section, the learned policy on the simulation platform will be firstly transferred to the real robot directly without tuning any of the hyperparameters and its performance will be evaluated. Secondly, a model that captures the mismatch between the learned trajectory on the simulation platform and the real environment is learned and then this model is used to learn a new policy iteratively. This is called a difference model [51] which is discussed in the next section.

6.3.1 Learning a Difference Model

The main idea behind this algorithm is to learn a difference model that captures the mismatch between the real system and the simulated model and use this difference model to learn a new policy. The mismatch could come from model uncertainties such as friction, measurement noise, etc.
This is done in the following way; the algorithm starts by learning an initial policy on the simulated model. This policy is expected to behave sub-optimally when applied on the real system. The learned policy is then applied on the real system to obtain data from a few trajectories $\{\tau_i\}_{i=1}^{N}$. These trajectories are composed of a tuple of the sequence of states and actions $\{s_1, a_1, s_2, a_2, ...\}$. The transitions obtained from these trajectories $\{(s_{1:T}^{\text{real},j}, a_{1:T}^{\text{real},j}, s_{2:T+1}^{\text{real},j})\}_{i=1}^{N}$ are then saved.

To obtain the same transitions on the simulated model, the state-action pairs $\{(s_{1:T}^{\text{real},j}, a_{1:T}^{\text{real},j})\}_{i=1}^{N}$ are executed on the simulated model and the next states $\{(s_{2:T}^{\text{sim},j})\}_{i=1}^{N}$ are observed. The difference model is then trained using the collected data in a *supervised learning* manner through a feed-forward deep neural network. This neural network takes as an input the current state and action and predicts the difference in next states between the simulated model and real system. The network is trained using stochastic gradient descent with samples obtained from the real system. The difference model compensates for the mismatch between the simulated model and the real system. Thus, the new policy obtained form the difference model is likely to perform better with respect to the real system. This process is repeated iteratively to enable the algorithm to converge to an optimal policy on the real system. Here it should be pointed out that the training of the new policy will have the actor and critic being bootstrapped from the previous policy and thus, the learning algorithm seeks to find an optimal policy in the neighbourhood of the previous one. The structure of the algorithm is summarized in Algorithm 9.

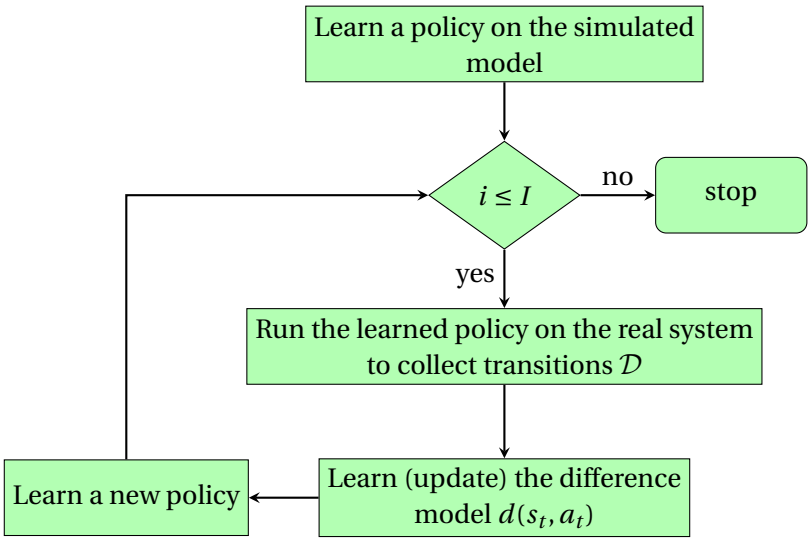


Figure 6.7: Flow chart illustrating the difference model learning.

As a matter of fact, in the ideal scenario, if $d(s_t, a_t) = s_{t+1}^{\text{real}} - s_{t+1}^{\text{sim}}$, the next state s_{t+1} would be the state corresponding to the real system.

6.3.1.1 Difference Model Experimental Setup

The training data for learning the difference model is collected by applying the learned policy on the virtual environment on the real robot. Before each model update, 2000 data-points are collected and saved. The training data is split into training and validation sets in a 3:1 ratio. The model is trained using a deep neural network having three hidden layers with 300 neurons in each layer. The activation function for each of the hidden layers is taken to be ReLU while the output layer has a linear activation function. The input layer of the deep neural network has 16 inputs (14 states and 2 actions) whereas the output has 14 states. The results of this experiment is discussed in section 7.5.

Algorithm 9 Learning a Difference Model

Initialize training data buffer $\mathcal{D}$,
Initialize critic and actor networks with random weights θ^Q and θ^π respectively,
Initialize target networks with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\pi'} \leftarrow \theta^\pi$,
Initialize a deep neural network d,
Learn a policy using the DDPG algorithm on the simulated model and obtain the final learned parameters θ_0^Q, θ_0^π.
for $i = 1 : M$ **do**
  Execute policy $\pi(s;\theta_{i-1}^\pi)$ on the real system and get the transitions $(s_t^{\text{real}}, a_t^{\text{real}}, s_{t+1}^{\text{real}})$.
  for each $(s_t^{\text{real}}, a_t^{\text{real}})$ pair **do**
    Determine $s_{t+1}^{\text{sim}} = f_{\text{ideal}}(s_t^{\text{real}}, a_t^{\text{real}})$
  end for
  Append training data to $\mathcal{D}$.
  Update d using $\mathcal{D}$.
  Initialize actor and critic networks with weights θ_{i-1}^π, θ_{i-1}^Q respectively.
  Learn a new policy and Q-function where

$$s_{t+1} \leftarrow s_{t+1}^{\text{sim}} + d(s_t, a_t).$$

end for

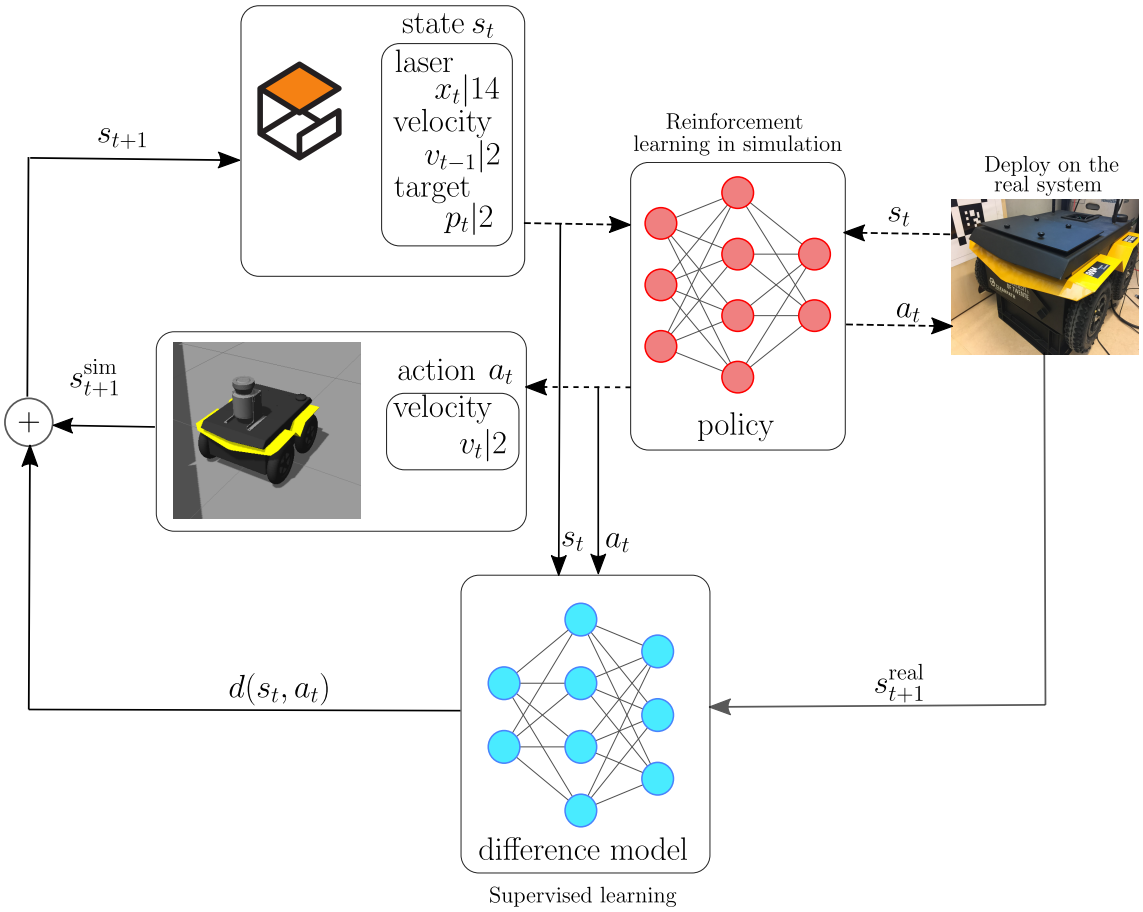


Figure 6.8: The framework for learning a difference model that captures the mismatch between the real system and the simulated model.

7 Results and Discussion

In this chapter, a comparison of the performance of both the standard deep deterministic policy gradient (DDPG) algorithm and the combined DDPG and SLAM algorithm is conducted on a large variety of settings. Different assessment criteria are introduced in evaluating the quality of the learned agent's behaviour including the number of iterations required for determining the optimal policy, the optimality of the generated trajectories and the evolution of the success ratio during both training and evaluation phases. In these experiments, the noise is modeled as an Ornstein-Uhlenbeck process except in the last experiment in which the parameter space noise is also introduced. At this point, it should be pointed out that, in the coming sections, the RL and RL&SLAM algorithms will be referred to as the standard and combined approaches respectively.

7.1 Preliminary Results for Experiment 1

To evaluate the performance of the proposed RL&SLAM combined approach, the agent is firstly trained to reach only a single desired target and then, in the next experiment, this method is enhanced and generalized such that the robot has the capability to reach any random goal inside the workspace. This training experiment is conducted on *Env-1* shown in Figure 6.2a where the target is selected to be 0.5m behind the frontal obstacle. In this experiment, the exponential euclidean distance to the target reward defined in equation 5.6 is used. The comparison is made with respect to the number of episodes required for the agent to figure out a (sub)optimal policy and the number of collision samples collected during training.
As shown in Figure 7.1, the proposed RL&SLAM combined approach starts with higher crash ratio until 250 episodes of training has elapsed. However, after that the crash ratio starts to decrease significantly to 20% after 460 episodes before it starts to increase again. The reason why the crash ratio increases again could be due to the fact that, at this point, the network is not robust enough against the exploration noise and thus the robot continues exploring in an attempt to figure out a better trajectory to the target. Then, after almost 700 episodes, the crash rate decreases dramatically from 80% to less than 4% in nearly 150 episodes. After this point, the robot manages to reach the target in most of the consecutive episodes. This is because the robot becomes more confident about the built map and it has more awareness about the surrounding obstacles.

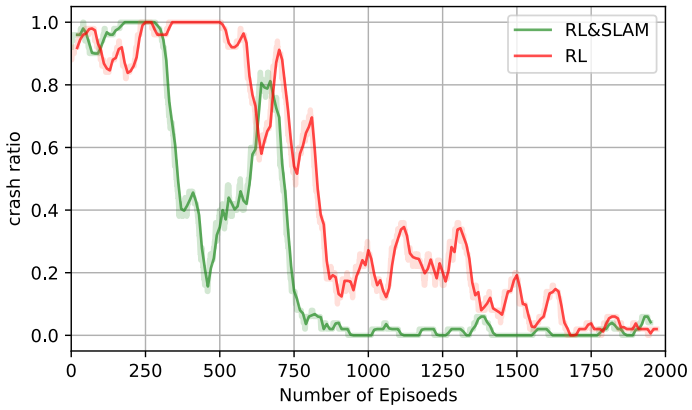


Figure 7.1: Evolution of the collision ratio with the number of training episodes. The rate in which the collision samples of the proposed approach (green) decreases is much higher than the standard approach (red).

On the other hand, the standard RL approach can achieve 6% crash ratio after almost 1700 episodes. This indicates that the number of episodes required for the robot to converge to a (sub)optimal policy is **560** and **1700** episodes for the proposed RL&SLAM combined approach and standard RL approach respectively. Furthermore, Figure 7.2 shows a **61.3**% reduction in the number of failed episodes which is recorded during training in case the proposed approach is adopted.

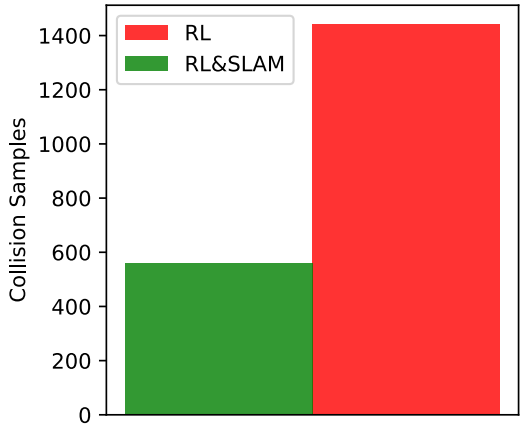


Figure 7.2: RL&SLAM combined approach has a 61.3% reduction in the collision samples.

Although this experiment proved that shaping the reward function based on the online-acquired knowledge of the environment during training improves drastically the convergence speed and decreases the number of collisions with surrounding obstacles, it is not of a great value from the practical point of view. This is due to the following reasons:

(i) The efficiency of the proposed approach has been proven for only a single target, thus there is no guarantee that it is going to generalize to other targets.

(ii) Training for a single target is not beneficial because this means that if the location of the target is changed, even a little bit, the algorithm is not going to work anymore since it gets biased to the trajectory to this single target on which it is trained and thus the whole training should start from scratch.

For these reasons, in the next section, the agent is trained to reach any randomly generated target within the workspace. To meet this purpose, in all experiments, at the beginning of each episode, the target position g is randomly chosen in such a way a collision-free path is guaranteed to exist between the robot and the desired target. These random positions of the target g are sampled from a uniform normal distribution function to ensure that the targets are well distributed all over the workspace. In this way, it is guaranteed that the robot will not get biased to some trajectories at the expense of other optimal ones.

In the next experiment, the assessment of the proposed RL&SLAM combined approach and RL standard approach on randomly distributed targets is done in two phases; training and evaluation phases. In this experiment, in the evaluation phase, the quality of the learned policy is evaluated on the same training environment.

7.2 Results for Experiment 2

In this section , the results for experiment 2 defined in 6.2.2 are discussed. To recall, it is aimed to evaluate the performance of the reward functions introduced in section 5.2 on the training environment *Env-1* where the agent is trained on randomly distributed targets.

7.2.1 Training Phase

In this section, a comparison between the performance of the standard reinforcement learning approach and the one in which the occupancy grid map is incorporated in the reward function is made. For the sake of a fair comparison, different assessment criteria are introduced to evaluate the learning performance during the training phase. These performance metrics can be summarized in the following points:

(i) The value of the estimated return to which the algorithm converges. This is obtained by firstly evaluating the mean of the Q-values in every mini-batch. Then, these mean values are appended together and can be shown as the shaded area in Figure 7.3. In order to make the results readable, the mean of these appended values are evaluated every 1000 training steps which is illustrated as the solid line in Figure 7.3.

(ii) The convergence speed which is characterized by the total number of executed actions required to converge to an optimal policy.

(iii) The success ratio which is computed by comparing the number of episodes in which the agent reaches the desired target to the total number of elapsed episodes. The success ratio is evaluated every 200 episodes.

The comparison between the standard and combined approaches is done based on both the exponential euclidean distance reward function defined in equation 5.6 and the difference in the distance to the desired target between two consecutive time-steps reward function given in equation 5.8.

7.2.1.1 Results for the Exponential Euclidean Distance Based Reward on *Env-1*

To realize the effect of the map-dependent term added to the reward function on the performance of the training phase, the Q-values for both algorithms are plotted. As noticeable from Figure 7.3, both approaches start with decreasing Q-values which is reasonable since the weights of the neural networks are randomly initialized and thus large number of collision samples are taking place. Additionally, it can be clearly noticed that the RL&SLAM combined approach has higher negative Q-values as opposed to the RL standard algorithm at the beginning of the training. The reason behind this behavior is that because of the map-dependent term added to the reward function, the agent is gaining more negative rewards when it comes closer to any of the obstacles inside the workspace. This is, for sure, in addition to the sparse negative reward given to the agent when it hits an obstacle. However, on the other hand, for the standard approach, it only gets a sparse negative reward after hitting an obstacle. The advantage of shaping the reward function based on the acquired knowledge of the environment is that it increases the awareness of the robot about the locations of the obstacles and, consequently, the number of collision samples can be decreased from 1189 to 718 samples as shown in Figure 7.4a. Because of the reduction in the number of collision samples, the Q-values of the combined approach increases much faster until it exceeds the ones obtained by the standard approach after almost 36000 training steps.. Additionally, at the end of the training period, the combined approach reaches a higher Q-value as opposed to the standard one (38.4 compared to 8.6). This means that the agent trained with the proposed approach can figure out more optimal trajectories to the desired targets. This is validated during the evaluation phase as discussed in section 7.2.2.1.
Furthermore, both algorithms were trained for 2000 episodes. However, as depicted in Figure 7.4b, the combined approach executes almost 149011 training steps compared to 229558 training steps taken by the standard approach. These results show that incorporating the knowledge of the environment obtained by the SLAM algorithm in the reward function dramatically outperforms the standard algorithm by performing much less number of actions to figure out a

(sub)optimal policy where there is **35.1**% reduction in the number of required training steps for convergence.

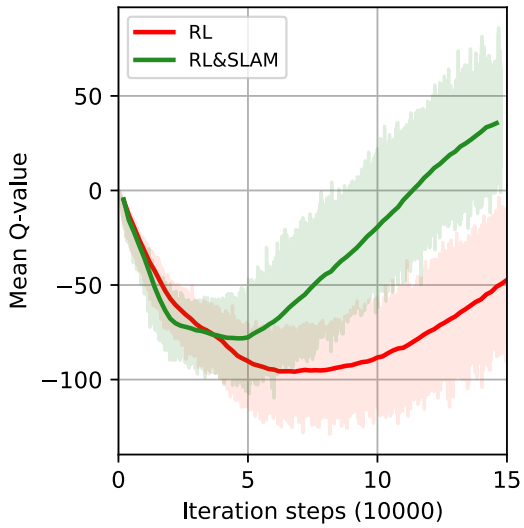
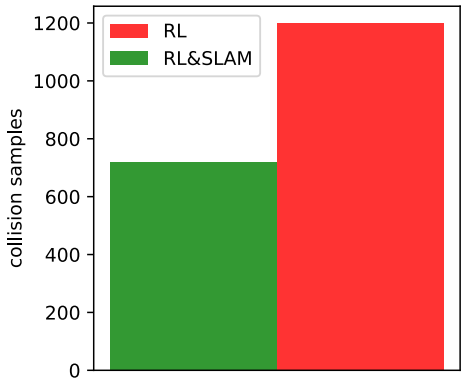


Figure 7.3: The mean Q-value of the training batch in every training step for both algorithms. This result is for the reward in equation 5.6

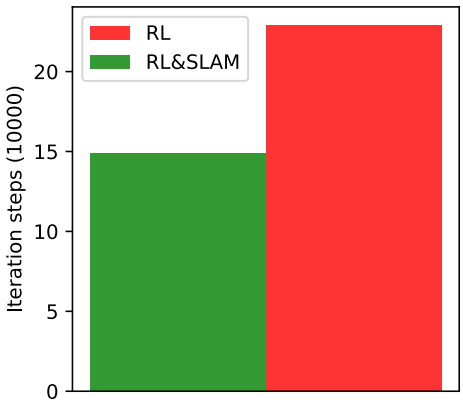
Moreover, the success ratio for both algorithms is recorded in Table 7.1 where it is evaluated every 200 episodes. From Table 7.1, it is clear that the combined approach has less collision samples and the success ratio increases much faster until it reaches 97% by the end of the training period compared to 82% achieved by the standard case where the maximum value is recorded at 84.5% after 1600 episodes. Here, it is worth mentioning that this value is reached by the combined approach after only 1400 episodes. Again, the reason is that, in the standard algorithm, there is a single sparse penalty imposed when the robot hits an obstacle. Thereby, the robot needs to hit the obstacles many times before executing a turning maneuver since it can only realize its location after hitting it due to the fact that the distance to the obstacles are represented only in the robot's states. However, on the other hand, this is not the case with the combined approach where the agent receives an exponential negative reward when getting closer to any of the obstacles. In this way, the agent does not have to hit the obstacle to detect its position and, accordingly, it can realize the optimal paths to the randomly distributed targets much faster. The evolution of the collision ratio during training can also be better visualized in Figure 7.5.

Table 7.1: Assessment of the success ratio RL and RL&SLAM combined approach during the training phase using exponential euclidean distance reward.

no. episodes	success ratio %		no. episodes	success ratio %	
	RL	RL&SLAM		RL	RL&SLAM
200	**3**%	1.5%	1200	68 %	**78** %
400	13%	**21.5**%	1400	79 %	**82** %
600	45%	45%	1600	84.5 %	**94** %
800	40%	**62**%	1800	84 %	**95**%
1000	46%	**65**%	2000	82 %	**97**%



(a) The combined RL&SLAM proposed approach achieves a 39.6% reduction in the number of collected collision samples.



(b) The combined RL&SLAM proposed approach achieves a 35.1% reduction in the number of training steps for convergence.

Figure 7.4: A comparison between the RL&SLAM combined approach (green) and the standard RL approach (red) in terms of the number iterations required before convergence and number of collision samples collected during training.

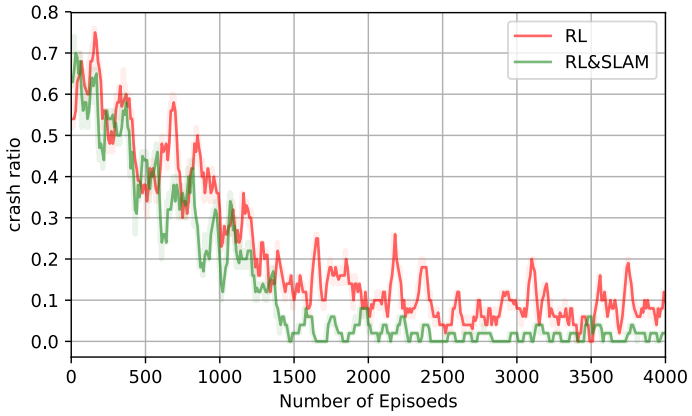


Figure 7.5: Evolution of the collision ratio with the number of training episodes. The proposed approach (green) can achieve lower collision ratio in less number of episodes compared to the standard approach (red).

7.2.1.2 Results for the Difference in the Distance to the Target in Two Consecutive Time-steps Based Reward on *Env-1*

The same comparison is made using the reward function defined in equation 5.8 and the results are shown in both Figure 7.6, 7.7 and Table 7.2. It is also observed that the collision ratio decreases in a higher rate by using the proposed approach compared to the standard one indicating that it has higher awareness about the surrounding obstacles. From Table 7.2, it is clear that the combined approach starts with higher collision samples, however, after 600 episodes, its success ratio starts to increase significantly until it achieves 97% by the end of the training phase compared to the standard algorithm that only reaches 74%.
Based on the aforementioned discussed results, it can be argued that, no matter which reward function is used, the RL&SLAM combined approach outperforms the standard RL algorithm significantly. For both reward functions, the combined approach reduces the number of collision samples by 39.6% and 22.67% respectively. In addition to that the number of iteration steps required for convergence is reduced by 35.1% and 17.4% respectively.

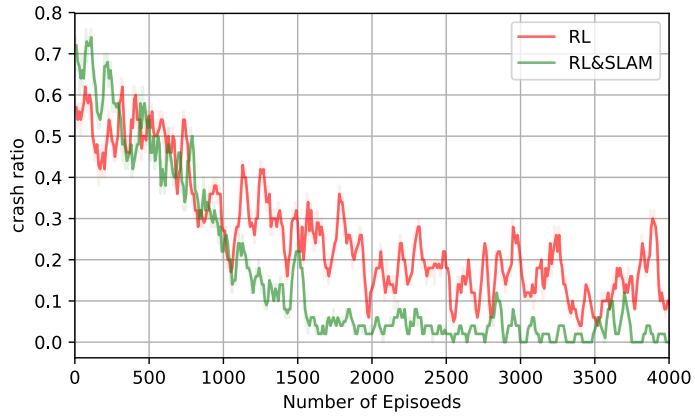
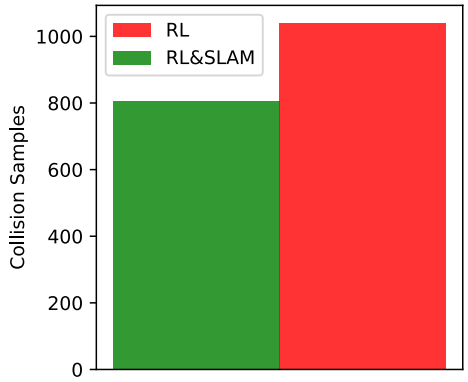
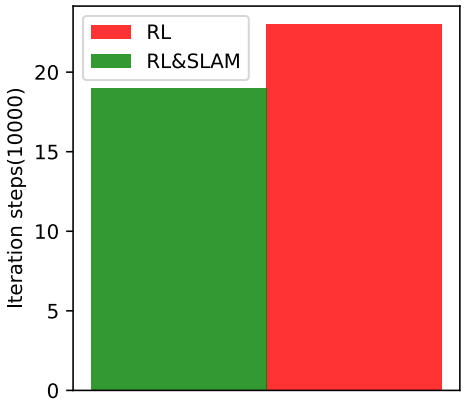


Figure 7.6: Evolution of the collision ratio with the number of training episodes. The rate in which the collision samples of the proposed approach (green) decreases is much higher than the standard approach (red).



(a) The combined RL&SLAM proposed approach has a 22.67% reduction in the number of collision samples.



(b) The combined RL&SLAM proposed approach achieves a 17.4% reduction in the number of training steps for convergence.

Figure 7.7: A comparison between the RL&SLAM combined approach (green) and the standard RL approach (red) in terms of the number iterations required before convergence and number of collision samples collected during training based on the reward defined in 5.8.

Table 7.2: Assessment of the success ratio RL and RL&SLAM combined approach during the training phase using difference in the distance to the target between two consecutive time-steps.

no. episodes	success ratio %		no. episodes	success ratio %	
	RL	RL&SLAM		RL	RL&SLAM
200	**19**%	2%	1200	61 %	**76.5** %
400	**30.5**%	22%	1400	68 %	**85.5** %
600	24%	**25**%	1600	59.5 %	**87** %
800	35%	**48.5**%	1800	70 %	**95.5**%
1000	38.5%	**58.5**%	2000	74 %	**97**%

At this point, it can also be inferred that, the RL&SLAM combined approach with exponential euclidean distance to the desired targets reward function achieves the best results with respect to convergence rate and the evolution of the success ratio compared to the difference in the

distance to the target in two consecutive time-steps reward. This is the case due to two different reasons.

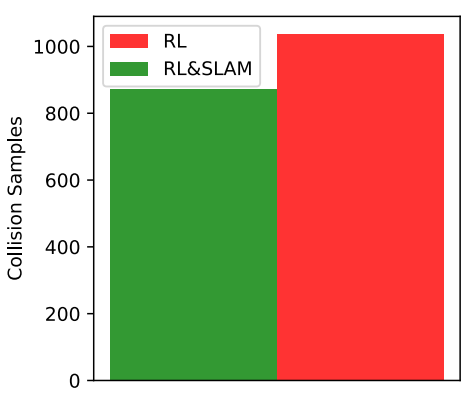
(i) Firstly, the second reward function does not differentiate between getting too closer to the target or being far away. In other words, if the robot is moving towards the target it will get a positive reward, however this reward will be the same whether, for example, it is 0.5m or 3m away from the target. However, in case of the exponential euclidean distance reward, the reward value will differ substantially based on how far the robot is from the target.

(ii) The second reason is that since the map-dependent term added to the reward function also depends on the exponential euclidean distance to the obstacle, it is consistent with the reward defined in 5.6, and thus no tuning for the hyperparameters are required in this case in contrast to the reward defined in 5.8.

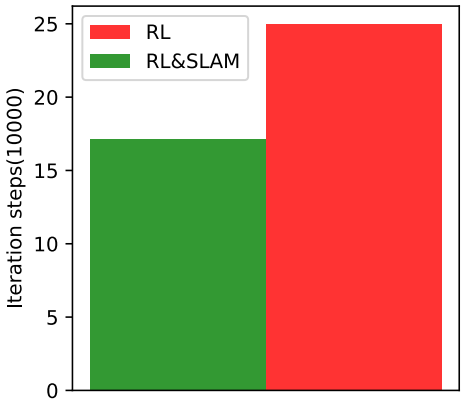
Thereby, in the coming sections, the reward based on the exponential euclidean distance to the target is considered while discarding the other reward function.

7.2.1.3 Results for Training on *Env-2* using Exponential Euclidean Distance Based Reward

Since the reward function defined in 5.6 achieved the best results as concluded form section 7.2.1, it is going to be applied for the experiments on *Env-2*. To validate that the proposed approach still achieves better results on new environments, a policy is trained from scratch on *Env-2* using also both combined RL&SLAM combined approach and RL standard approach. Since it has been already discussed in detail why the proposed approach outperforms the standard RL approach, in this section, only the results will be displayed where the logic behind achieving better results is the same as discussed in previous sections.



(a) The combined RL&SLAM proposed approach has a 15.8% reduction in the number of collision samples.



(b) The combined RL&SLAM proposed approach achieves a 31.3% reduction in the number of training steps for convergence.

Figure 7.8: A comparison between the RL&SLAM combined approach (green) and the standard RL approach (red) in terms of the number iterations required before convergence and number of collision samples collected during training based on the reward defined in 5.11 on *Env*-2

As shown in Figure 7.8, the combined approach sill achieves much better results on the new environment with different obstacle shapes and orientations. In terms of the number of collision samples, the combined approach achieved a 15.8% reduction compared to the standard one. Not to mention, like the case on *Env-2*, the proposed approach converges to the optimal policy in less number of training steps (171463 compared to 249648 steps). It is also clear that the training on *Env-2* has higher collision samples compared to the training on *Env-1* which

is reasonable since *Env2* has more obstacles. The success ratio during the training phase on *Env-2* is shown in Table 7.3.

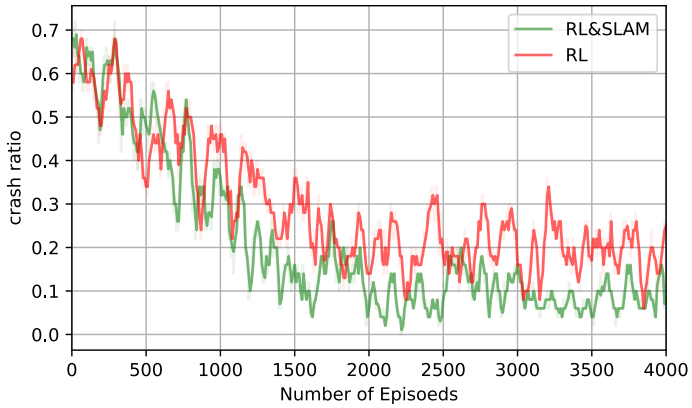


Figure 7.9: Evolution of the collision ratio with the number of training episodes. The proposed approach (green) can achieve lower collision ratio in less number of episodes compared to the standard approach (red) on *Env*-2.

Table 7.3: Assessment of the success ratio RL and RL&SLAM combined approach during the training phase using exponential euclidean distance reward on *Env-2*

no. episodes	success ratio %		no. episodes	success ratio %	
	RL	RL&SLAM		RL	RL&SLAM
200	3.5%	**3.5**%	1200	56 %	**68** %
400	**13**%	11.5%	1400	59 %	**79** %
600	**45**%	28.5%	1600	65 %	**84.5** %
800	40%	**54**%	1800	76 %	**88**%
1000	45.5%	**58.5**%	2000	78 %	**88**%

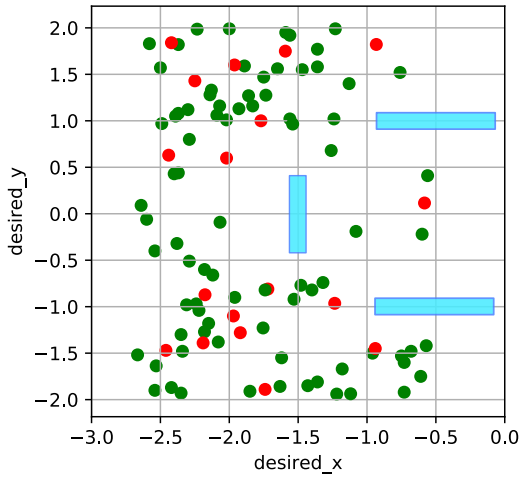
From Table 7.3, it is shown that the combined approach has achieved a **10**% higher success ratio during training on *Env-2*.
Thus, it can be concluded that, after training both algorithms on two different environments with different reward function definitions, it has been proven that the proposed RL&SLAM approach with exponential euclidean distance based reward has achieved the best results during the training phase. The comparison was done in terms of the number of iteration steps required by the algorithms for convergence, the number of collision samples, the Q-value to which the algorithm converges and the success ratio. With respect to all these performance metrics, the proposed approach significantly outperformed the standard method.
In the next section, the quality of the generated trajectories is compared along with the number of actions required for both approaches to reach pre-determined targets during the evaluation phase. Firstly, the learned policy is tested on the same training environment and then it will be generalized to an unseen virtual environment.

7.2.2 Evaluation Phase

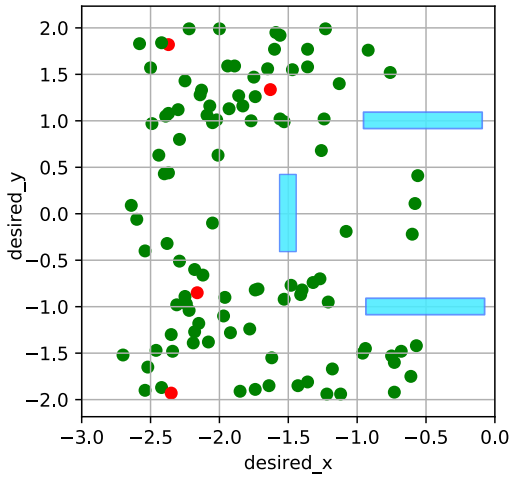
In this section, the learned policies obtained from both the proposed RL&SLAM combined approach and the standard RL approach with the reward function defined in 5.6 on *Env-1* are evaluated on the same training environment.

7.2.2.1 Evaluation Phase on the same Environment

After each training phase, the trained model for both approaches was evaluated on the same set of 100 random targets to guarantee a fair comparison and the percentage of the successfully solved episodes was monitored. An episode is considered to be successful, when the robot reaches the target without either hitting any of the obstacles or exceeding the maximum number of allowed iterations per episode which is set to 400 steps during the evaluation phase. Additionally, different evaluation metrics are introduced in this section to evaluate the quality of the learned policy for both the standard reinforcement learning algorithm and the one combined with SLAM. Here it should be pointed out that, during the evaluation phase, the Ornstein-Uhlenbeck random noise added to the action space for exploration purposes is disabled. Figure 7.10 illustrates that the combined RL&SLAM approach achieved a higher success ratio, 96%, compared to the standard approach that reaches only 82%. Furthermore, to investigate the effect of incorporating knowledge of the environment into the training process on the optimality of the generated trajectories, the average number of actions required to reach these targets is recorded in Table 7.4.



(a) The standard RL approach reaches an 82 % success ratio with respect to 100 randomly generated targets during the evaluation phase.



(b) The combined RL&SLAM approach achieves a success ratio of 96% with respect to 100 randomly generated targets during the evaluation phase.

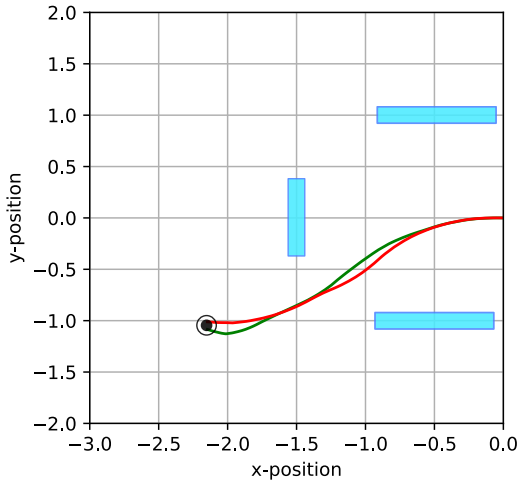
Figure 7.10: A comparison between the RL&SLAM combined approach (right) and the standard RL approach (left) in terms of the success ratio during evaluation phase. The green circles represent the reached target whereas the red ones indicate that the robot either hits an obstacle or exceeds the maximum number of allowable iteration steps per episode.

Table 7.4: Assessment of RL and RL&SLAM combined approach on the same training environment.

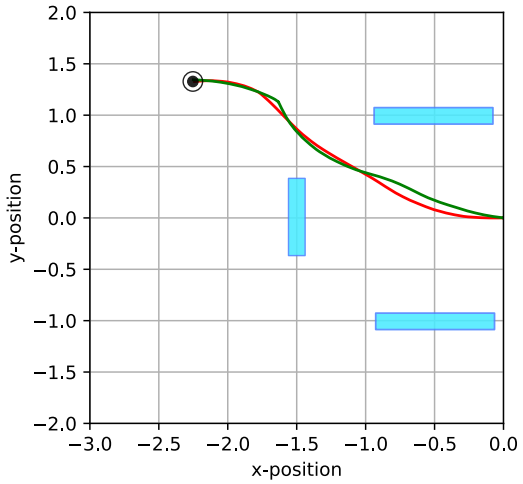
	success ratio %	number of actions (mean ± std)
RL	82 %	71.7 ± 75.34
RL&SLAM	**96** %	**52.86** ± **54.73**

As reported in Table 7.4, it can be seen that the combined RL&SLAM not only achieves a higher success ratio but also tends to execute much less number of actions, on average, to reach the desired targets since there is **26.7**% reduction in the number of executed actions. This can be speculated as, because of the dense reward, the combined approach has more awareness about the locations of the obstacles and thus, it can figure out more optimal trajectories to the desired targets. However, for the standard approach, since the agent learns the positions of the

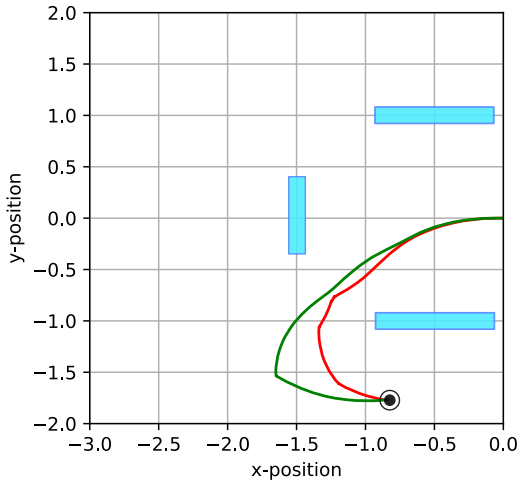
obstacles only when it comes into its vicinity due to the sparse reward, there is nothing that motivates the robot to move away from the obstacles and thus it cannot escape local optima of the policy resulting in a higher number of actions to reach the targets.
To get a better insight regarding the difference in the generated trajectories, the generated trajectories for four different desired targets can be visualized in Figure 7.11. As shown in Figure 7.11, the travelled distance from the initial pose of the robot to the desired target is less in case of the combined approach resulting in more optimal trajectories.



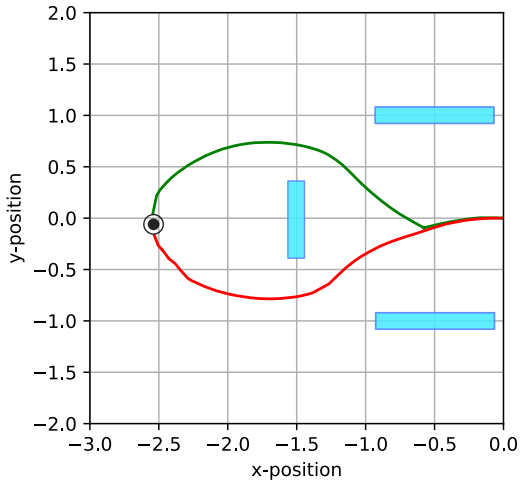
(a) The total distance travelled by the combined approach is 2.47m compared to 2.53m by the standard approach.



(b) The total distance travelled by the combined approach is 2.19m compared to 2.58m by the standard approach.



(c) The total distance travelled by the combined approach is 2.88m compared to 3.32m by the standard approach.



(d) The total distance travelled by the combined approach is 3.24m compared to 3.55m by the standard approach.

Figure 7.11: A comparison of the generated trajectories by both the RL&SLAM combined approach (visualized in red) and the RL standard approach (visualized in green). The target is depicted by the black circle.

7.3 Results for Experiment 3

7.3.1 A generalization of the Learned Policy to Unseen Virtual Environments

In this section, the generalization of the learned policy to unseen virtual environments is discussed in more details. It has been observed that if the learned policy, based on the settings discussed in the previous sections, is transferred directly to an unseen environment different from the one it was trained on, it will have a poor performance with a success ratio of 34%.

It can be speculated that the reason for this behaviour is because after the termination of every episode, the robot returns back to its initial pose. As a result, the robot gets biased to the environment on which it is trained since every episode it perceives the same structure of the environment. Thus, in order to tackle this issue, every time the episode terminates, the robot starts the new episode with a random pose. In this way, it is guaranteed that the robot does not get biased to the structure of the environment it is trained on since every time it is going to perceive a different perspective of the environment and consequently its new policy is trained using different variety of input states perceived by the lidar.
After doing so, the learned policy on *Env-1* is transferred to and tested on *Env-2* which was unseen to the robot during training. By applying this adjustment, the new learned policy is transferred directly to the new environment and a success ratio of 74% was recorded which is much better than the previous policy. Moreover, when the agent gets trained on the new environment for just 300 episodes, it can achieve 89% success ratio. This can be seen as a major advantage over starting training the policy network from scratch when switching from one environment to another. The randomly distributed targets on the evaluation environment are depicted in Figure 7.12 where the green circles represent the reached targets and the red ones indicate that the robot either crashes or exceeds the maximum number of allowable training steps (400 steps during evaluation phase) on its way to these targets.

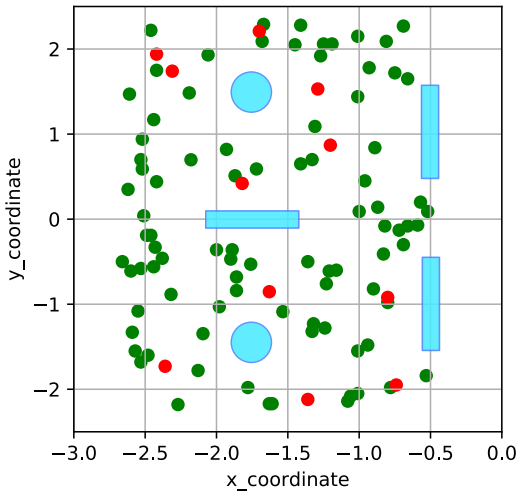
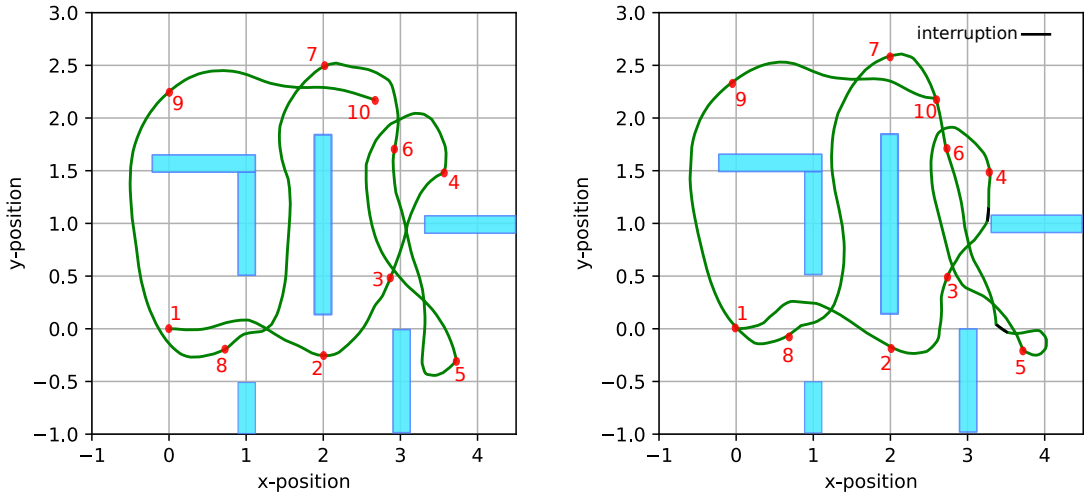


Figure 7.12: The trained policy on the training environment achieved 89% success ratio after being transferred and trained for 300 episodes on the evaluation environment.

To get a better insight in how much improvement has been made by transferring the policy learned on *Env-1* to *Env-2*, the obtained results are compared with the results achieved by training the robot from scratch on *Env-2* that has been already discussed in section 7.2.1.3. From Table 7.3, it is recorded that a 74% success ratio is obtained after nearly 1300 training episodes from scratch where this same success ratio was achieved by transferring the learned policy directly from *Env-1* to *Env-2* indicating that the proposed algorithm is easily adaptable to changing environments and makes tremendous improvement in the number of required training episodes for convergence.
Additionally, since the robot starts every episode at different poses, it is not necessary to return back to (0,0) to navigate to the desired target.In other words, the learning algorithm is independent of the initial pose of the robot and the position of the desired target. To validate this argument, 10 target positions are set for the motion planner. In that sense, the motion planner should navigate the robot to the target positions along the sequence number. The trajectory taking by the robot while navigating from one point to another is shown in Figure 7.13a.

As shown in Figure 7.13a, the robot manages to complete the navigation task even in narrow places. On the other hand, the standard RL approach was not able to complete the navigation task and the simulation had to be interrupted at the black lines depicted in Figure 7.13b since the robot collides with the obstacles close to targets 4 and 5. Here it should be pointed out that the target points in Figure 7.13b are slightly different from the ones in Figure 7.13a. This is due to the tolerance added in the vicinity of the desired target which is 0.2m. Moreover, there is 7.3% reduction in the total travelled distance achieved by using the proposed RL&SLAM combined approach. This indicates that adopting the proposed approach motivates the robot to learn more efficient planning strategies.



(a) Proposed RL&SLAM combined approach. The total travelled distance is 29.85m

(b) RL standard approach. The total travelled distance is 32.16m

Figure 7.13: Trajectory tracking in a virtual test environment. The motion planner based on the standard RL approach was not able to finish the navigation task.

7.4 Results for Experiment 4

7.4.1 Evaluation of Different Exploration Noise in Training Phase

In this section, different exploration noises discussed in section 5.3 are applied to the learning agent and the quality of training is assessed. For that purpose, two different exploration methods are considered:

(i) Exploration with correlated additive Gaussian noise; in this configuration, correlated additive Gaussian noise is included using Ornstein-Uhlenbeck process before executing an action: $a_t = \pi(s) + \mathrm{OU}(\sigma, \theta)$. A standard deviation of $\sigma = 0.2$ and $\theta = 0.15$ are considered in this case.

(ii) Exploration with parameter space noise; in this configuration, adaptive parameter space noise is used in the same way as described in section 5.3: $a_t = \tilde{\pi}(s)$ where the adaption interval is chosen to be $T_{\text{adapt}} = 50$.

The same performance evaluation metrics introduced in the previous sections are used here. Thus, the collision rate plot for the trained agent with parameter space and action space noise is illustrated in Figure 7.14. As expected, from this Figure, it is clear that the rate in which the collision ratio decreases for the agent with parameter space noise is higher than the one with action noise. This is due to the fact that the noise is conditioned on the state and not just randomly added to the output actions. By doing so, the algorithm can figure out the (sub)optimal paths to certain targets faster. However, on the other hand, with action space noise, even if the algo-

rithm figures out a (sub)optimal trajectory to a certain target in one iteration, when this same target is repeated in another iteration, there is no guarantee that an optimal trajectory can be determined in this case since a random noise is added to the actions predicted by the policy. This, of course, occurs at the early iteration steps, before an optimal policy is learned. However, once the algorithm converges to an optimal policy, both algorithms can achieve nearly the same success ratio which is shown in Table 7.5. Hence, it can be concluded that the parameter space noise can converge to the optimal policy with less number of collision samples compared to the action space noise and also less training steps. This can be shown clearly in Figure 7.15 where there is a **32.03**% reduction in the collision samples. At the same time, at the end of the training phase, both algorithms can achieve almost the same success ratio.

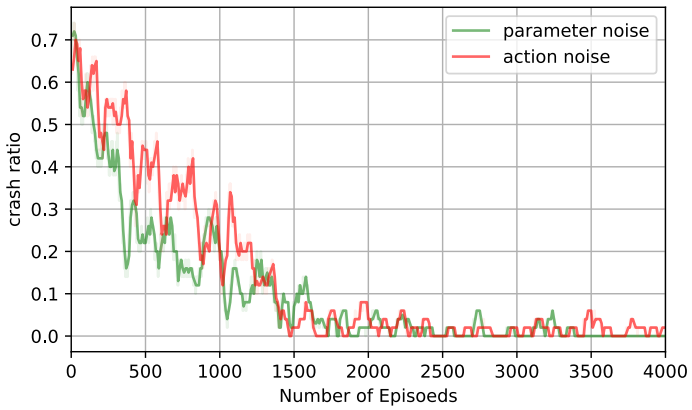


Figure 7.14: Evolution of the collision ratio with the number of training episodes. The rate in which the collision samples of an agent with parameter space noise (green) decreases is higher than the one with action space noise (red). Both agents converge to the same crash ratio at the end of the training

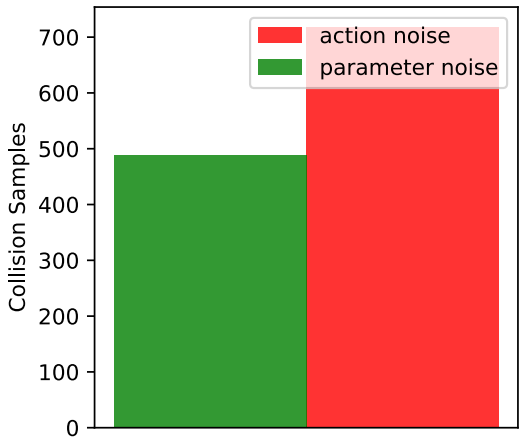


Figure 7.15: Exploration using parameter space noise has a 32.03% reduction in the collision samples.

Table 7.5: Assessment of the success ratio using parameter space noise and correlated OU noise during the training phase using exponential euclidean distance reward.

no. episodes	success ratio %		no. episodes	success ratio %	
	parameter	action		parameter	action
200	**14**%	1.5%	1200	**86** %	78 %
400	**56**%	21.5%	1400	**86** %	82 %
600	**72.5**%	45%	1600	92.5 %	**94** %
800	**77**%	62%	1800	**95.5** %	95%
1000	**78.5**%	65%	2000	**98** %	97%

7.5 Results for Real-world Experiments

In this section, it is aimed to transfer the learned policy on the virtual environment to the real robot. Different target positions are set for the robot and a comparison is made between the travelled trajectory on the real and virtual environments. At this point, it should be pointed out that the learned policy is transferred directly without any tuning of the hyperparameters or further training on the real robot. The efficiency of transferring the learned policy has been validated on different targets where four of them are depicted in Figure 7.16. Figure 7.16 shows the generated trajectories for both the simulated robot and the real robot after transferring the learned policy on the virtual environment. It can be interpreted that the learned policy is transferred successfully in the sense that the robot manages to avoid the obstacles and steers itself towards the desired target. On the other hand, it can also be noticed that the real robot does not follow exactly the same trajectory learned on the virtual environment. This could be due to different reasons including model inaccuracies in mass, inertia or lengths of links, model uncertainties such as friction and measurement noise. However, this cannot be considered as a drawback since the learned policy is robust enough to adapt itself to these changes.
At this point, in case it is required to follow exactly the same trajectory as the one learned on the virtual environment, it is still possible to learn a difference model that captures the mismatch between the real system and the simulated model as discussed in section 6.3.1. The evolution of the loss during training on both the training and validation sets is shown in Figure 7.17.

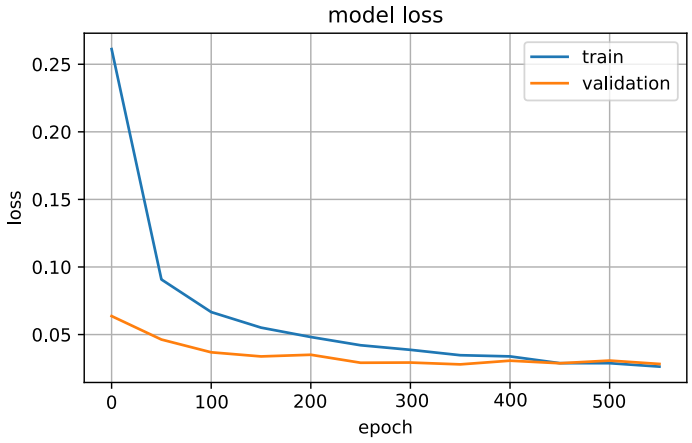
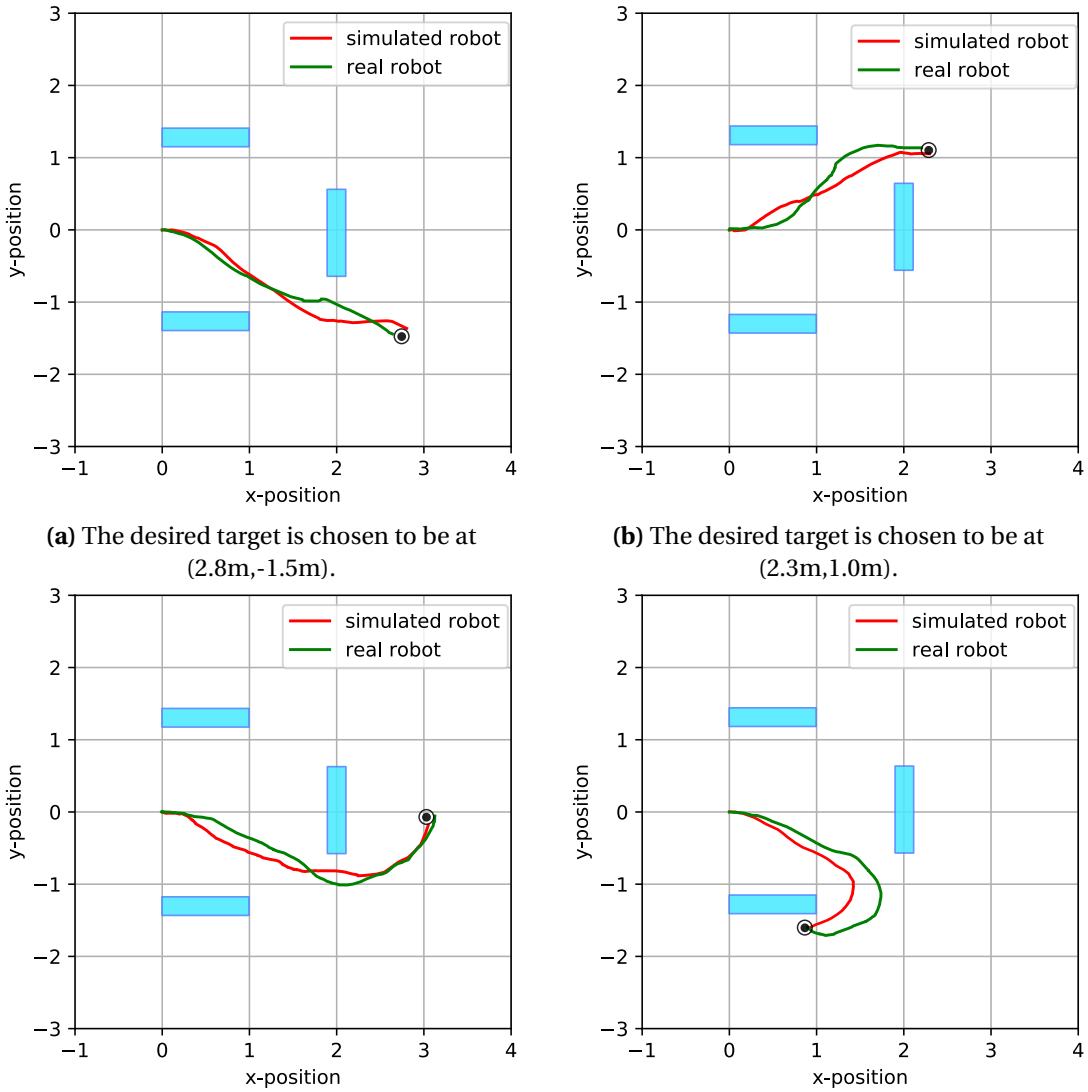


Figure 7.17: The evolution of the loss function during training on train and validation data.

After the difference model is trained, it is used to compensate for the mismatch between the real system and the simulated model during state transition and a new policy is obtained. Based on that the new policy is applied to the real robot and the travelled trajectories are recorded in Figure 7.18. As can be noticed from Figure 7.18, the generated trajectories by the



(a) The desired target is chosen to be at (2.8m,-1.5m).

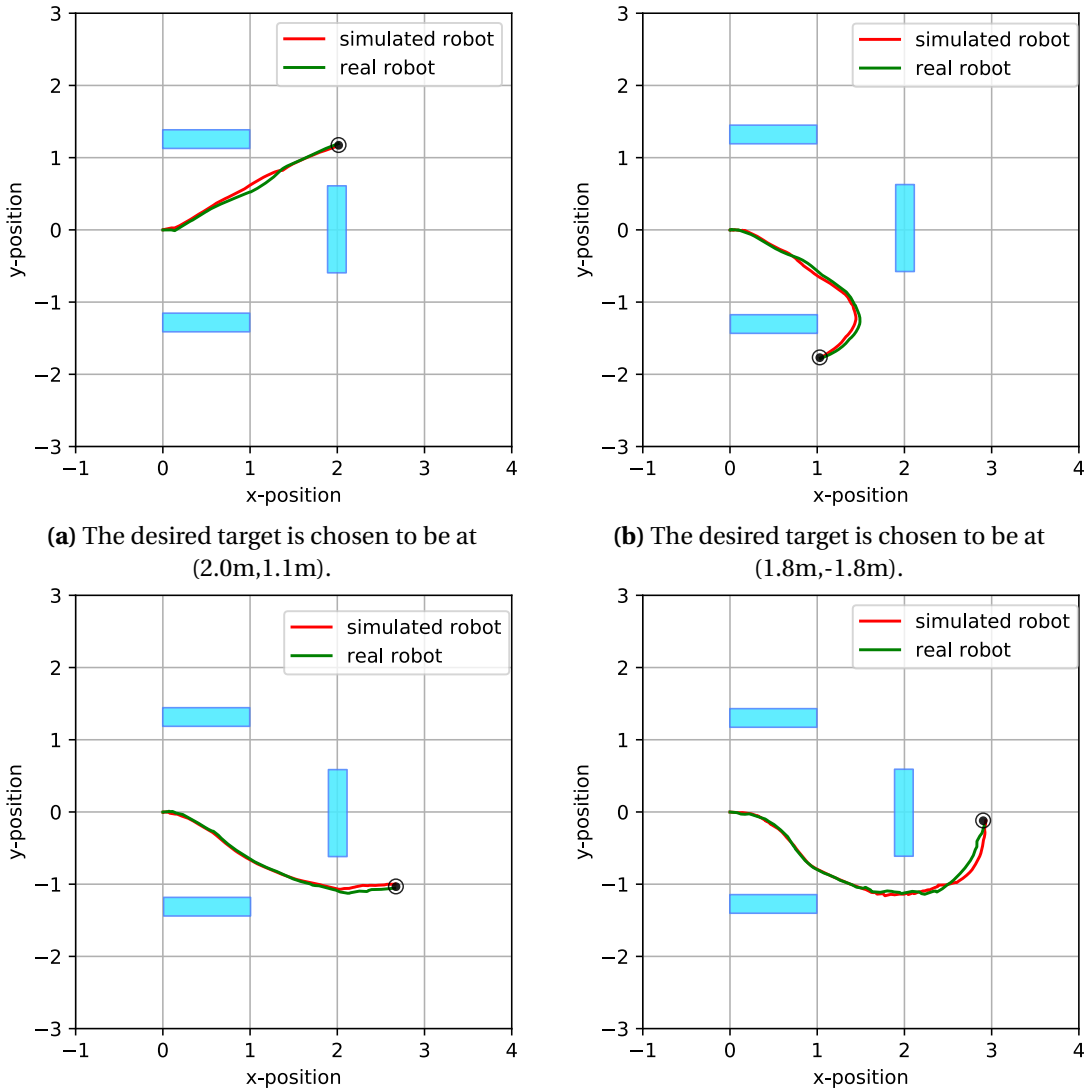
(b) The desired target is chosen to be at (2.3m,1.0m).

(c) The desired target is chosen to be at (3.0m,0.0m).

(d) The desired target is chosen to be at (0.9m,-1.6m).

Figure 7.16: A comparison of the generated trajectories by the simulated robot in gazebo (visualized in red) and the real robot after transferring the learned policy (visualized in green). The target is depicted by the black circle.

real robot are almost the same as the one generated on the virtual environment where a significant improvement is realized compared to the results shown in Figure 7.16. The number of iterative policy updates required to achieve these results is three.
Here it should be pointed out that since the model used for the robot in the simulation is a ROS compatible robot which means that the model of the robot already exists by the manufacturer, there is no much difference in the model between the simulated model and the real one. This is one of the reasons why the learned policy on the virtual environment worked pretty well after being transferred to the real system even before learning a difference model. However, for this problem, the difference model is only used to generate real-time trajectories that mimic the same trajectories obtained on the virtual environment. In cases where the developer has to create a model of the robot on the virtual environment, there is a high probability that model inaccuracies would occur. In these situations, learning a difference model would be crucial to employ the learned policy on the real robot instead of starting the training from scratch on real-time which may result in the drawbacks that were described beforehand in section 1.2.



(a) The desired target is chosen to be at (2.0m,1.1m).

(b) The desired target is chosen to be at (1.8m,-1.8m).

(c) The desired target is chosen to be at (2.8m,-1.0m).

(d) The desired target is chosen to be at (3.0m,0.0m).

Figure 7.18: A comparison of the generated trajectories by the simulated robot in gazebo (visualized in red) and the real robot after transferring the learned policy (visualized in green). The target is depicted by the black circle.

7.6 Critical Appraisal

In this section, a critical appraisal is reported focusing on the points of strength and weakness of the proposed navigation approach.

Strengths

(i) **No accurate model is required for the environment**: Since the proposed learning approach is model-free, there is no need for an accurate model of the environment where the optimal policy is learned through interactions between the robot and the environment.

(ii) **Adaptability**: The implemented algorithm is easily adaptable to unseen environments that it has never trained on before. In other words, if transfer learning is employed, it allows a robot to learn similar environment layout quickly.

(iii) **Low-cost solution**: The proposed approach only requires odometry data that can be acquired from rotary encoders and sparse laser data. Although, a lidar has been used in this research for the perception of the environment, it can be extended to low-cost 1D range sensors with distance information from only 20 directions.

(iv) **High repeatability**: Some of the experiments presented in this thesis have been conducted more than once to check the repeatability of the proposed approach. It has been observed that the overall performance of the learned policy does not change from one experiment to another. As a matter of fact, not exact same results are obtained because of the stochastic behavioral policy used for exploration.

(v) **Reliability**: An advantage of using a lidar over a visual system for perceiving the environment is that the learned algorithm does not depend on the light conditions which makes it reliable for navigation tasks in ballast tanks and pipes.

(vi) **Real-world Experiment**: The learned policy on the virtual environment can be transferred successfully to a real robot without the need for tuning any of the hyperparameters or any extra training on the real robot. This is can be considered as a major advantage due to the fact that obtaining real data from robotic systems is extremely difficult and time consuming. Moreover, it could result in serious damages to real robots since the training depend on a trial-and-error process. Transferring the policy to the real system was possible since lidar data was used to perceive the environment. This is in contrast to visual systems where significant discrepancies do exist between rendered color images and real camera readings.

(vii) **Continuous control navigation**: Since the proposed learning approach is policy search based, the motion planner can output continuous linear and angular velocities directly resulting in smooth maneuvers of the robot. Additionally, since a policy gradient algorithm is used to update the parameters of the policy network, the parameters of the policy are slightly modified which guarantees smooth transition between states in contrast to value-based methods where large jumps between estimated polices are possible.

(viii) **Less number of required samples**: The advantage of learning a deterministic policy is that he mapping from states to actions becomes fixed and accordingly there is no need to integrate over the whole action space. Furthermore, since it is an off-policy algorithm, it is possible to learn a deterministic policy while the agent is following a stochastic behavioral policy to guarantee adequate exploration of the environment.

(ix) **Reduction in training steps**: Shaping the reward function based on the knowledge of the environment that robot acquires during training decreases the number of collision samples significantly and thus the training steps required for convergence are decreased as well.

Weaknesses

(i) **Long training time**: One of the drawbacks of learning continuous control actions for systems with high-dimensional state and action spaces is the training time. For the navigation problem, it takes from 3 to 4 days to learn a (sub)optimal policy. This imposed a restriction on running different preliminary experiments to tune the hyperparameters.

(ii) **Pose correction**: It has been observed that if the robot missed the target with few centimeters, it becomes difficult for it to correct its pose and return back to it. The reason for this behavior could be due to restricting the backward action. Thus, it is challenging for the robot to rotate on the spot especially in narrow places.

(iii) **Simulation crashing**: Although gazebo is a powerful simulation platform for robotics applications, it is not learning-friendly since it crashes a lot during training. Thus, it can be recommended to consider other robotic simulation platform such as v-rep which is used a lot in robotics learning literature.

8 Conclusion

This research started with the following question

" How the navigation problem of non-holonomic mobile robots can be formulated as a reinforcement learning problem that could be solved by using deep deterministic policy gradient (DDPG) actor-critic algorithm?"

This is achieved by designing a motion planner that takes 10-dimensional sparse laser range findings, the target position relative to the mobile robot coordinate frame and the last executed action as inputs where the proposed motion planner can output continuous linear and angular velocities directly. Then, from this question, two other research question arose. The first one is

"How to determine a proper reward function that can reflect the quality of the learned trajectory?"

This is done by using the exponential euclidean distance to the target as the basic reward. Then this reward was shaped based on a probabilistic occupancy grid-map that the robot builds during training to increase its awareness about the obstacles inside the environment. This leaded to the third research question which is

"To what extent does the incorporation of the partial map obtained about the environment via the SLAM algorithm help the learning algorithm?"

To answer this question, two virtual simulation environments were constructed using Gazebo simulation platform and the robot was trained on these environments using both the standard RL algorithm and the proposed combined RL&SLAM based approach. Accordingly, the results were assessed during both the training phase and evaluation phase. For the first environment, during the training phase, it has been observed that the number of executed actions required to converge to a (sub)optimal policy is reduced dramatically by about 35.1% by adopting the proposed approach. The reason behind this reduction is that the robot learned the positions of the obstacles much faster and thus the number of collision samples were also reduced significantly by 21.19% resulting in a convergence to a higher Q-value. This result is interesting because the added map-depended term to the reward function is a negative dense reward. Thus, logically, the Q-values obtained by the combined RL&SLAM approach should be less than the ones achieved by the standard RL algorithm. This means that the proposed approach has a higher success ratio than the standard one and also the learned policy can figure out more optimal trajectories to the desired targets using less number of control actions. This claim was also validated by evaluating both algorithms on the same environment using the same set of 100 random targets. It was shown that the proposed approach achieved a success ratio of 96% compared to 82% achieved by the standard algorithm. Moreover, the number of executed actions to reach these targets were dropped by 35.3% indicating that the learned policy by the proposed approach is more optimal. To validate that the proposed approach still has better results on different environments, a second training environment was constructed with different obstacles' shapes and orientations. It was also noticed that the proposed approach converges to higher Q-values and has a 15.9% reduction in the number of collision samples as opposed to the standard approach.
The fourth research question was

"How applicable is it to generalize the learned policy on one environment to another environment through transfer learning?"

To check the applicability of the generalization of the learned policy to different environments, the learned policy on the first environment was transferred to the second environment. The preliminary results show that the transferred policy had a poor performance and achieved only 34% success ratio after being transferred. It was speculated that the cause of this behaviour is that after the termination of each episode, the robot returns back to its initial pose from where it starts the new episode. Thereby, the robot gets biased to the environment on which it is trained since it perceives the same structure of the environment at the beginning of every episode. Thus, it was proposed that the initial pose of the robot is randomized according to a random normal distribution. In this way, it is guaranteed that the robot will get trained on a large variety of input states perceived by the lidar. The same experiment was repeated on the first environment and the learned policy was transferred again to the second environment. As expected, there was a significant improvement since the success ration increases to 74%. In addition to that, an 89% success ratio was realized when the robot got trained for 300 episodes on the new environment. To get a better insight in how good these results are, it was compared with the results obtained when the robot started learning from scratch on the new environment. It was shown that this 74% success ratio can be obtained after almost 1300 training episodes from scratch. Thus, a 77% reduction in the number of training episodes can be achieved by transferring the policy which is prominent. Hence, these results highlight the generalization capability of the learned policy through transfer learning.
The success of reinforcement learning hinges on the agent's capabilities to effectively explore its environment. This is necessary because, initially, the agent has no knowledge about its environment and has to try different strategies in order to find successful ones. This results in the following research question

"What is the effect of different exploration noise on the quality of the learned trajectory and the learning rate?"

Most of today's state of the art algorithms still rely on traditional action space noise due to its implementation simplicity. In literature, it was proven that parameter space noise in which the perturbation is added to the network's parameters instead of the predicted actions achieves better results than the action space noise. By applying this exploration technique to the autonomous navigation problem, better performance in terms of the obtained Q-value and the reduction in collision sample by 32.03% is achieved. This improvement occurs due to the fact that the executed actions by the low level controller of the robot are conditioned on the current state. Thus, there's no much deviation in the trajectory to the same target point from one step to another. This is in contrast to the action space noise where the random perturbation is added to the predicted actions of the policy networks and thus the executed actions are not conditioned on the states.
Finally, the last research question was

"Is it possible to transfer the learned policy on the virtual environment directly to the real robot?"

The learned policy on the virtual environment has been transferred successfully to the real-robot where the robot managed to figure out an obstacle-free path to random desired targets without tuning any of the hyperparameters or conducting further training on the real-robot. Additionally, by learning a difference model that captures the mismatch between the real system and the simulated model and using this model to update the learned policy iteratively, it becomes possible to mimic exactly the learned trajectory on the virtual environment on the real robot.

8.1 Directions for Future Work

The presented research can be extended in different directions as detailed below.

Asynchronous Advantage Actor Critic (A3C)
Although the used deep deterministic policy gradient approach has achieved good learning results, it is still possible to improve these results by integrating A3C proposed in [52] with DDPG. The key idea of A3C is that learning can be parallelized using different threads that independently collect experience. The independent execution of multiple different environments may reduce the variance of the trained estimators since it provides the learning algorithm with many decorrelated training examples at one time. In this way, the robot can learn a more efficient policy to finish a navigation task.

Combining Reinforcement Learning with Imitation Learning
As mentioned in the critical appraisal in the previous chapter, the training time for learning continuous actions in a high-dimensional state-space is pretty long. One possible solution to this problem is to combine reinforcement learning with imitation learning. In that sense, it is possible to pre-train the navigation policy using expert demonstrations that can be generated through a path planning approach, i.e. *Move Base* in ROS. This can reduce the training time and the number of samples significantly compared to starting training by a random initialization of the weights and biases.

Curiosity-driven Exploration
To improve the exploration capability of the robot, especially for environments that impose considerable challenges, curiosity-driven exploration approach proposed in [53] can be used to urge the robot to explore states that it never or rarely visited before. This could motivate the agent to better explore the current environment, and make use of the structures of the environment for more efficient planning strategies.

Formulating the Navigation Problem as POMDP
In literature, the navigation problem is formulated as either a Markov decision process (MDP) or partially observable Markov decision process (POMDP). This is due to the fact that the state of the robot x_t can hardly capture all the necessary information for the agent to make decisions in the future since the state of the robot differs from the state of the environment s_t. Thus, the Markov property is not satisfied anymore, in which case the underlying procedure is called POMDP. This can be dealt with by either stacking several consecutive observations $\{x_{t-N+1}, ..., x_{t-1}, x_t\}$ to represent s_t as done in [11] or by feeding x_t into a recurrent neural network such that the past information is naturally taken into consideration. Hence, it is worth investigating if this would improve the efficiency of the learned policy.

A Appendix 1

A.1 Model-based Methods

A.1.1 Dynamic Programming

Dynamic programming is a class of approaches that can be used to find an optimal policy for a Markov decision process with the assumption that the transition probabilities that entail the dynamics of the environment is known. The model of the environment is not necessarily to be predetermined, but can rather be learned from data incrementally. Thus, given $\big(\mathcal{S}, \mathcal{A}, p(s_{t+1}|s_t, a_t), r(s_t, a_t), \gamma\big)$, it is required to find the optimal behavior by calculating an optimal value function and extracting an optimal policy π^* from it. Since a model is utilized in order to find the optimal policy π^*, this approach of solving a Markov decision process, is called a model-based approach. Optimal value functions and optimal policies are usually reached with iterative methods. Two of the most common algorithms in dynamic programming are called policy iteration and value iteration that can be used to find an exact solution for an MDP.

A.1.1.1 Value Iteration

The value iteration approach, also called backward induction, is first proposed by Bellman in 1957 [22]. This iterative algorithm calculates the expected value of each state using the value of the adjacent states until convergence to the real estimated value. The algorithm is shown in the following pseudo-code:

Algorithm 10 Value Iteration

Initialize value function $V(s)$, e.g. by $V(s) = 0$ for all states $s \in S$.
Initialize $\delta \leftarrow 0$ and $\epsilon > 0$ indicating the precision in estimation.
repeat
  for all $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) = \max_{a \in A} \sum_{s_{t+1}, r} p\,(s_{t+1}, r|s, a)\,\big(R_{t+1} + \gamma V(s_{t+1})\big)$
    $\delta = \max(\delta, |v - V(s)|)$
  end for
until $\delta < \epsilon$

Here it should be pointed out that value iteration combines the steps of policy evaluation and policy improvement that are discussed in the next section by updating the value function every time a state is updated.

A.1.1.2 Policy Iteration

As discussed in the previous subsection, value iteration algorithm keeps improving the value function at each iteration until the value function converges. However, the main goal of the agent is to find the optimal policy , therefore, another algorithm called policy iteration was proposed [54]. Instead of improving the value-function estimate repeatedly, it will redefine the policy at each step and compute the value function according to this new policy until the policy converges. Policy iteration alternates between two phases, policy evaluation and policy improvement that are discussed in this subsection.

Policy Evaluation
In dynamic programming, policy evaluation is a way of computing the state-value function $V^\pi(s)$ for an arbitrary policy π. This algorithm can be used to evaluate an arbitrary policy for all

states in the state space $s \rightarrow \mathcal{S}$ and update them iteratively by applying Bellman's expectation equation introduced in the previous section. This is done by means of synchronous backups where each state is visited and its value is updated based on the current value estimates of all its possible successor states, weighed by the associated transition probabilities as well as the policy. The update rule is given by

$$V_{k+1}(s_t) = \sum_a \pi(a|s) \sum_{s_{t+1},r} p(s_{t+1}, r|s, a) \left(R_{t+1} + \gamma V^{\pi}(s_{t+1})\right) \tag{A.1}$$

For iterative policy evaluation, a natural question is when does the algorithm converge to $V^{\pi}(s)$ and the iteration can be stopped. For a given threshold $\epsilon > 0$, the algorithm can be stopped when the difference between one iteration to the other is below this threshold. Mathematically, this can be expressed as

$$\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \epsilon \tag{A.2}$$

At this point, it is guaranteed that the value function converges to its true value. After policy π has been evaluated, one wants to improve π by finding a new policy π' that is at least as good as the old policy π or even better. This approach is called policy improvement.

Policy Improvement

The second phase of policy iteration is policy improvement which makes the policy greedy with respect to the current value function [18]. As stated before, firstly, an initial policy is chosen. Then, the policy is evaluated until the iterative policy evaluation converges to a predefined desired accuracy of estimation. By maximizing over the new value function, a new better policy π' can be extracted, which will be again evaluated and improved to an even better policy π'' until this process converges to the optimal policy π^*.

$$\pi_0 \xrightarrow{PE} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{PE} V^{\pi_1} \xrightarrow{I} \pi_2 ... \xrightarrow{I} \pi^* \xrightarrow{PE} V^{\pi^*}$$

where $\xrightarrow{PE}$ denotes policy evaluation and $\xrightarrow{I}$ denotes policy improvement. The algorithm terminates when the policy stabilizes and an optimal policy is obtained. The process of alternating between policy evaluation and policy improvement is called generalized policy iteration (GPI) which is depicted in Figure A.1. The policy iteration algorithm is described in Algorithm 11.

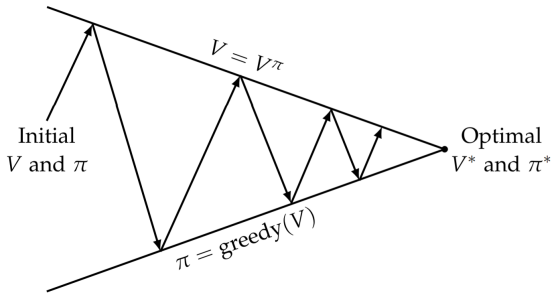


Figure A.1: Illustration of how policy iteration converges to the optimal policy and value function.

Algorithm 11 Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi^*$

Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ randomly for all $s \in S$
Initialize $\delta \leftarrow 0$ and stable $\leftarrow True$
while stable = $True$ **do**

Phase 1 – Iterative Policy Evaluation, for estimating $V \approx V^{\pi}$

repeat
for each state $s \in S$ **do**
$v \leftarrow V(s)$
$V(s) = \sum_a \pi(a|s) \sum_{s_{t+1},r} p(s_{t+1}, r|s, a) \left(R_s^a + \gamma V(s_{t+1})\right)$
$\delta = \max(\delta, |v - V(s)|)$
end for
until $\delta < \epsilon$

Phase 2 – Policy Improvement

for each state $s \in S$ **do**
$a = \pi(s)$
$\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s_{t+1},r} p(s_{t+1}, r|s, a) \left(R_{t+1} + \gamma V^{\pi}(s_{t+1})\right)$
if $a \neq \pi(s)$ **then**
stable $\leftarrow False$
end if
end for
end while

B Appendix 2

B.1 Improved Proposal Distribution

As shown in (4.8), since the product $p(x_t^{(n)}|x_{t-1}^{(n)}, u_{t-1})p(z_t|x_t^{(n)}, m_{t-1}^{(n)})$ is dominated by the observation likelihood function $p(z_t|x_t^{(n)}, m_{t-1}^{(n)})$, the odometry motion model $p(x_t^{(n)}|x_{t-1}^{(n)}, u_{t-1})$ is approximated by a constant k within the meaningful area $L^{(n)}$ given by

$$L^{(n)} = \left\{ x_t^{(n)} \;\middle|\; p(z_t|x_t^{(n)}, m_{t-1}^{(n)}) > \epsilon \right\} \tag{B.1}$$

where ϵ is a user-defined threshold. Under this approximation, equation (4.8) can be reformulated as

$$p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_{t-1}, z_t, m_{t-1}^{(n)}\right) \approx \frac{p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right)}{\int_{x_t \in L^{(n)}} p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) \mathrm{d}x_t}, \tag{B.2}$$

In [55], it is argued that equation (B.2) is still not suitable for efficient sampling. However, since in most of the cases the observation model is a uni-modal distribution, thus, it is possible to approximate the optimal proposal by a Gaussian function:

$$p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_t, z_t, m_{t-1}^{(n)}\right) \approx \mathcal{N}\left(\mu_t^{(n)}, \Sigma_t^{(n)}\right) \tag{B.3}$$

where $\mu_t^{(n)}$ and $\Sigma_t^{(n)}$ represent the mean and covariance matrices for the Gaussian distribution respectively and given by

$$\mu_t^{(n)} = \frac{1}{\eta} \sum_{j=1}^{K} x_j \, p\left(z_t \;\middle|\; m_{t-1}^{(n)}, x_j\right), \tag{B.4}$$

$$\Sigma_t^{(n)} = \frac{1}{\eta} \sum_{j=1}^{K} x_j \, p\left(z_t \;\middle|\; m_{t-1}^{(n)}, x_j\right) \left(x_j - \mu_t^{(n)}\right) \left(x_j - \mu_t^{(n)}\right)^T. \tag{B.5}$$

where the normalizer η is given by $\sum_{j=1}^{K} p(z_t|m_{t-1}^{(n)}, x_j)$. In this way, a closed form approximation of the optimal proposal distribution is obtained which is suitable for sampling. Additionally, by substituting the improved proposal in equation (4.7), the weights can be computed as

$$\begin{aligned}
w_t^{(n)} &= w_{t-1}^{(n)} \frac{\eta p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_{t-1}\right)}{p\left(x_t \;\middle|\; m_{t-1}^{(n)}, x_{t-1}^{(n)}, z_t, u_{t-1}\right)} \\
&= w_{t-1}^{(n)} \frac{p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_{t-1}\right)}{\frac{p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_t\right) p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right)}{\int_{x_t} p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_{t-1}\right) \mathrm{d}x_t}} \\
&= w_{t-1}^{(n)} \int_{x_t} p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) p\left(x_t^{(n)} \;\middle|\; x_{t-1}^{(n)}, u_{t-1}\right) \mathrm{d}x_t = w_{t-1}^{(n)} k \int_{x_t \in L^{(n)}} p\left(z_t \;\middle|\; x_t^{(n)}, m_{t-1}^{(n)}\right) \mathrm{d}x_t \\
&= w_{t-1}^{(n)} k \sum_{j=1}^{K} p(z_t|m_{t-1}^{(n)}, x_j) = w_{t-1}^{(n)} k \eta^{(n)}
\end{aligned} \tag{B.6}$$

where $\eta^{(n)}$ is the same normalization factor that is used in the computation of the Gaussian approximation of the proposal in equation (*B*.5).

Bibliography

[1] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *IEEE International Conference on Robotics and Automation*, pp. 500–505, 1985.

[2] A. Stentz, "Optimal and efficient path planning for partially-known environments," *IEEE International Conference on Robotics and Automation*, pp. 3310–3317, 1994.

[3] J. Kuffner and S. LaValle, "Rrt-connect: An efficient approach to single-query path planning," *IEEE International Conference on Robotics and Automation*, pp. 995–1001, 2000.

[4] J. Koper, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: a survey," *The International Journal of Robotics Research*, pp. 1238–1274, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersin, C. Beatti, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature International Journal of Science*, pp. 529–533, 2015.

[6] P. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tasaa, D. Silver, and D. Wierstra, "Continous control with deep reinforcement learning," *International Conference on Learning Representations*, 2016.

[7] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Mortiz, "Trust region policy optimization," *The 32nd International Conference on Machine Learning,ICML*, pp. 1889,1897, 2015.

[8] S. Levine, C. Finn, T. Darrell, and P. Abbeel., "End-to-end training of deep visuomotor policies," *arXiv:1504.00702*, 2015.

[9] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection.," *arXiv:1603.02199*, 2016.

[10] L. Pinto and A. Gupta., "Supersizing self-supervision: Learning to grasp from 50k tries and700 robot hours," *IEEE International Conference on Robotics and Automation, ICRA*, pp. 3406,3413, 2016.

[11] J. Zhang, J. Springenberg, J. Boedecker, and W. Burgard, "Deep reinforcement learning with successor features for navigation across similar environments," *arXiv:161205533*, 2016.

[12] G. Brunner, O. Richter, Y. Wang, and R. Wattenhofer, "Teaching a machine to read maps with deep reinforcement learning," *arXiv:171107479*, 2017.

[13] J. Zhang, L. Tai, J. Boedecker, and M. Liu, "Neural slam: Learning to explore with external memory," *arXiv:170609520*, 2017.

[14] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, and I. Antonoglou, "Mastering the game of go with deep neural networks and tree search.," *Nature Journal*, pp. 484–489, 2016.

[15] Y. Bengio, "Learning deep architectures for ai.," *Foundations and Trends in Machine Learning*, pp. 1–127, 2009.

[16] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator.," *International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[17] M. L. Puterman, "Markov decision processes: discrete stochastic dynamic programming," *John Wile & Sons*, 2014.

[18] R. Sutton and A. Barto, "Reinforcement learning: An introduction," *MIT Press*, 2017.

[19] K. Arulkumaran, M. Deisenroth, M. Brundage, and A. Bharath, "A Brief Survey of Deep Reinforcement Learning,," in *IEEE Signal Processing Magazine*, 2017.

[20] A. Ng, D. Harada, and S. Russel, "Policy invariance under reward transformations: Theory and application to reward shaping," *ICML '99 Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278,287, 1999.

[21] L. Tai, J. Zhang, M. Liu, J. Boedecker, and W. Burgard, "A survey of deep network solutions for learning control in robotics: From reinforcement to imitation," *arXiv preprint arXiv:1612.07139*, 2018.

[22] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[23] R. Sutton, "Temporal credit assignment in reinforcement learning," *Doctoral dissertation, University of Massachusetts, Amherst*, 1984.

[24] C. Watkins, "Learning from delayed rewards," in *PhD thesis, King's College, London*, 1989.

[25] J. Peters and J. Bagnell, "Policy Gradient Methods," in *Encyclopedia of Machine Learning*, pp. 746–819, 2011.

[26] R. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," in *Machine Learning*, pp. 229–256, 1992.

[27] S. Haykin, "Neural Networks and Learning Machines," in *Third Edition, Pearson Education*, pp. 229–256, 1993.

[28] D. Kingma and J. Ba, "Adam: a Method for Stochastic Optimization,," in *International Conference on Learning Representations*, pp. 1–15, 2015.

[29] D. Silver, "A course on reinforcement learning," *University College London*, 2015.

[30] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *AAAI Conference on Artificial Intelligence*, pp. 2094–2100, 2016.

[31] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[32] T. Degris, M. White, and R. Sutton, "Off-policy Actor-critic,," in *International Conference on Machine Learning*, 2012.

[33] R. Sutton, D. McAllester, and S. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Neural Information Processing Systems*, vol. 12, pp. 1057–1063, 1999.

[34] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *31st International Conference on Machine Learning*, vol. 32, pp. 1309–1332, 2014.

[35] K. Murphy, "Bayesian map learning in dynamic environments," *Neural Information Processing Systems*, vol. 12, pp. 1015–1021, 1999.

[36] S. Thrun, W. Burgard, and D. Fox, "Probabilistic robotics," *MIT Press*, 2005.

[37] B. Siciliano and O. Khatib, "Springer handbook of robotics," *Springer Publishing Company, 2nd edition*, 2016.

[38] A. Weitzenfeld, J. Fellous, A. Barrera, and G. Tejera, "Allothetic and idiothetic sensor fusion in rat-inspired robot localization," *Multisensor, Multisource Information Fusion: Architectures, Algorithms, and Applications*, vol. 8407, 2012.

[39] H. Moravec, "Sensor fusion in certainty grids for mobile robots," *AI Magazine*, pp. 61–74, 1988.

[40] A. Doucet, N. de Freitas, and N. Gordan, "Sequential monte carlo methods in practice," *Springer Verlag*, 2001.

[41] G. Grisetti, C. Stachniss, and W. Burgard., "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," *IEEE conference on robotics and automation*, pp. 2443–2448, 2005.

[42] F. Dellaert, D. Fox, W. Burgard, and S. Thrun., "Monte carlo localization for mobile robots," *IEEE conference on robotics and automation*, 1998.

[43] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit., "Fastslam: A factored solution to simultaneous localization and mapping," *The national conference on artificial intelligence*, 2002.

[44] A. Doucet, "On sequential simulation-based methods for bayesian filtering," *Technical report, Signal processing group, University of Cambridge*, 1998.

[45] G. Grisetti, C. Stachniss, and W. Burgard., "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE Transactions on Robotics*, pp. 34–47, 2007.

[46] J. Liu, "Metropolized independent sampling with comparisons to rejection sampling and importance sampling," *Statistics and computing*, pp. 113–119, 1996.

[47] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv:1502.03167v3*, 2015.

[48] I. Goodfellow, Y. Benglo, and A. Courville, "Deep learning (adaptive computation and machine learning series)," *The MIT Press*, 2016.

[49] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, "Parameter space noise for exploration," *International Conference on Learning Representation*, 2018.

[50] J. L. Ba, J. Kiros, and G. Hinton, "Layer normalization," *arXiv:1607.06450*, 2016.

[51] D. Rastogi, I. Koryakovskiy, and J. Kober, "Sample-efficient reinforcement learning via difference mod," *3rd Machine Learning in Planning and Control of Robot Motion Workshop at ICRA*, 2018.

[52] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

[53] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," *International Conference on Machine Learning*, 2017.

[54] E. Pashenkova, I. Rish, and R. Dechter, "Value iteration and policy iteration algorithms for markov decision problem," *In AAAI'96: Workshop on Structural Issues in Planning and Temporal Reasoning*, 1996.

[55] C. Stachniss, G. Grisetti, W. Burgard, and N. Roy., "Analyzing gaussian proposal distributions for mapping with rao-blackwellized particle filters," *IEEE international conference on intelligent robots and systems*, 2007.