Politecnico di Torino

Master's Degree in MECHATRONIC ENGINEERING

Department of Electronics and Telecommunications

Master Thesis

Hybrid Deep Reinforcement Learning-based Collision Avoidance Algorithm for a Ground Robot in Indoor Environments

Supervisors:
Dr. Elisa Capello
Dr. Hyeongjun Park

Candidate:
Francesco Melis

Academic Year 2020–2021

Ai miei amati nonni Alfredo ed Efisio

Acknowledgments

I would like to thank my supervisors, Dr. Elisa Capello and Dr. Hyeongjun Park, for their invaluable assistance toward the exploration of such a fascinating and innovative research field. I am very grateful for their motivational support and experienced suggestions.

Hybrid Deep Reinforcement Learning-based Collision Avoidance Algorithm for a Ground Robot in Indoor Environments

Candidate: Francesco Melis
Supervisors: Dr. Elisa Capello
Dr. Hyeongjun Park

Abstract

The rapid development of Artificial Intelligence (AI) is revolutionizing an increasing number of fields and industries. The exploration of another planet through autonomous rovers could be considered the most fascinating application of such technologies. These robots need efficient and robust autonomous guidance and navigation techniques to make decisions and avoid obstacles in challenging and partially unknown environments. Machine Learning is a type of AI, and Deep Reinforcement Learning is one of the most recent and promising techniques to face this challenge among its branches. It combines the framework of the Reinforcement Learning approaches, where an agent learns a policy that maps states into actions by interacting with an environment and obtaining a numerical reward depending on its behaviour, with the approximation ability of the Deep Neural Networks.

Inspired by these considerations, this thesis focuses on the development of a collision avoidance algorithm based on Deep Reinforcement Learning applied to a ground robot equipped with a depth camera. A Neural Network, that represents the policy of the agent, has been trained to map a set of inputs obtained from simulated odometry and camera data into linear and angular velocity of the robot. The training has been performed using Proximal Policy Optimization (PPO), that is a state-of-the-art policy learning algorithm, and a multi-stage approach, consisting in training the agent in simulated scenarios characterized by incremental complexity, suitably designed. To increase the performance of the control algorithm, a hybrid control framework has been implemented to switch between the PPO stochastic policy and a deterministic policy able to find a time-convenient path to reach the target in absence of obstacles. The deterministic policy has been obtained by training a Neural Network using the Deep Deterministic Policy Gradient (DDPG) and the switching process is based on robot's sensors measurements of the environment.

The algorithm has been tested through MATLAB simulations in several different and challenging scenarios, showing good performance in avoiding static obstacles. Finally, the algorithm has been verified on a Gazebo simulator, revealing acceptable performance dealing with complex environments in a more realistic framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The rapid development of Artificial Intelligence (AI) is revolutionizing our world: from finance to agriculture, from healthcare to cybersecurity, from surveillance to marketing, a huge number of fields and industries are taking advantages of such innovative and powerful techniques [1–3]. The space sector is recently catching up to this trend, applying advanced computer algorithms to problems in space science and technologies [4–6]. For example, the exploration of another planet through autonomous rovers could be considered as one of the most fascinating applications of these technologies. Autonomous guidance and navigation techniques are essential in space exploration missions: they allow robots to traverse unknown and complex terrains covering more distance with respect to the situation with human drivers in the loop [7]. In addition to autonomy, rover navigation must be safe and efficient: the aim is to reach a human-like ability to identify and avoid possible hazards, such as steep slopes or large rocks, allowing the rovers to safely cross over hard terrains while saving energy consumption, since the robots feature a limited amount of energy per day that is used for moving, using science instruments, and communicating with Earth. The evolution of NASA Mars Rovers in the last three decades, motivated by the great communication latency between Mars and Earth, shows great developments in this direction: from Sojourner to Perseverance, passing through Spirit, Opportunity, and Curiosity, each rover has been designed to travel safely and efficiently more distance in a Martian day operation with fewer instructions from engineers on Earth [8–14].

In recent years, the literature has shown an increasing trend in applying Deep Reinforcement Learning (DRL), a branch of Machine Learning (ML), which in turn is a type of AI, to Autonomous Mobile Robot Navigation. DRL combines the frame-

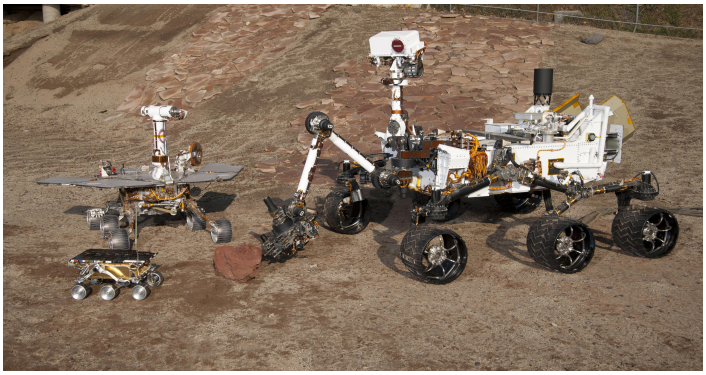


Figure 1.1: From left to right: Sojourner, a Mars Exploration Rover project test rover and Curiosity in JPL's Mars Yard testing area [credit: NASA/JPL-Caltech].

work of the Reinforcement Learning approaches, where an agent learns a policy that maps states into actions by interacting with an environment and obtaining a numerical reward depending on its behaviour, with the approximation ability of the Deep Neural Networks. The advantages of such innovative methods, that rely on map-free, robust adaptability, and limited influence from sensors quality [15], have attracted the interest of the robotic research community. Inspired by these considerations, the aim of the thesis is to study and implement Deep Reinforcement Learning techniques to develop a collision avoidance algorithm to make a ground robot able to autonomously move in an indoor environment, reaching a user-defined target avoiding the obstacles present in the path.

1.2 Background

1.2.1 Machine Learning

Machine Learning represents one of the most important and prolific AI branches. It has been defined by Michie et al. [16] as an ensemble of *"automatic computing procedures based on logical or binary operations that learn a task from a series of examples"* . ML approaches are usually divided into three separate categories [17–19]:

- In *supervised learning*, an agent is provided with a labelled set of training examples, consisting of input and expected output couples, by an external supervisor. The objective is to make the agent able to develop a generalization ability to work properly with different situations with respect to the ones present in the training set.
- In *unsupervised learning*, the agent is provided with unlabelled training data without the presence of a supervisor and is tasked with finding patterns hidden in the collection to determine how to group (cluster) the elements accordingly.

- In *reinforcement learning*, an agent learns in a trial-and-error manner by acting interactively within an environment that provides feedbacks in the form of rewards based on the actions taken.

1.2.2 Reinforcement Learning

Reinforcement Learning has been described by Sutton and Barto [19] as *"learning what to do - how to map situations to actions - so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them"*.

1.2.2.1 Markov Decision Process

RL algorithms work on systems that are formalized as *Markov Decision Processes* (MDPs). MDPs are mathematical structures that idealize sequential decision-making problems in finite forms, aimed to represent their essential features in a simple way. The core of the MDP is learning from interaction to achieve a goal. The *agent*, that is the learner and decision maker, interacts with the *environment*, defined as everything that is not the agent. The interaction occurs at each discrete time step: the agent receives a representation of some aspects of the environment's *state* $S_t \in \mathcal{S}$ and as a response selects a specific *action* $A_t \in \mathcal{A}(s)$. Consequently, in the next time step, the agent receives a *reward* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, that is a simple number, and is located in a new state S_{t+1} [19].

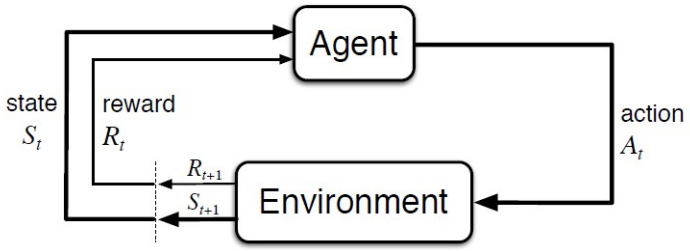


Figure 1.2: The agent–environment interaction in a Markov Decision Process [19].

A *finite* MDP is defined as a MDP whose sets of states, actions, and rewards have a finite number of elements. In a finite MDP the random variables R_t and S_t can be described by precise discrete probability distributions that depend only on the preceding state and action. This is defined by the *transition dynamics function* p, that represents the dynamics of the MDP:

$$p(s', r \mid s, a) \doteq Pr\{S_t = s', R_t = r, \mid S_{t-1} = s, A_{t-1} = a\},$$

where p indicates, for particular values of these random variables $s' \in S$ and $r \in R$, the probability of those state and reward to occur at time t given specific values of

the anterior state and action values.

1.2.2.2 Goals and Rewards

The reward signal represents the goal of the reinforcement learning problem. It is crucial to use it to communicate the agent *what* is the goal, not *how* to reach it, otherwise hypothetically the agent could learn a way to achieve some eventually set subgoals without obtaining the final desired goal. More precisely, the agent's objective is to maximize the *discounted expected return*, where the return G_t is defined as a function of the sequence of the rewards received after time step t over the future:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ is a parameter from 0 to 1, called the *discount rate*, whose effect is to make that immediate rewards contribute more to the sum than the future ones. The discount factor has to be suitably chosen for the specific problem: when $\gamma = 0$ the agent is myopic since is interested only in maximizing immediate rewards, while as γ approaches to 1 the agent becomes more farsighted since it gains interest in future rewards.

1.2.2.3 Policies and Value Functions

The core of a RL agent, or, to use the fittest analogy, the *brain*, is represented by the *policy.* The policy is an input-output relation that defines how the agent interacts with the environment: it is a mapping that selects actions based on the perceived states from the environment. In simple problems, the policy can be a simple function or lookup table. A policy can be *deterministic*, mapping each state to a single specific action, or, in the more general case, *stochastic*, assigning probabilities to each action in each state [19, 20].

Algongside the concept of policy, present in all the RL algorithms, a huge variety of them it is based also on the concept of *value function.* Value functions estimate the expected reward that an agent can expect in the future for being in a certain state or for acting through a specific action in a certain state. Since the behaviour of the agent depends on the policy it has learned, value functions are defined on the basis of a specific policy.

To give a formal definition, the *value function* $v_\pi(s)$ of a state s under a policy π is defined as the expected return when starting from state s and acting following π from there on. In parallel, the *action value function* $q_\pi(s, a)$ represents the value of taking action a in state s following a policy π, and it is defined as the expected return starting from s, selecting action a, and from there on acting through policy

π [20].

1.2.2.4 Bellman Equations

Bellman equations represent the basis on which many RL methods have been developed. The *Bellman equation for the state value function* $v_\pi(s)$ provides a powerful general relationship for MDPs by defining a relationship between the value of a state and the value of its possible successor states [19]:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')]$$

It states that the value of a state is equal to the sum of the expected reward obtained plus the value of the expected next state, considering all the possibilities and weighting each of these by the probability it occurs.

Analogously, The *Bellman equation for the action vale function* $q_\pi(s, a)$ provides a relationship between the value of a state-action pair and the possible next state-action pairs [20]:

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r \mid s, a)[r + \gamma \sum_{a'} \pi(a \mid s') q_\pi(s', a')]$$

Their importance is related to the fact that they can reduce an infinite sum over possible futures to a system of linear equations. However, their direct use is possible only for MDPs characterized by small sets.

1.2.2.5 Optimal Policies and Optimal Value Functions

In general, finding the solution of an RL problem can be interpretd as finding a policy able to achieve a sufficiently large amount of rewards. For a finite MDP it is possible to precisely define an *optimal policy* as the policy whose expected return is greater than or equal to that of all the other policies for all states. The optimal policy could be one or more than ones. In any case, all the optimal policies are indicated as π_* and they share the same *optimal state-value function*, denoted v_*, and the same *optimal action-value function*, denoted q_* [19].

For finite MDPs with n states, the *Bellman optimality equation*, i.e., the Bellman equation for v_*, represents a system of n equations in n unknowns and it has a unique solution. In principle it is possible to solve this system of nonlinear equations, provided that 3 conditions are met [19]:

- The dynamics p of the environment is completely known;
- It is available sufficient computational power to compute the solution;

- The state must include information about all aspects of the past interaction between the agent and the environment that are significant for the future (this is known as *Markov property*).

In practice, these conditions are systematically violated, and the search moves to approximate solutions. However, many RL algorithms are based on the approximate solution of the Bellman optimally equation.

1.2.2.6 Features of RL Algorithms

Modern RL literature presents a huge variety of different methods. In order to close this brief background review a collection of the most important features used for distinguishing among the several different classes of algorithms is presented:

- **Model-free and model-based algorithms:** some RL methods, called *model-based* methods, exploit models to replicate the behaviour of the environment to *plan*, i.e., to decide an action by considering possible future scenarios before they actually happen [19]. By contrast, more popular *model-free* methods are based on an explicit trial-and-error way of learning and in general are easier to implement.

- **Online and offline learning algorithms:** in the *offline* learning only a limited amount of previously collected data about a specific environment is available, whereas in *online* learning algorithms the agent gradually obtains experience in the environment [21].

- **On-policy and off-policy algorithms:** *on-policy* algorithms are based on the improvement of the same policy that is used to make decisions, whereas *off-policy* algorithms evaluate a policy different from the one used to generate the data [21].

- **Value-based and policy-based algorithms:** *value-based* methods imply the optimization of the action-value function $q_\pi(s, a)$ and the optimal policy is then derived, whereas *policy-based* methods directly optimize the policy until the cumulative reward is maximized [22]. Some algorithms exploit a combination of the two methods.

1.2.3 Deep Reinforcement Learning

For most real-world situations, the high-dimensionality of the state space does not allow to apply traditional RL approaches to these problems. *Deep Reinforcement Learning*, that combines RL framework with deep learning techniques, has made it possible to tackle a wide range of complex decision-making problems previously intractable for the machines [22]. The basic idea is to use multi-layer Artificial

Neural Networks (ANNs) as function approximators of the value functions or to directly represent the policy functions. The reason is that ANNs are effective when dealing with large datasets and they do not need an exponential increase of data when the dimension of the action or state space is augmented. Furthermore, they can be trained in an incremental way as the additional samples are obtained during the learning process [21].

1.2.3.1 Neural Networks

ANNs are networks of interconnected units or nodes, called *neurons*, that emulate the biological neurons that compose the human brain. Each connection, that resembles the synapses of biological brains, can transmit a signal, that is a real number, to other neurons. Each neuron receives this signal, processes it, and then sends the output impulse to other neurons. The connections are characterized by a weight, that is adjusted during the learning process and represents the strength of the influence of a neuron on the others. Typically, the neurons produce their output by computing the weighted sum of their input signals, eventually adding a bias, and then applying the result to a nonlinear function, called the *activation function* [22–24].

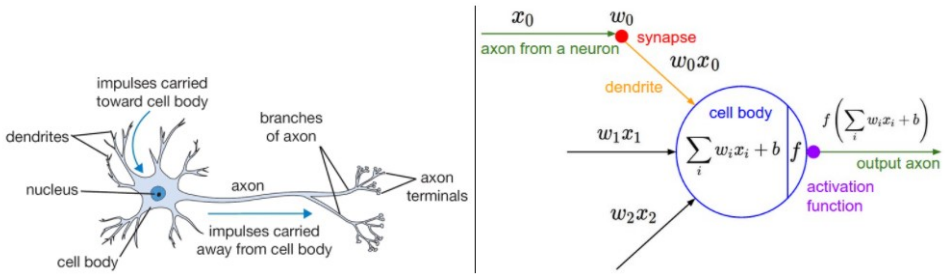


Figure 1.3: A sketch of a biological neuron (left) and its mathematical model (right) [23].

The nonlinearity introduced in this way is essential for the approximation power of the ANNs and the choice of the activation function varies depending on the application. The following are some of the most common activation functions:

- The logistic **sigmoid** squashes real numbers to range between 0 and 1 as defined by the following equation:

$$f(z) = \frac{1}{1 + e^{-z}}$$

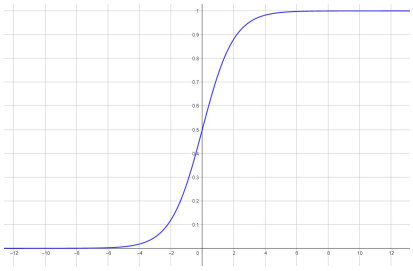


Figure 1.4: Sigmoid function.

- The **hyperbolic tangent (tanh)** constrains output values to a range between -1 and 1 as defined by the following equation:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

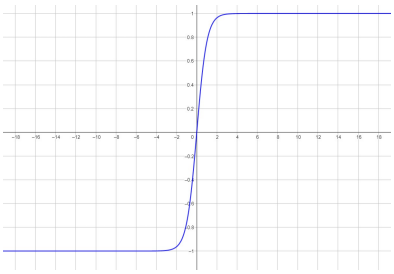


Figure 1.5: Tanh function.

- The **rectified linear unit (ReLu)**, also called rectifier, simply outputs 0 if the input is negative or equal to 0, or outputs the input itself if it is positive:

$$f(z) = \begin{cases} 0 & when \ z \leq 0 \\ z & when \ z > 0 \end{cases}$$

Figure 1.6: ReLu function.

- The **softmax** function is very different from the previous ones, since it takes

a vector of K real numbers as input and normalizes it into a probability distribution composed by K probabilities proportional to the exponentials of the input numbers. Each component of the output vector is ranged between 0 and 1 and their sum is 1 [25].

$$f(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^z}$$

In general, the neurons are organized into distinct layers, and different layers may apply different transformation on their inputs. The signal is transmitted from the input layer, that takes the input features, through the hidden layers, called in this way because they are generally not accessible from outside the network, to the output layer. In the simplest case, the *feedforward neural network*, the signal is transmitted from the front to the back in a very straightforward way [26], but there exist dozens of different and articulated types of ANN structures, each one characterized by unique features.

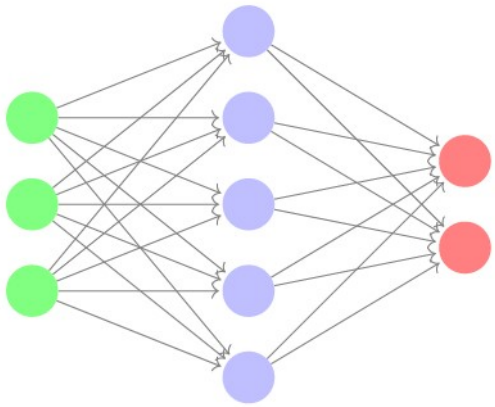


Figure 1.7: An example of feedforward ANN with 3 input units, one hidden layer with 5 units and 2 output units [21].

1.2.3.2 Deep Neural Networks

Therefore, functions are parametrized by the network's connection weights. The *universal approximation theorem* states that a feedforward neural network with a single hidden layer using arbitrary squashing functions is able to approximate any Borel measurable function to any desired level of accuracy if containing a sufficiently large number of neurons [27]. Nonetheless, in practice, if the single hidden layer has too many neurons, inflexible training or overfitting problems may arise [22]. For this reason, it is usually preferred to use more than one hidden layers, and this kind of structure is generally known as *Deep Neural Network* (DNN). The *depth* of a neural

network is defined as the number of hidden layers that contains. The successive layers of a DNN allow to implement an abstract hierarchical representation of the network's input [2, 19], making easier the task of approximating complex functions.

1.3 Thesis Outline

The thesis is organized as follows:

- In chapter 2, the problem of robotic autonomous navigation through Deep Reinforcement Learning is formulated. Furthermore, the structure of the reinforcement learning agent and the kinematic model of the robot are presented.
- In chapter 3, the ingredients for the development of the main character of this work, i.e., the PPO agent, are presented, as well as the features and working principles of the adopted learning algorithm.
- In chapter 4, the motivation behind the implementation of a hybrid architecture is presented, as well as the implementation of the second reinforcement learning agent, trained by DDPG algorithm, and the features and working principles of the algorithm.
- In chapter 5, all the implementation details adopted for the design of the two agents are presented, e.g., hyperparameters used. Furthermore, the learning curves for both the agents are displayed, and results about the performances of the agents are illustrated.
- In chapter 6, the control algorithm that deploys the RL agents is presented, and the results regarding MATLAB and Gazebo simulations of its implementation are discussed.
- In chapter 7, the conclusions about the thesis and the ideas for future works are discussed.

Chapter 2

Deep Reinforcement Learning-based Autonomous Navigation

2.1 Introduction

In recent years, the application of Deep Reinforcement Learning to automatic collision avoidance has rapidly achieved an increasing interest in robotic research community. Robotic autonomous navigation problems involve, in addition to the aforementioned problem of the high dimensionality of the state space, another critical challenge for the traditional RL methods: the partial observability of the environment. Actually, the application of Deep Neural Networks has also made possible to tackle partially observable tasks, in which the agent has not a complete knowledge about the environment's state. Indeed, in most real-world situations the agent features a limited perception of the environment, increasing the complexity of the problem. To deal with this kind of situations, the concept of *Partially Observable Markov Decision Process* (POMDP) has been introduced [28]: a POMDP is defined as an MDP where an agent receives an observation that contains only partial information of the environment state [22, 29].

2.2 Problem Formulation

The collision avoidance problem can be formulated as a POMDP defined by a six-tuple (S,A,P,R,Ω,O), where S is the state space, A is the action space, P is the transition function, R is the reward function, Ω is the observation space, and O is the probability function, that defines how the observations are obtained from the environment state [30–33]. The agent, at each time step, has access to an observation vector which it uses to compute a collision-free action that drives

the robot towards its goal from the current position by avoiding collisions with the obstacles. The robot has been modelled as a differential drive vehicle, a configuration where the wheels rotate around the center point of the robot [34]. Through the observations, the agent, that can be likened to the robot's brain, can perceive the environment up to the possibilities of the sensors the robot is equipped with, and no global information about the environment is available. More precisely, it detects the presence of the obstacles in front of it thanks to a Depth Camera and it knows the relative position of the user-defined goal, since it is computed on the basis of the data obtained from its odometry sensors. The action that the agent produces as an output is a bidimensional vector composed by the linear and angular velocity, used to command robot movements. To guide the agent to achieve the desired performance, the Reward function is suitably designed.

2.3 Reinforcement Learning Agent

The most important thing of the problem is, as mentioned, the agent. In general, a Reinforcement Learning agent can be considered composed of two elements [35]:

- The policy, that maps observations into actions and it is represented by a Deep Neural Network, whose structure has to be suitably designed.
- The *learning algorithm*, that updates the policy parameters based on the input observations, actions taken, and rewards collected. Its goal is to find an optimal policy that maximizes the cumulative reward received during the task. The learning algorithm can be chosen among the state-of-the-art ones present in literature; however, they present a number of hyperparameters that need to be suitably tuned for the specific problem.

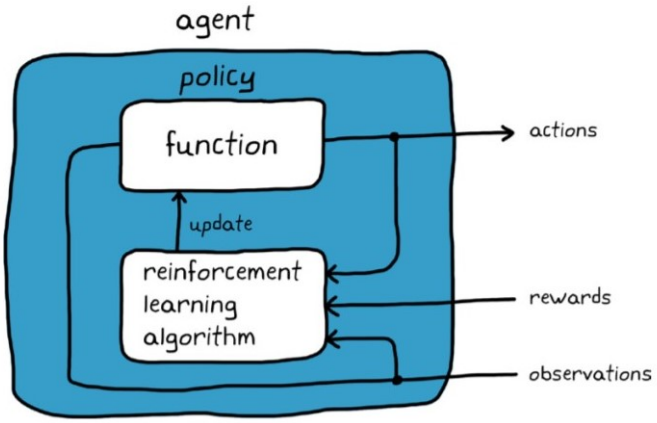


Figure 2.1: A scheme of a generic Reinforcement Learning agent [35].

2.4 Robot Kinematic Model

The target hardware is a differential drive robot. It is described by the same kinematic model of a unicycle, but it features a more stable structure from a mechanical point of view [36]. The robot pose in Cartesian coordinates is represented by a vector of three variables:

$$p = [x, y, \theta]^T ,$$

where x and y represent the position of the center of mass of the robot [**?**, 37] and the angle θ represents the orientation of the robot. The model is described by the following equation:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} cos\theta \\ sin\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega,$$

where v and ω are the *driving* (linear) and *steering* (angular) velocity. Furthermore, v and w are related to the *rotation velocities* of the two wheels, ω_L and ω_R, by the following relations:

$$v = \frac{r(\omega_R + \omega_L)}{2},$$

$$\omega = \frac{r(\omega_R - \omega_L)}{d},$$

where r is the radius of the wheels and d is the distance between them.

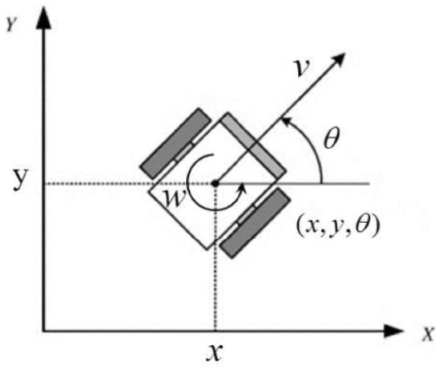


Figure 2.2: Kinematic model of a differential drive robot [37].

Chapter 3

PPO Agent Development

The core and decision-maker of the collision avoidance algorithm developed in this work is represented by a Proximal Policy Optimization (PPO) agent. PPO is the reinforcement learning algorithm chosen to train the agent. To create the agent, the elements introduced in the previous chapter have to be suitably designed. These are the action and observation spaces, the reward function, the learning algorithm, and the policy representation. Thereafter, the agent has to be properly trained to obtain the desired performance.

3.1 Observations, Actions, and Rewards

The observation vector is the input on the basis of which the agent decides the action to compute at each time instant. This vector has been designed to give information to the robot about the target direction and the local presence of obstacles in the path. For the first purpose, the observation vector contains the target position in polar coordinates with respect to the robot local frame, i.e., distance and angle. Secondly, the real robot is equipped with a Depth Camera that outputs the depth measurement of the objects present inside its Field Of View, up to specific limits. This data can be used to derive the ranges of the obstacles with respect to the robot, emulating a 2-D LiDAR featured with additional 3-D information. More in detail, the raw output of the camera is processed inside the algorithm developed to obtain a manageable number n_{ranges} that encapsulates the distances from the robot of the objects detected by the camera. Keeping the size of the observation vector small, that is $n_{obs} = n_{ranges} + 2$, allows to have faster training phases and simpler neural networks.

The action space has been designed as a set of couples of permissible linear and angular velocities in a discrete space. A finite set of atomic actions enables a faster learning rate and, furthermore, can represent a flexible distribution if characterized by a proper number of actions, even more flexible with respect to a continuous Gaus-

sian distribution [38]. The dimension of the set and the specific values of the n_{act} actions have been decided based on the maximum linear and angular velocity of the target robot, making several trials and evaluating learning speed and performances in simulation.

The reward function has been designed with the primary goal of reaching the target defined and avoiding the obstacles present in the path. Moreover, some elements have been introduced to achieve secondary objectives. The function is the sum of several terms:

$$R^t = R^t_{goal} + R^t_{fail} + R^t_{dist} + R^t_{safe} - c_{time}$$

R^t_{goal} and R^t_{fail} are *sparse* rewards: the agent receives a huge positive reward if it reaches the goal, a huge negative reward if it collides with an obstacle, 0 reward in any other case. The goal-reached and collision conditions are evaluated on the basis of suitably defined thresholds, ρ_{goal} and ρ_{fail}:

$$R^t_{goal} = \begin{cases} c_{goal} & \text{if } d^t < \rho_{goal} \\ 0 & \text{otherwise} \end{cases},$$

$$R^t_{fail} = \begin{cases} c_{fail} & \text{if } \min(\text{ranges}) < \rho_{fail} \\ 0 & \text{otherwise} \end{cases},$$

where d^t represents the distance of the robot from the target at the time instant t, while c_{goal} and c_{fail} are hyperparameters. Both these rewards also cause the termination of the episode.

R^t_{dist} represents instead a *dense* feedback and it is the difference of the distance of the robot to the target compared to the previous time instant, multiplied for a hyperparameter c_{dist}, so it encourages the agent to get closer to the target:

$$R^t_{dist} = c_{dist}(d^t - d^{t-1})$$

Regarding the secondary objectives, $R^t_{safe}$is a function designed to promote a safe distance ρ_{safe} of the robot from the obstacles, providing incremental negative rewards when this is not respected. In fact, even if the agent is penalized for collisions, this in general does not coincide with the maintaining of a safe distance from the obstacles [31]. Furthermore, the robot has a very limited perception of the environment around it, so it has been decided to train it in a more "conservative" way.

$$R^t_{safe} = \begin{cases} c_{safe}\left[tanh\left(\frac{\lambda}{min(ranges)+b_1}\right)+b_2\right] & \text{if min(ranges)} < \rho_{safe} \\ 0 & \text{otherwise} \end{cases},$$

where c_{safe}, λ, b_1, and b_2 are hyperparamters.

The last term, c_{time}, is a penalty constant applied at each time instant, that encourages the robot to reach the target in the smallest amount of time.

3.2 PPO Algorithm

Proximal Policy Optimization is a state-of-the-art Reinforcement Learning algorithm that has shown impressive performance on a wide range of different tasks in recent years [39–42]. It is a model-free, online, on policy, policy-gradient algorithm that can work with either discrete or continuous observation and action spaces. More accurately, PPO refers to a family of algorithms; however, it is common to use the term to refer directly to the most performing version, the "*clipped*" variant, that is also the one that has been used in this work. PPO is a descendant of the Trust Region Policy Optimization (TRPO) algorithm [43], that has been designed to overcome the main problem of the policy-gradient algorithms: the "destroying" effect that large policy update makes on the policy. On the heels of its predecessor, PPO is based on the idea of ensuring gentle policy updates, but it is also simple to implement, sample efficient, and it requires minimal hyperparameter tuning [39]. More in detail, PPO exploits an actor-critic structure where the policy, represented by a neural network, called the *actor*, is trained concurrently to an advantage estimate function, that estimates the value of the selected action in the current state, and it is represented by the second neural network, called the c*ritic*.

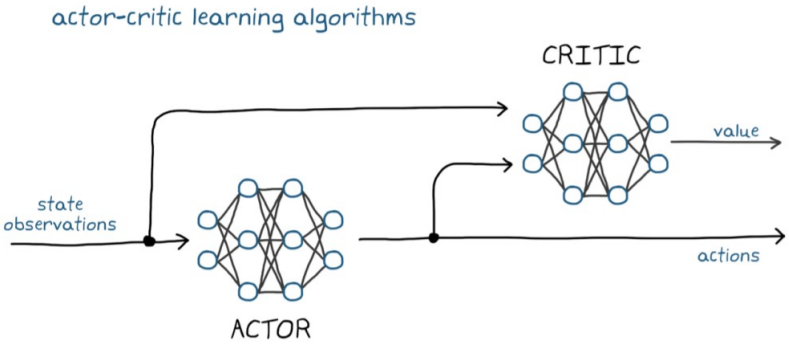


Figure 3.1: Representation of the actor-critic structure [35].

Basically, the algorithm works alternating two operations: the sampling of the data obtained online by interacting with the environment, and the optimization of

a couple of objective functions [44]. Fundamental concepts in the algorithm are the *policy probability ratio* and the *advantage estimate function*:

- The first compares the probability of selecting specific action a_t in state s_t before ($\pi_{\theta_{old}}$) and after (π_θ) the policy update:

$$p_t(\theta) = \frac{\pi_{\theta_{old}}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}$$

- On the other hand, the *advantage function* $A^\pi(s_t, a_t)$ measures whether the specific action a in state s is better or worse than randomly selecting the action according to the policy π. This function is unkwown and it has to be estimated, and this procedure can be done with different methods. The one suggested by the original authors of PPO, the Generalized Advantage Estimator [45], involves the difference between the return G_t and the value function baseline $V(s_t)$.

These two elements are directly used in the objective function, that is approximately maximized at each iteration:

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t],$$

where the expectation $\hat{\mathbb{E}}_t[...]$ indicates the empirical average over a finite batch of samples. The *clip* function is intended to bound the update of the policy parameters θ to avoid that the new policy "moves" far away from the old policy. ϵ is a parameter between 0 and 1, typically 0.2. Furthermore, this function can be modificated with the addition of an *entropy term*, H_t, that increases agent exploration.

In parallel, to learn the critic parameters ϕ, the mean squared difference between the state-value function estimate and the return is minimized [42, 44–46]:

$$L_t^{VF}(\phi) = \hat{\mathbb{E}}_t[(V_\phi(s_t) - G_t)^2]$$

Stochastic gradient descent algorithms, typically Adam Optimizer [47] are used to otpimize the two functions and learn the network parameters.

The specific implementation of the PPO used in this work, offered by Reinforcement Learning Toolbox of MATLAB, is represented by the following pseudocode:

Algorithm 3.1 PPO algorithm [48].

Input: initial actor parameter values θ_0, initial critic parameter values ϕ_0
for training episode j=1,2,3,... **do**
  repeat
    Generate N experiences by following the policy $\pi_{\theta old}$.
    Compute the return G_t and the advantage estimate $\hat{A}_t$ for each episode step.
    for learning epoch k=1,2,3,... **do**
      Sample from current experiences set a random mini-batch data set of M elements.
      Update the critic network parameters θ by minimizing the loss L_{critic} across all sampled mini-batch data with any optimization algorithm:

$$L_{critic}(\phi) = \frac{1}{M}\sum_{i=1}^{M}(G_i - V_\phi(s_i))^2$$

      Update the actor network parameters ϕ by minimizing the loss L_{actor} across all sampled mini-batch data with any optimization algorithm:

$$L_{actor}(\theta) = -\frac{1}{M}\sum_{i=1}^{M} min(p_i(\theta)\hat{A}_i, c_i(\theta)\hat{A}_i) + H_i$$

$$c_i(\theta) = max(min(p_i(\theta), 1+\epsilon), 1-\epsilon)$$

    end for
  until the training episode reaches a terminal state
end for

3.3 Neural Network Architecture

As described in the previous section, the PPO agent is based on an actor-critic representation. Therefore, it is necessary to design two Neural Networks. The first, the critic, takes in input the observation vector and outputs a single number, that represent the value of the action chosen by the actor. This network has been designed using two hidden layers of N_{c1} and N_{c2} neurons. The second is the actor, that represents the policy. The input is again the observation vector, however the output it produces is a vector that contains the probability of executing each of the possible couple of velocities defined in the action space. To obtain this behaviour, a softmax has been used as an activation function in the output layer. As the previous, the network contains two hidden layers with N_{a1} and N_{a2} neurons. All the hidden layers have been designed as Fully Connected (FC) layers, i.e., the neurons have full connections to all the activations of the previous layer [23], using ReLu as activation functions.

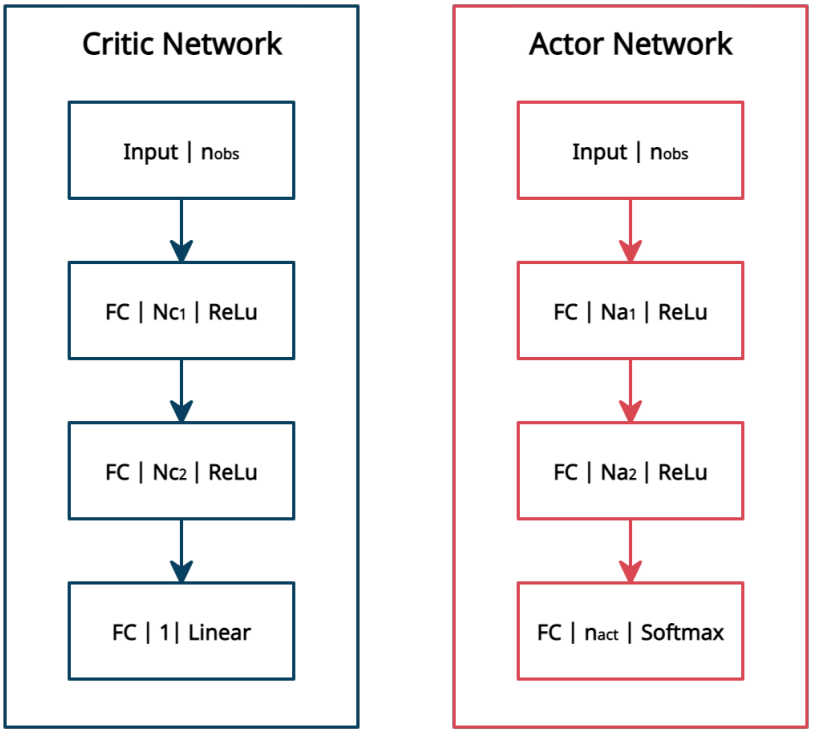


Figure 3.2: PPO Critic and Actor Neural Networks scheme.

3.4 Multi-Stage Training

Once the different elements that characterize the Reinforcement Learning problem have been designed, the agent is substantially a Neural Network characterized by initial random weights. Then, it needs to learn the desired skills by updating the weights through the learning algorithm during the training. It has been showed that, as humans and animals, reinforcement learning agents learn much better when they are subject to problems of incremental complexity [49]. On the basis of these considerations, the agent has been trained using a multi-stage approach, starting from environment characterized by elementary complexity and changing the environment toward more complicated ones where the agent performance is sufficiently satisfactory. In fact, it is known the sensitivity of the neural networks with respect to initial values of their weights [50,51]. Hence, this kind of approach can accelerate the learning phase and allow to obtain higher rewards [30]. Furthermore, the randomness has been considered a key element for the training: in each episode of the training phase, the starting position, starting orientation, and the target, as well as the positions of the obstacles in some of the environments, are randomly defined, making the agent enable to explore the observation space and to acquire a robust policy.

More in depth, the training conceived firstly involves a simple environment with-

out obstacles, in which the agent can learn the ability to reach the target. At each episode, the agent interacts with the environment until it reaches the goal, collides with an obstacle, or after a specific number of time steps, like a time-out. When it achieves a satisfactory level of performance, measured averaging the rewards it collects in a episode for a certain number of episodes, the training phase is stopped. By preserving the learned neural network parameters, it is then resumed in a more complex environment, where the agent can gain, and later affine, collision avoidance abilities. This process is repeated for a certain number of stages, characterized by gradually more complex environments, with an increasing number of obstacles and a decreasing operating space in between.

Chapter 4

Hybrid Control Architecture

4.1 Introduction

The beating heart of the collision avoidance algorithm implemented in this work is represented by the PPO agent introduced in Chapter 3. As it will be exposed in Chapter 5, it shows powerful generalization abilities and it is able to successfully face complex scenarios. However, it shows limited performance when dealing with some trivial situations: an example is that, if the robot is asked to reach a target ahead of it in total abscense of obstacles, most likely it will reach the goal following a pointless curve trajectory supplemented by narrow but frequent directions changes, rather than acting through a simple straight line. This limitation, shown in other researches, for example [30], it is mostly due to the stochastic nature of the PPO policy, that computes the probability of executing each of the possible actions given the observation vector as input. The second reason is that the agent is trained in this kind of situation only in the first part of the training phase, and then it will not face anymore situations where it it expected to act following straight trajectories. Hence, roughly speaking, the agent is not trained to behave in this manner, and even if it would, it always act in a stochastic way.

Since this kind of situations can be easily tackled with traditional control techniques, the aforementioned research used a PID controller to overcome this problem, that has been integrated in a switching architecture that can act using the RL stochastic policy or the PID depending on the situation detected. Inspired by this research, the idea in this work is to train a second reinforcement learning agent, characterized by a deterministic policy, and to switch between the two agents depending on the specific situation approached. Deep Deterministic Policy Gradient (DDPG) has been chosen as the learning algorithm for this second agent due to its deterministic nature, ease of implementation and generalization ability.

4.2 DDPG Agent

4.2.1 DDPG Algorithm

The Deep Deterministic Policy Gradient algorithm [52] is a model-free, online, off-policy reinforcement learning algorithm. It has been designed specifically for continous action spaces; however, it can work with both continuos or discrete observation spaces. It is based on an actor-critic configuration: the actor network, that represents the current policy $\pi_\theta(s)$, deterministically maps states into actions, and the critic $Q_\phi(s,a)$ is learned using the Bellman equation. The objective for the policy function is to maximize the expect reward, and the actor weights θ are updated in the direction of the performance gradient, using the *policy gradent theorem* [52, 53]:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &\approx \mathbb{E}\left[\nabla_\theta Q_\phi(s,a)|_{s=s_t,a=\pi_\theta(s_t)}\right] \\ &= \mathbb{E}\left[\nabla_a Q_\phi(s,a)|_{s=s_t,a=\pi(s_t)} \nabla_\theta \pi_\theta(s)|_{s=s_t}\right]\end{aligned}$$

The idea is to apply the chain rule to the optimal function J, obtaining the gradient of the critic output with respect to the policy weights $\nabla_\theta Q_\phi(s,a)$ by computing the product of the gradient of the critic output with respect to the action taken by the actor $\nabla_a Q_\phi(s,a)$ and the gradient of the actor output with respect to the policy weights $\nabla_\theta \pi_\theta(s)$ [54].

The key features of the DDPG are the replay buffer, the target networks, and the exploration noise:

- The replay buffer is a cache memory with a fined size, used to store the transitions $(s_{t,}, a_t, R_t, s_{t+1})$ sampled from the environment acting according to the exploration policy. The update of the networks is performed by randomly extracting a minibacth of samples from the buffer.

- The target networks represent a copy of the actor and critic networks, $Q'_{\phi'}(s,a)$ and $\pi'_{\theta'}(s)$, and they have been implemented to overcome a problem of the Q-learning approach: this algorithm, commonly used until a few years ago, updates the parameters ϕ of the Q function approximator by minimizing the following function:

  $$L(\phi) = E\left[\left(Q_\phi(s_t,a_t) - y_t\right)^2\right],$$

  where

  $$y_t = R(s_t,a_t) + \gamma Q_\phi(s_{t+1}, \pi(s_{t+1}))$$

  The problem is that y_t is itself a function of Q_ϕ and this, in practice, leads to unstable learning [52, 55]. In DDPG, the computation of y_t is performed using $Q'_{\phi'}(s,a)$ and $\pi'_{\theta'}(s)$, whose weights are updated in a "soft" delayed manner with respect to their original counterparts. This slow change in the target

values improves the stability of learning.

- The exploration noise is essential to enable the agent to explore: the exploration policy is obtained by adding a noise process $\mathcal{N}$ to the deterministic policy. In the implementation proposed by the authors [52], the noise is modelled as an Ornstein-Ulhenbeck process [56], that is a stochastic process that as times goes on dritfs toward its mean function [57].

The specific implementation of the PPO used in this work, offered by Reinforcement Learning Toolbox of MATLAB, is represented by the following pseudocode:

Algorithm 4.1 DDPG pseudocode [54].

1: Input: initial actor parameter values θ_0, initial critic parameter values ϕ_0
2: Initialize target networks Q' and π' with weights θ'_0=θ_0 and ϕ'_0=ϕ_0
3: **for** training time step t=1,2,3,... **do**
4: Select the action $a_t = \pi_\theta(s_t) + N_t$ depending on current exploration policy
5: Execute action a_t, observe the reward R_t and the next observation s_{t+1}
6: Store the experience collected (s_t, a_t, R_t, s_{t+1}) in the replay buffer
7: Randomly extract a minibatch of M experiences (s_i, a_i, R_i, s_{i+1}) from the buffer replay
8: Set $y_i = R_i + \gamma Q'_{\phi'}(s_{t+1}, \pi'_{\theta'})$
9: Update the weights of the critic network by minimizing the loss L across all sampled mini-batch data:

$$L_(\phi) = \frac{1}{M}\sum_{i=1}^{M}(y_i - Q_\phi(s_i, a_i))^2$$

10: Update the weights of the actor network by using the sampled policy gradient:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{M}\sum_{i=1}^{M}\nabla_{\pi_\theta(s_i)}Q_\phi(s_i, \pi_\theta(s_i))\nabla_\theta\pi_\theta(s_i)$$

11: Update the target networks parameters depending the smoothing factor τ:

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

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

12: **end for**

4.2.2 Observations, Actions, and Rewards

The observation vector for the agent is made up of a single value: the angle between the target and the robot reference frame. Even if the lack of knowledge of the target distance could seem a limitation, the agent has been trained to quickly point toward the goal by rotating in a quasi-static manner, being able to approach the target direction in a very marginal space.

Again, the actions are permissible couples of linear and angular velocities of the robot. However, in this case, the action space has to be continuous, hence translational and rotational velocities can be any value inside, respectively, the ranges $[0, v_{max}]$ and $[-w_{max}, w_{max}]$, defined according to real robot specifications. Hence, the dimension of the action space in this case is 2.

The reward function used for this agent is:

$$R^t = R^t_{goal} + R^t_{fail} + R^t_{dir},$$

where R^t_{goal} and R^t_{fail} are the same terms presented for the PPO agent, except for the fact that R_{fail} is assigned when the robot touches the borders of the room, since it has no access to any range measurement, while the last term is defined as:

$$R^t_{dir} = c_{dir} \left[tanh\left(\frac{\lambda}{\varphi}\right) + b \right] v^{t-1},$$

where φ is the angle between the goal and the robot heading and c_{dir}, λ, and b are hyperparameters. As introduced before, the idea behind this function is to instruct the robot to rotate almost statically toward the goal and to increase the linear velocity only when the heading direction is close to point toward the goal.

4.2.3 Training Environment

The agent has been trained in a simple environment, that emulates a room with fixed walls but without obstacles in the middle. At each episode, the starting pose of the robot and the target position are randomly generated. Actually, to ensure a partial consistency in the amount of time needed for reaching the target and in the total reward collectable in the epidose, one of the coordinates that represent the initial and target position of the robot has been kept fixed. Therefore, the random generation of one coordinate of the starting position and one coordinate of the goal position, together with the random initial orientation, ensures enough exploration of the observation space from the agent. The episode finishes when the robot either reaches the target, collides with a wall, or after a time-out.

4.2.4 Neural Network Architecture

In this case, the algorithm is based on an actor-critic architecture, therefore it is necessary to design two neural networks. The critic takes the observations and the actions as input, as well as the estimation of the action-value function as output. It has been designed using two different input branches, one for the observation vector and the other for the action taken, connected to two separate FC hidden layers of N_{c1} and N_{c2} neurons. The two layers are merged together using a third FC layer of N_{c3} neurons, whose output is the estimate of the Q value. On the other hand,

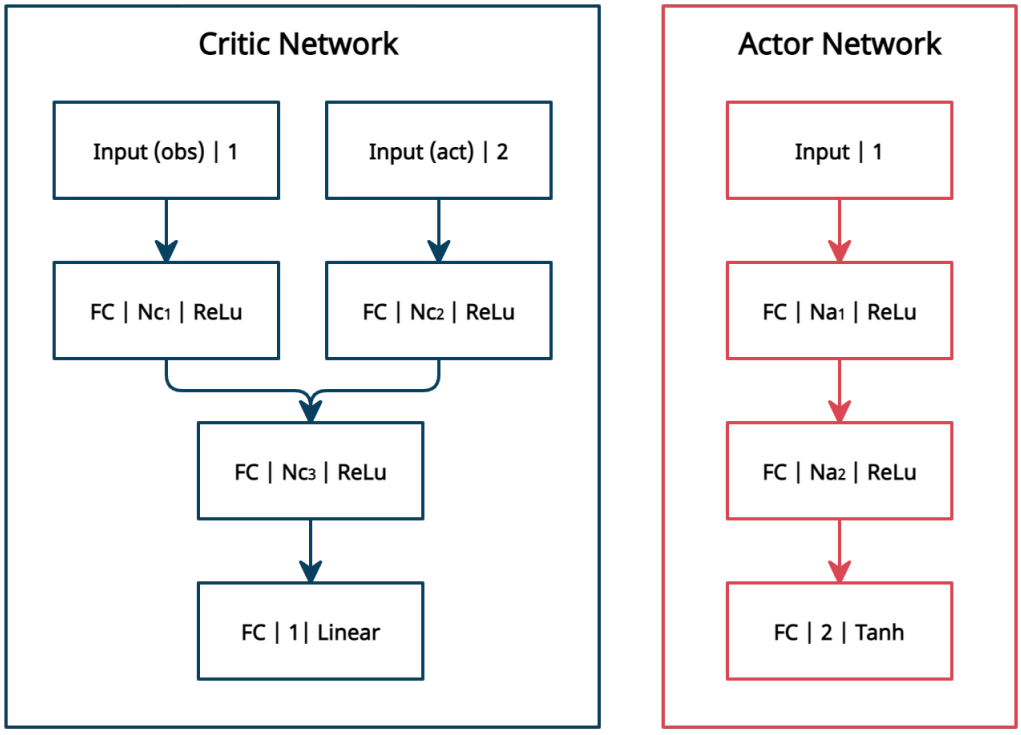


Figure 4.1: DDPG Critic and Actor Neural Networks scheme.

the actor takes only the observation vector as input, that is processed through two FC hidden layers of N_{a1} and N_{a2} neurons, connected to the output layer composed of 2 neurons, one for each dimension of the action space. This layer is activated by a hyperbolic tangent function, therefore the output values are consequently scaled with respect to the ranges considered in the action space. Lastly, all the activation functions used in the hidden layers of both the networks are rectified linear units.

4.3 Scenario Classification

The idea behind the concurrent implementation of two different reinforcement learning agents is related to the development of a control algorithm that takes robot sensor data as input, decides which agent to exploit, and produces the action of the robot as output. As will be explained in depth in Chapter 5, at each sampling time, based on robot sensor measurements, through a simple "if" loop, the action is computed using one of the two agents. More specifically, if the target is inside the horizontal Field Of View (FOV) of the camera, and if the path towards it is obstacle-free, the action is taken by the deterministic DDPG agent, otherwise it is the stochastic PPO agent that decides what to do.

Chapter 5

Implementation Details

5.1 Hardware Details

The training sessions of the agents, as well as MATLAB and Gazebo simulations, have been performed using a laptop with an AMD Ryzen 7 4800H CPU and 16 GB RAM. The target robotic hardware is a ground robot developed by the Department of Mechanical and Aerospace Engineering (DIMEAS) of Politecnico di Torino. It has been developed out of a commercial robot kit (Devastator Tank Mobile Robot Platform), and the hardware it is equipped with, together with its velocity limits, are presented in Table 5.1.

As reported in the aforementioned Table, the robot is equipped with a Intel Real Sense D435, that is a stereo vision depth camera system. It exploits two imagers at a certain distance to capture the scene from different perspectives, and a vision processor that correlates the points in the two images to obtain a depth value for each pixel [58]. These data are used to produce a depth map of the scene detected by the camera. The essential features of the sensor, needed to model the interaction between the agent and the environment, are presented in Table 5.2.

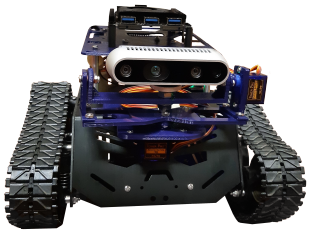
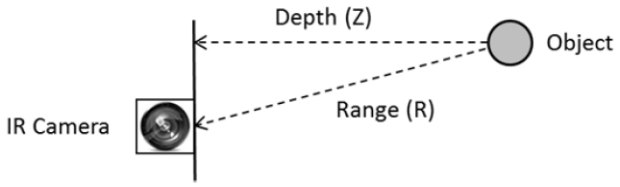


Figure 5.1: DIMEAS ground robot.

Figure 5.2: Depth measurment versus range [58].



5.2 PPO Agent Implementation

Both the reinforcement learning agents have been implemented and trained by modelling the robot as a *point particle,* that moves according to the kinematic model presented in Section 2.4, using MATLAB. Nevertheless, each of the two agents has been specifically designed for the task it has to solve, and in this section there will be presented the details of the design procedure adopted for the PPO agent.

5.2.1 PPO Agent Details

The PPO agent, as introduced in Section 3.1, takes as input a number n_{ranges} of range measurements, together with the position of the target in polar coordinates. To obtain a trade-off between the accuracy of the sensing information and a small number of inputs, hence a fast learning phase, it has be decided to use $n_{ranges} = 13$. The idea is to convert the depth data from the camera into range measurements, and then divide the Horizontal Field of View covered by the camera into 13 angular sectors, by taking the minimum range in each sector. In this way, the agent is provided with a knowledge of the distance from obstacles in the area in front of it. In the implementation and training of the agent, the data used for the composition of the observation vector has been generated by simulating range-bearing sensor readings. The properties of the simulated sensor have been designed to emulate the behaviour of the real camera. However, the minimum and maximum ranges of the real camera, that are crucial values, depend on the lighting conditions. Hence, it has been decided to take this information by considering an accurate Gazebo model of the robot, developed by the DIMEAS, that it has also been used to verify the behaviour of the robot in a realistic framework, as it will explained in Section 6.2. By performing some tests with the model, it has been observed that the range of the simulated camera is approximately from 0.3 m to 4 m with respect to robot center (the camera is mounted in the front side of the robot), and the horizontal FOV results slightly lower than 86°. Therefore, the properties of the simulated range-bearing sensor have been tuned to $[0.3\,m, 4\,m]$ range and 85° as horizontal FOV. Furthermore, the range measurements are normalized from range [0.3, 4] to [0, 1] before to be used to compose the observation vector.

Table 5.1: DIMEAS ground robot features.

On-Board Computers	
Lattepanda Delta 432 (Companion PC)	Intel 8th Gen Celeron Processor N4100 (1.1-2.4 GHz Quad-Core, Four-Thread)
	Intel UHD Graphics 600, 200-700 MHz
	4G LPDDR4 2400 MHz Dual-Channel
	Xubuntu 20.04 with ROS
NXP FRDM K64F (Microcontroller)	ARM Cortex-M4 32-bit single core 120 MHz
	1024 KB program flash memory
	256 KB RAM
	Mbed OS
Sensors	
Intel Real Sense D435	Depth Camera Sensor
IMU 9DoF	Accelerometer + Gyroscope + Magnetometer
Velocity Limits	
Maximum Linear Velocity	0.1 m/s
Maximum Angular Velocity	0.115 rad/s

Due to the relatively low maximum velocities of the robot, it has been designed a set of $n_{act} = 11$ discrete couples of velocities, and experiments to further fractionate the velocities values have led to inconclusive results. In practice, when the robot needs to move forward, it is pointless to use a velocity lower than the maximum linear speed v_{max}, and when it needs to rotate, due to the very limited ω_{max}, it is to convenient to encourage it to make the maneuver with the minimum possible linear speed (0 in MATLAB simulation), and the maximum angular velocity ω_{max}. The complete set of possible actions actuable by the agent is presented in Table 5.3 , where $v_{max} = 0.1\, m/s$, and $\omega = 0.115\, rad/s$.

Regarding the reward function, the designed coefficients are presented in Table 5.4.

About the neural networks, it has been decided to take $N_{c1} = N_{c2} = N_{a1} = N_{a2} = 100$. Furthermore, an important hyperparameter to tune is the learning rate for both the networks: as explained in Section 3.2, the actor and critic weights are updated by minimizing a couple of loss functions; more in depth, the gradient of a loss function is computed to derive the steepest direction of decrease of the function itself [23], and the learning rate measures the size of the step, hence how

Table 5.2: Intel Real Sense D435 features.

Parameter	Value
Horizontal FOV	86°
Vertical FOV	57°
Nominal range	0.2 m to 10 m
Ideal range	0.3 m to 3 m

Table 5.3: PPO actions.

v	ω
v_{max}	0
v_{max}	$\pm 0.3\,\omega_{max}$
v_{max}	$\pm\omega_{max}$
$0.3\,v_{max}$	$\pm 0.3\,\omega_{max}$
$0.3\,v_{max}$	$\pm\omega_{max}$
0	$\pm\omega_{max}$

much the weights of the networks are changed with respect to previous values at each iteration. In practice, a small learning rate leads to long training phases, while an high rate increases the probability to be trapped in local minima. In this case, it has been decided to take a learning rate equal to 0.001 for both actor and critic.

Finally, the parameters of the learning algorithms have been tuned: the agent has been designed to collect experiences with a sample time $Ts = 1s$ for the whole duration of the episode, since the experience horizon has been taken equal to the final time instant of the episode, 500 s. In fact, the experience horizon is the number of experience steps that the agent collects before learning, unless the episode terminates before, and then it is trained by sampling a random batch of 128 elements for 3 epochs. The discount factor γ has been tuned to 0.998, while for the entropy loss weight, that modulates the relevance of the entropy term H_t, and for the clip factor ϵ, there have been taken the values used by the original authors [44], i.e., 0.01 and 0.2 respectively. Even the method for the computation of the advantage function has been taken according to the choice of the original authors, i.e., the Generalized Advantage Estimator (GAE) [45], with a GAE factor equal to 0.95. In Table 5.5, all the hyperparameters of the learning algorithm are resumed.

Table 5.4: PPO reward function hyperparameters.

Parameter	Value
c_{goal}	50
ρ_{goal}	$0.1\,m$
c_{fail}	-50
ρ_{goal}	$0.35\,m$
c_{dist}	3
c_{safe}	-10
λ	0.1
b_1	-0.3
b_2	-0.2
ρ_{safe}	$0.6\,m$
c_{time}	0.1

Table 5.5: PPO algorithm hyperparameters.

Parameter	Value
Ts	1
Experience horizon	500
Mini-batch size	128
Number of epochs	3
Discount factor	0.998
Clip factor	0.2
Entropy loss weight	0.01
GAE factor	0.1

5.2.2 PPO Agent Training

Randomness is a key element in the training of a reinforcement learning agent. Indeed, randomness rules the initialization of neural network weights, as well as the sampling of the minibatch data used for updating the same weights, and usually even the environment is related to a stochastic component. Furthermore, the PPO policy is based on the stochastic choice of an action, depending on a probability distribution. Actually, it has been showed that the variation of the random seed among different trials of the same learning process can drastically vary the performance of the algorithms [50]. This represents a huge problem for the achievement of reproducible results [59]. To mitigate this problem, and to better verify the converge ability of the tuned algorithm, as well as the performance of the resulting agent, it has been decided to reproduce the training phase 5 times, varying the random seed. The seed has been fixed before the initialization of the neural networks, hence there have been considered 5 different initial conditions for the neural networks. Moreover, to verify and show an example of the high variability of the learning procedure, that includes the initialization of the weights but goes also beyond it, the first run of the learning process has been repeated for a second time, with the same network weights, but a random generator out of phase with respect to the previous. Actually, after the initialization of the neural weights, before to start the learning process, the random generator has been exploited to produce different results with respect to the first trial.

The whole training process for the PPO agent in this work has been based on a multi-stage procedure, made up by different environments of incremental complexity, that has been showed to ensure faster converge and higher results. In each stage, the training is performed until the agent gets an average reward of 50, measured in the last 100 episodes. When it achieves this level of performance, the agent faces the next, more complex, environment:

1. The first stage, the one the agent deals with when it is nothing more than a random initialized neural network, is an obstacle-free room surrounded by walls. The idea is to provide the agent the ability to exploit the inputs it receives to reach the target. To simultaneoulsy obtain a generalized policy and partially uniform the maximum reward the agent can achieves, at each episode the starting orientation of the robot is randomly generated, while the starting and target position are randomly chosen from two lines, as illustrated in Fig 5.3.

2. When the agent has gained the ability to reach a target, it is the moment for learning to overcome obstacles. For this reason, the second stage resembles the same environment presented in stage 1, but with the generation of a wall of length 0.5 m . To make things more complex, and to ensure the usually desired

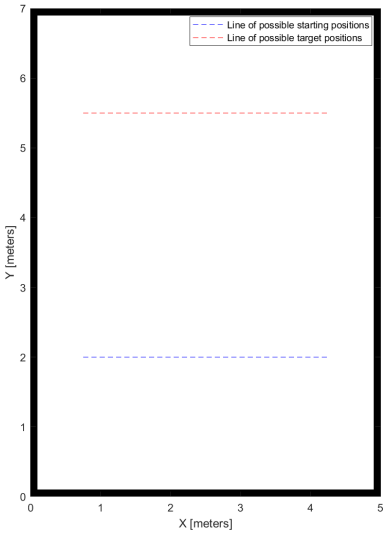


Figure 5.3: First training stage for the PPO agent.

generalization ability, the target is always generated in front of the robot, but the wall is generated in the middle of the path. Actually, the starting position of the wall can vary, resulting in some episodes with the wall closer to the robot than others, as wells as episodes with the wall slightly moved with respect to the left or the right direction.

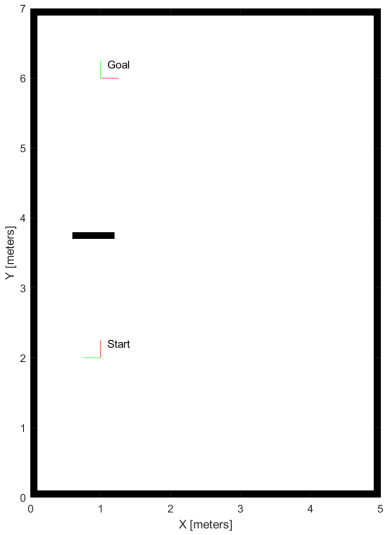


Figure 5.4: An example of the second training stage for the PPO agent.

3. In the third stage, two couples of horizontal walls are generated at fixed y-positions in the room; however, the x positions of the walls, as well as their lengths, can vary depending on random initial conditions. Furthermore, this stage adopts the same random generation for the starting pose and target position explained for the first training environment.

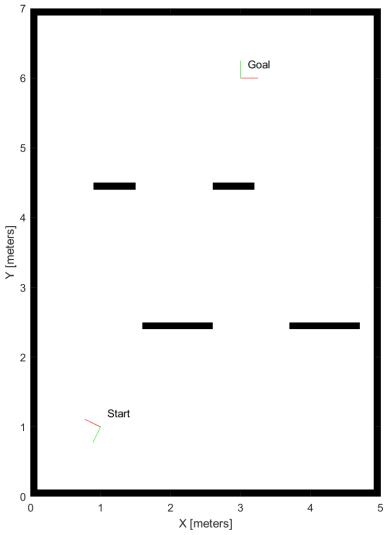


Figure 5.5: An example of the third training stage for the PPO agent.

4. The fourth stage replicates the philosophy of the previous one, but with 4 couples of walls.

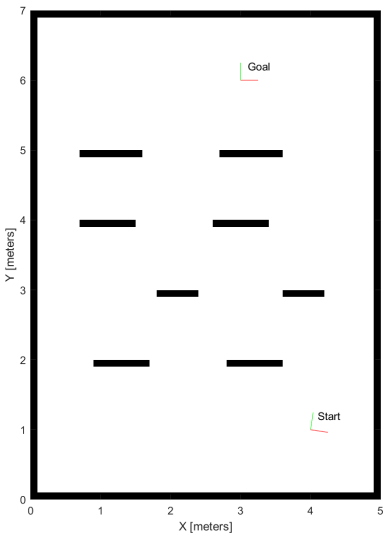


Figure 5.6: An example of the fourth training stage for the PPO agent.

5. The last stage, is quite different from the previous ones, since it involves a specific fixed obstacles environment. Hence, in each episode the walls are in the same manner as presented in Fig 5.7. In any case, the generalization is ensured by the random generalization of the initial and target conditions explained in the first stage and hence characteristic of each scenario except for the second.

As stated, this procedure has been repeatead 6 times. After the completition of the

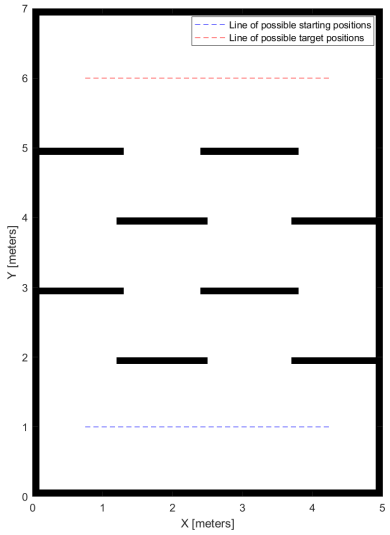


Figure 5.7: Fifth training stage for the PPO agent.

training, hence, after the achievement of the 50 average reward in the final stage, the resulting agents show great collision avoidance abilities, that cannot be distinguished through visual comparisons of individual trajectories. Consequently, they have been evaluated by measuring their performance when facing the fourth and fifth training environments for 1000 episodes.

5.2.3 Evaluation and Testing

As expected, the training phase of the agents varies depending on the random seed. For some unkown reasons, the fourth run failed in the first stage: the training has been stopped after 8000 episodes, while in the other runs the level of 50 average reward was reached between 238 and 578 episodes. However, this problem has been found also in [59], where 2 runs out of 10 of their PPO configuration with different random seeds failed. Nevertheless, in this case, the other 5 runs arrived to the completition of the final stage, and the performance of the resulting agents have been measured by testing each agent for 1000 episodes in the environment n°5 and for 1000 episodes in the environment n°6. The performance in the fifth stage have been considered due to the huge variability of the environment, useful to test the generalization ability of the agents. The drawback of this environment is that its difficulty can vary significantly depending on the randomness; however, 1000 tests can allow to make comparisons between the different policies. Actually, even if the agents have been trained in the same way, and even if they seem to act in the same way, they are related to stochastic policies represented by neural networks with different weights, hence with different input-output probability distributions.

The huge variability in training phases due to stochasticity is observable in Table 5.6, where the number of learning episodes before the achievment of the 50 average

reward in each stage is presented, and in Table 5.7, that presents the success rates of the agents after training completition. Since the goal of the work is to deploy a reinforcement learning agent to develop a collision avoidance algorithm, there have been chosen two policies for more accurate validation tests, and they are the ones related to run 1B and run 3. Furthermore, an example of successful trajectory of the robot in stage 4 and 5 is shown in Fig 5.8.

Table 5.6: PPO training episodes for each stage.

Run	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1A	578	969	2013	9377	3658
1B	243	653	1126	8688	7444
2	293	482	627	9736	8212
3	568	2174	1382	14896	6061
4	-	-	-	-	-
5	238	4366	3575	19186	7433

Table 5.7: PPO success rates in stage 5 and 6.

Run	Success Rate in Stage 5	Success Rate in Stage 6
1A	95.8%	92.8%
1B	99.8%	86.4%
2	93.1%	82%
3	98.8%	92.2%
4	-	-
5	99.4%	87.1%

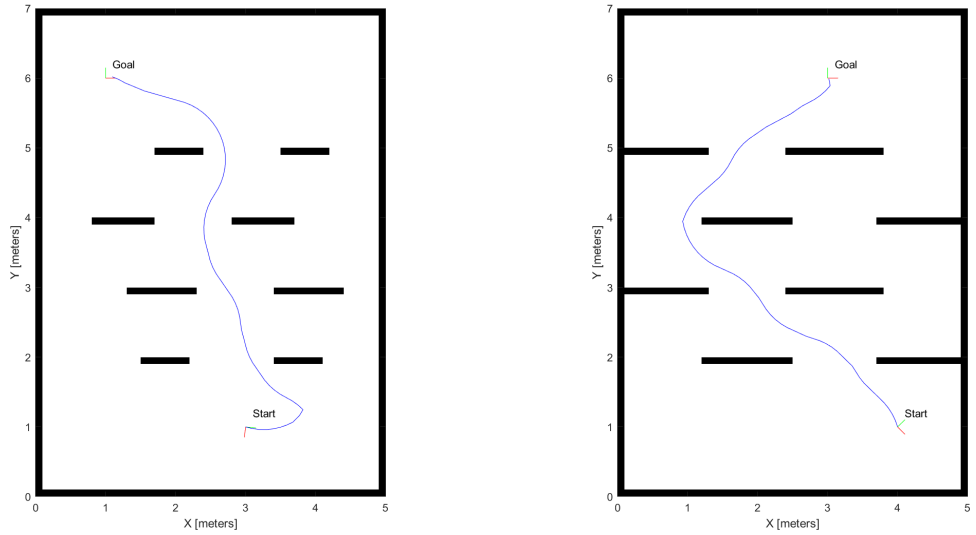


Figure 5.8: PPO successful trajectories in stages 4 and 5.

Finally, to provide a visual information about the training phase, Fig 5.9 displays the different learning curves for the 6 runs in the first stage, while Fig 5.10 and Fig 5.11 present the complete learning curves for the policies chosen for the validation phase, compared with their counterparts obtained by training the agent without the multi-stage approach, i.e., by using only the final training scenario after the networks initialization. The comparison shows the glearing benefits for choosing the adopted multi-stage paradigm.

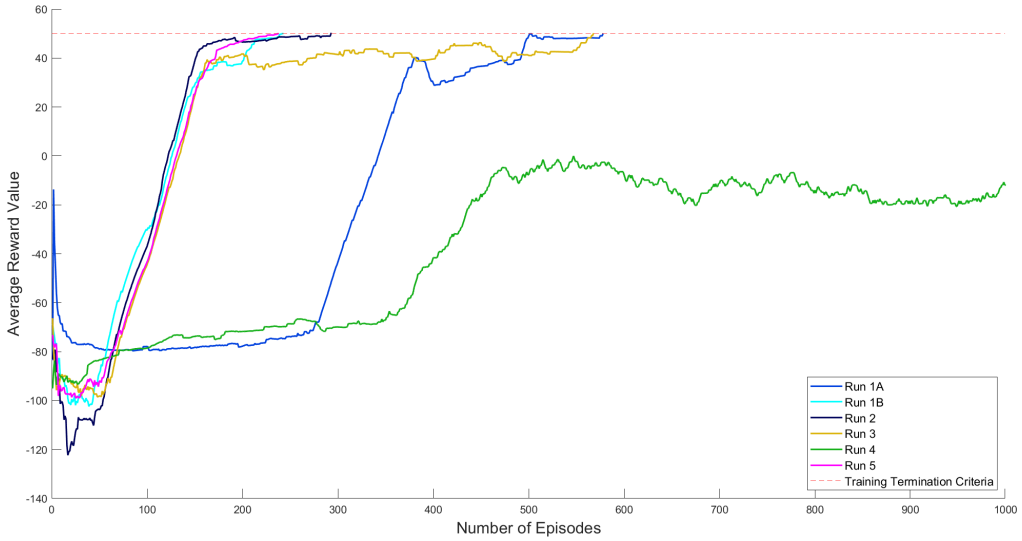


Figure 5.9: Learning curves of PPO agents related to stage 1.

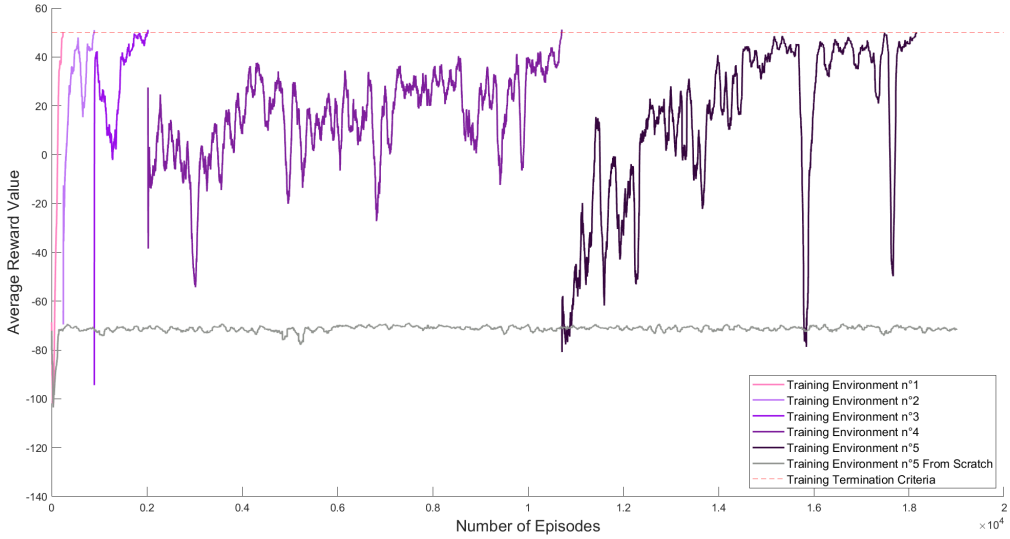


Figure 5.10: Comparison between average rewards obtained through multi-strage training and training from scratch in PPO run 1b.

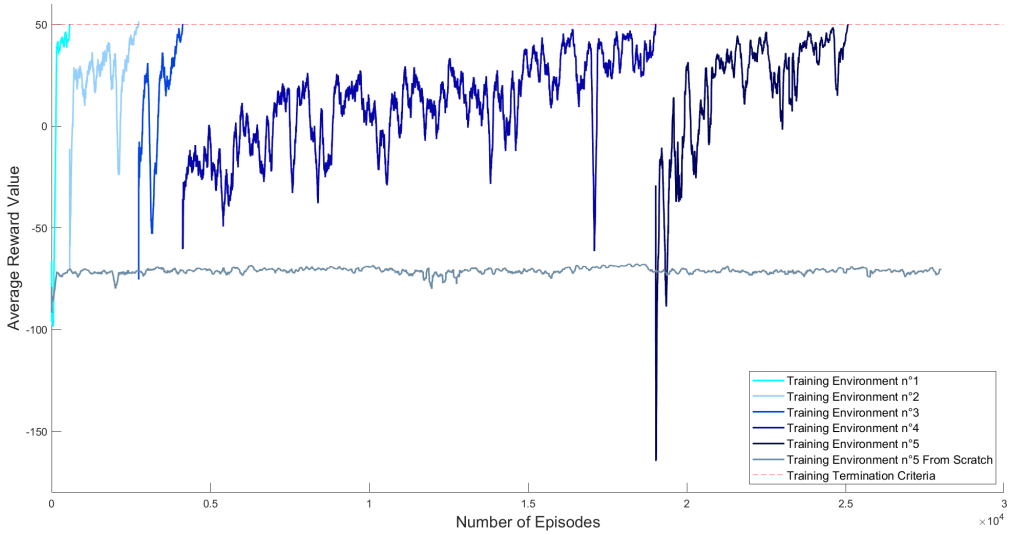


Figure 5.11: Comparison between average rewards obtained through multi-strage training and training from scratch in PPO run 3.

5.3 DDPG Agent Implementation

5.3.1 DDPG Agent Details

The DDPG agent takes as input the angle of the target with respect to the robot coordinate frame, and outputs a couple of values that represent a linear velocity in the range $[0, 0.1\, m/s]$, and an angular velocity in the range $[-0.115\, rad/s, 0.115\, rad/s]$. The hyperparameters of the reward function are presented in Table 5.8.

The neural networks have been designed by picking $N_{c1} = N_{c2} = N_{c3} = N_{a1} =$

Table 5.8: DDPG reward function hyperparameters.

Parameter	Value
c_{goal}	50
ρ_{goal}	$0.1\,m$
c_{fail}	-50
c_{dir}	10
λ	0.1
b	-0.5

Table 5.9: DDPG algorithm hyperparameters.

Parameter	Value
Ts	1
Experience Buffer Lenght	10^6
Mini-batch size	64
Discount factor	0.99
Smoothing factor	0.001

$N_{a2} = 30$, and a learning rate equal to 0.01 for the critic network, and to 0.001 for the actor. Additionally, a gradient threshold of 1 has been implemented for both the networks: its function is to clip the gradient when it overcomes a threshold.

Finally, the learning algorithm hyperparameters are presented in Table 5.9. Regarding the exploration noise, it has been used an Ornstein-Ulhenbeck exploration noise model with 0 mean, 0 initial action, 0.1 variance and 10^{-5} decay rate of the standard deviation [60].

5.3.2 DDPG Agent Training

The DDPG agent has been trained using a single training environment, that is the one correspondent to stage 1 in the PPO agent training, considering a maximum number of steps for episode equal to 100. The training phase is stopped when the agent achieves an average reward of 60, measured in the last 50 episodes. Due to the lower complexity of the task of the DDPG agent, and hence a straightforward training phase, in this case it has not been adopted a systematic procedure with parallel learning procedures as in PPO. However, the λ coefficient has been finely tuned to obtain the desired performance: from further experiments with the hybrid architecture, that will be better discussed in Chapter 6, it has been observed that a too sharp variation in the trajectory of the agent commanded by the deterministic

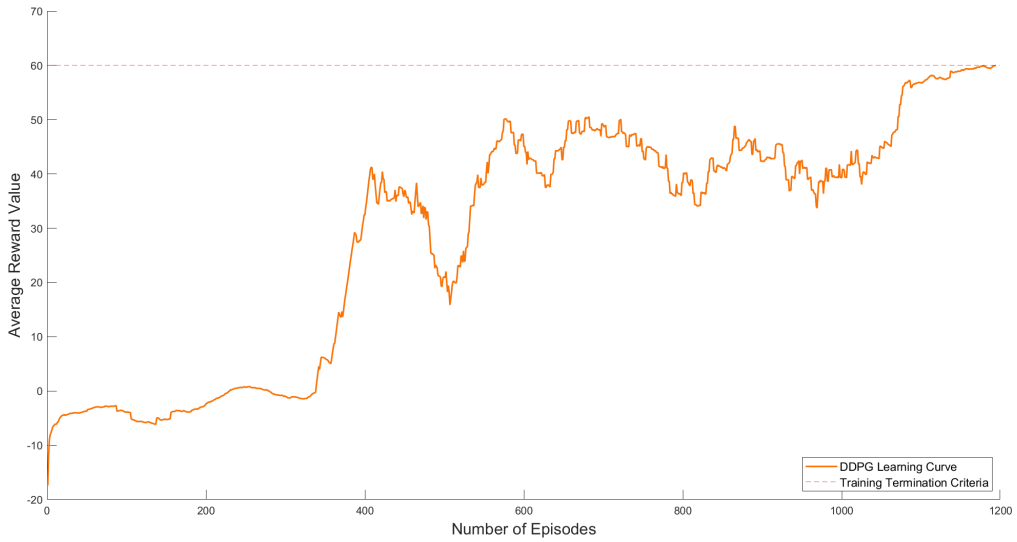


Figure 5.12: DDPG learning curve.

policy could lead to unexpected collisions. Actually, when the PPO agent is going to complete an obstacle avoidance maneuver, at a certain moment the robot cannot sense the obstacle anymore, or when it is out from the horizontal FOV of the camera, or when it is closer than the minimum range of the vision sensor. Nevertheless, it has learnt to keep a safe distance from obstacles and to perform safe and large trajectories. The problem arises when the PPO agent is not allowed to complete a safe and large maneuver because the action decision is passed to the DDPG agent, and to avoid this situation two strategies have been adopted: the addition of conservativeness to restrict the cases when the action is taken by the deterministic policy, and the adoption of a DDPG agent that makes a trajectory not too straight when it adjusts the direction to reach the target. Hence, λ has been tuned to obtain a slight curvature in the first limited portion of path. Actually, even the random seed variation can lead to slight different trajectories; however, in this case, the difference is miniscule. The learning curve of the chosen policy is reported in Fig 5.12, and the behaviour of the DDPG agent, compared with the PPO, is presented in Fig 5.13. It can be noted that the DDPG policy leads to a straightforward reaching of the target. To verify the agent ability despite its lack of knowledge of the target distance, the policy has been tested with 1000 cases in the training environment, applying the random generation of both the coordinates of the starting and goal positions, along with the initial orientation, and the goal has been reached all the times.

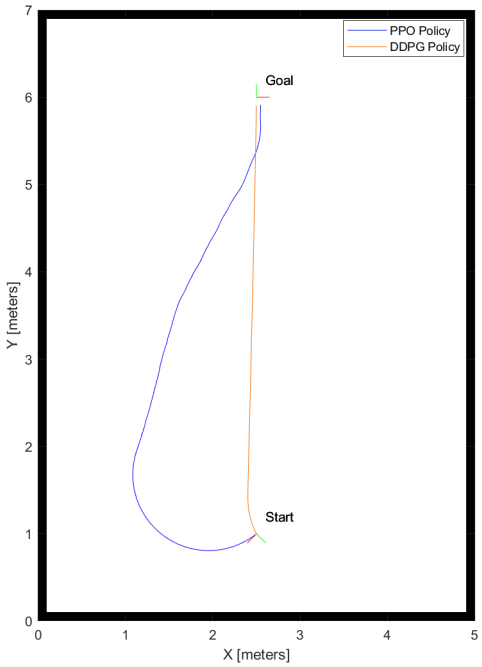


Figure 5.13: Comparison between the behaviour of the DDPG and PPO agents.

Chapter 6

Simulations

The two candidate PPO policies showed good success rates in the last two training environments, that represent challenging tests for measuring the collision ability of the robot. Consequently, they have been deployed in the hybrid collision avoidance algorithm, object of this work. The algorithm has been first implemented in MATLAB, adopting a more realistic simulation with respect to the one used for the training. In this phase, performance measurements have been finely collected to deeply evaluate the performance of both the policies and also of the algorithm. Afterwards, the control algorithm has been implemented on a GAZEBO simulator, to test the behaviour of the algorithm through a highly realistic framework.

6.1 MATLAB Simulation

6.1.1 Control Algorithm

In the validation phase, the robot has not been considered a point particle anymore, but it has been modelled as a circle of radius $r = 0.11\,m$, since the robot measures approssimatevely 21 cm x 22 cm. Furthermore, the sampling time has not been taken equal to 1 second anymore, but it has been tuned to resemble the real control frequency obtained in the Gazebo simulation. As will be presented in Section 6.2, the control frequency of the algorithm is around 2 Hz. Hence, in MATLAB simulation it has been designed the robot to take an action every 0.5 s; however, to increase the accuracy of the simulation, precise measurements about the distance from the obstacles have been measured implementing a fictitious second range bearing sensor, that sense every 0.05 s all the environment sorrounding the robot without distance limits. Obviously, the agent has not access to this information.

At the starting instance of the control algorithm, it is avalaible the starting pose of the robot and the target position. The algorithm is constitued by a while statement, that runs until the robot reaches the goal, considering a threshold of 0.10 m, collides with an obstacle, detectable when the distance from an obstacles is equal to

or lower than 0.11 m, or after a timeout at 1000 s. At each iteration of the algorithm, with a $Ts_{control} = 0.05\,s$,the pose of the robot is updated and the fictitious sensor is exploited to compute effective distance from obstacles. Nevertheless, the robot action is computed with a $Ts_{action} = 0.5\,s$, hence the same action is commanded for 10 consecutive time instants. The computation of the action is based on the construction of the observation vector, that involves the simulation of the vision sensor, the shrinking of the range measurements to 13, their normalization, and the computation of the distance and angle of the robot with respect to the target. This data is also exploited to decide the policy responsible for action computation, and the choice is based on a if statement. Basically, if the angle with respect to the target is inside the horizontal field of view of the camera, hence if the robot can sense the environment between itself and the target, and if the distance from the target is lower than the range measurement related to the angular sector that includes the target, hence the path towards the goal is obstacle-free, the action is computed through the DDPG policy, else through the PPO policy. However, after some experiments, it has been decided to consider more conservative conditions for the DDPG choice, ensuring safer trajectories: the horizontal field of view considered in the if statement has been reduced to 80° instead of 86° and, instead of considering just the range measurement of the angular sector that contains the target, it has been decided that also the range measurements of the two angular sectors closer to the one that includes the target have to fulfill the condition, i.e., the distances of obstacles in those sectors have to be higher than the distance from the target.

Algorithm 6.1 MATLAB validation control algorithm.

```
Input: starting pose of the robot, target position
repeat
    if the time step is a multiple of Ts/Ts_control then
        Simulate vision sensor
        Reduce the number of ranges
        Construct the observation vector
        Choose the policy responsible for making the action
        Compute the action exploiting the chosen policy
    end if
    Command robot action
    Update the pose of the robot
    Simulate the fictitious sensor to evaluate effective distance from obstacles
until a termination condition happens
```

6.1.2 Results

The control algorithm discussed above has been tested for both the policies related to PPO run 1B and run 3 by using 7 validation environments. These en-

vironments have been designed with the idea of experimenting several challenging situations, different with respect to the ones used for training (the exception is the first validation environment, that resembles the final training stage of the PPO agent). To better evaluate the performance of the algorithm, and to provide a uniformity in the test, all the environments involved are based on fixed configurations of obstacles. However, the verification of a huge number of different situations is allowed by the random generation of initial and target conditions. More in depth, the validation environment n°1, n°2, n°3, and n°4 involve the same random generation structure of the initial and target conditions adopted for the first, third, fourth and fifth training stage of the PPO agent. Instead, the validation environment n°5 is based on a fixed initial position, with the usual random initial orientation, and two lines of possible target positions, one in the upper part of the map and the other in the lower. On the other hand, the validation environments n°6 and n°7 are based on fixed starting positions and fixed target locations, while the initial orientation is randomly generated.

Both the policies showed very good performance, and, again, their difference cannot be sensed by comparing the trajectories. To make a significant comparison between the two, the algorithm has been evalueted in each environment for 500 times deploying the first policy and for other 500 times with the second. It has been measured the number of times that the robot reaches the goal (success), collides with an obstacles (crash) or the timeout happens. Moreover, at each test, it has been measured the minimum distance between the robot and the obstacles. This value, averaged over the total number of tests for each environment, can provide additional information about the safety of the collision-free trajectories performed by the robot.

Fig. 6.1, Fig. 6.2, Fig. 6.3, and Fig. 6.4 display an example of successful trajectory for each of the 7 environments, and in Table 6.1 and Table 6.2 the performance metrics are presented for policy 1B and policy 3. The control algorithm shows great collision-avoidance skills and huge generalization abilities for both the policies, that present very close results. However, the policy obtained from the run 3 is characterized by slightly higher results, and for this reason it has been chosen to be deployed in the target hardware, thorugh a Gazebo simulation. For the same reason, the trajectories presented in this section are related to the PPO policy obtained from run 3.

The hybrid architecture provides enormous accuracy in target reaching; however, it should be mentioned that the minimum distance from an obstacle is often detected when the action si computed by the DDPG policy. Nevertheless, thanks to the more restrictive conditions adopted for the deterministic policy transition, the drawback of the hybrid configuration is limited to the reduction of the safe margin, but has not resulted in an increase of failures. Actually, in this phase the observed crashes

Table 6.1: Performance of the control algorithm with PPO policy 1B.

Environment	Success rate	Crash rate	Timeout rate	Avg min distance
1	99.8%	0.2%	0%	0.070 m
2	84.2%	15.8%	0%	0.084 m
3	97.8%	2.2%	0%	0.118 m
4	98%	0.6%	1.4%	0.103 m
5	95.6%	4.4%	0%	0.065 m
6	100%	0%	0%	0.083 m
7	95.8%	0%	4.2%	0.084 m

Table 6.2: Performance of the control algorithm with PPO policy 3.

Environment	Success rate	Crash rate	Timeout rate	Avg min distance
1	100%	0%	0%	0.087 m
2	90.2%	9.8%	0%	0.100 m
3	96.6%	3.4%	0%	0.120 m
4	98.8%	1%	0.2%	0.086 m
5	97.8%	2%	0.2%	0.074 m
6	100%	0%	0%	0.091 m
7	100%	0%	0%	0.067 m

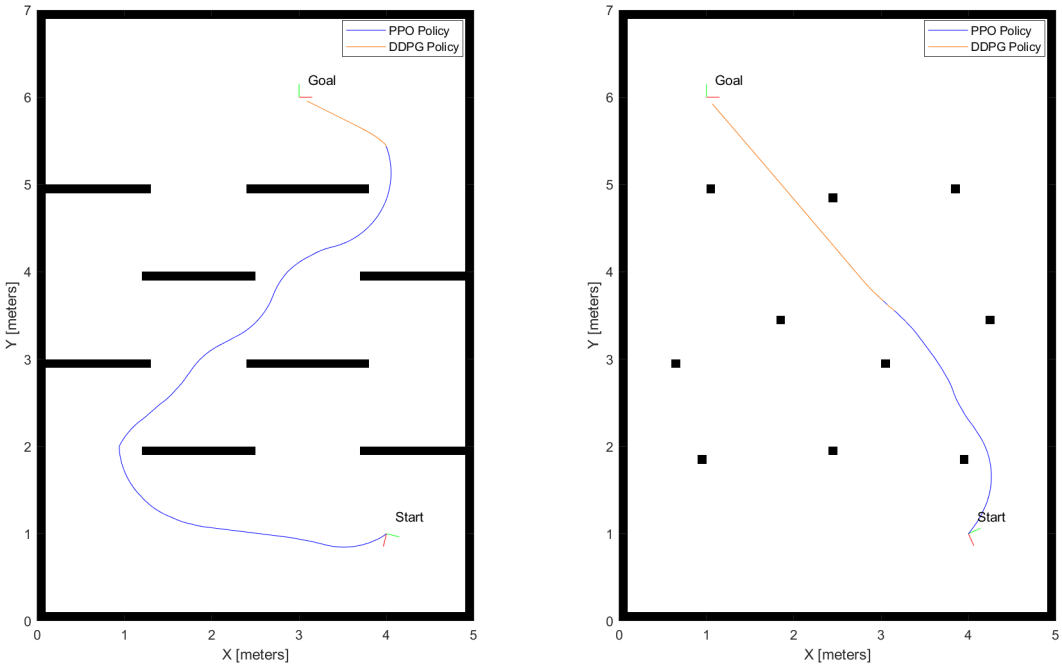


Figure 6.1: Robot trajectories in validation environments n°1 and n°2.

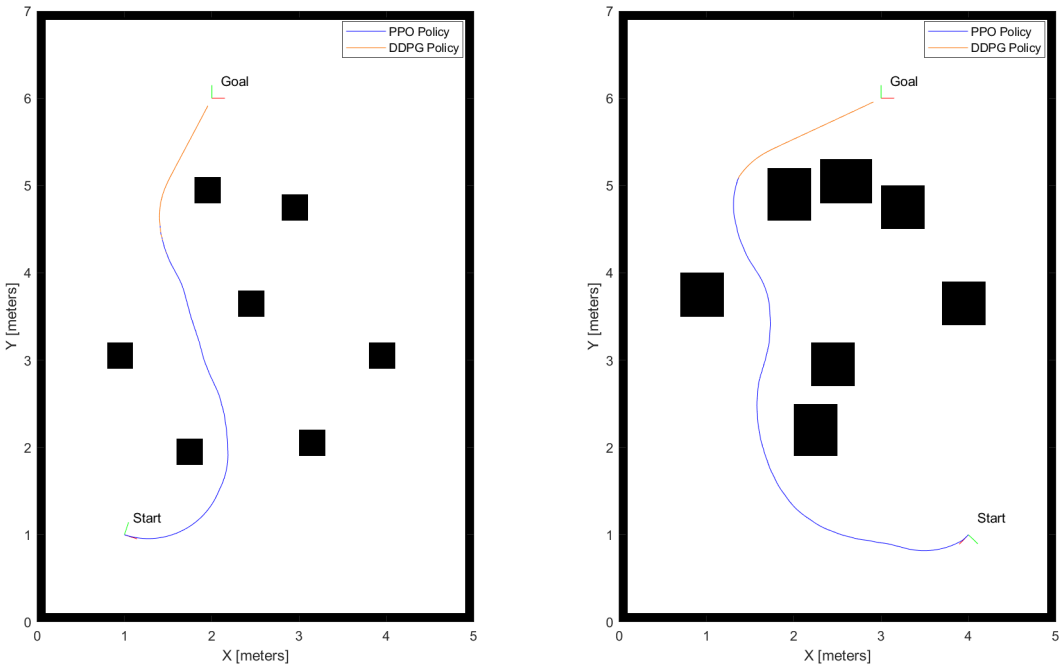


Figure 6.2: Robot trajectories in validation environments n°3 and n°4.

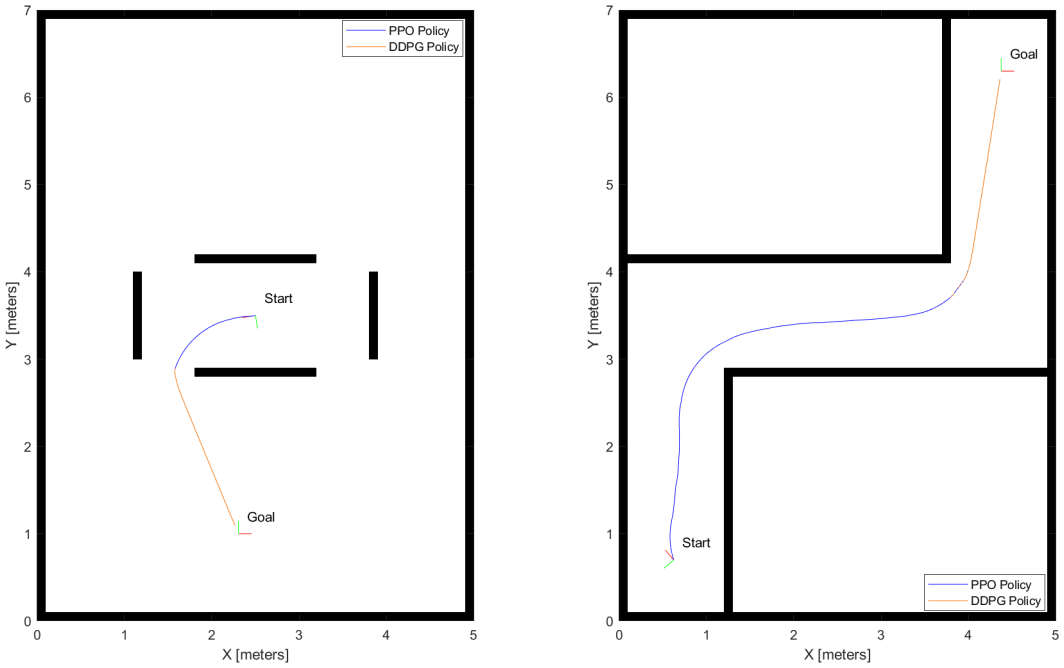


Figure 6.3: Robot trajectories in validation environments n°5 and n°6.

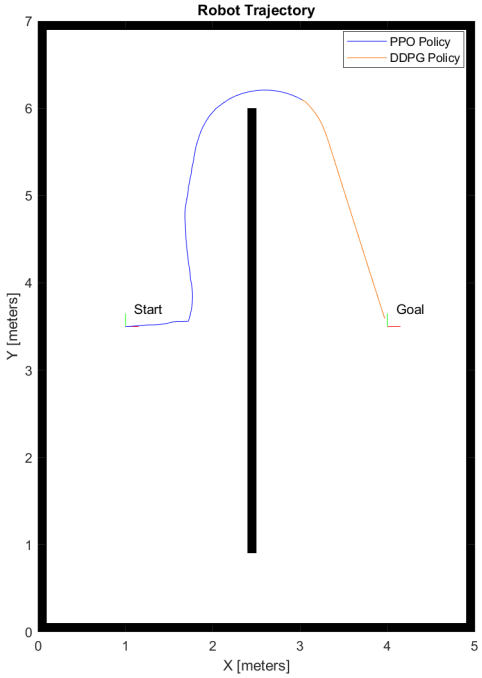


Figure 6.4: Robot trajectory in validation environment n°7.

have been occurred during the execution of the PPO policy. Regarding the timeout situation, it has been observed that the robot occasionally ends up in a deadlock: in practice, when the direction towards the goal is occluded by obstacles, the robot tries to rotate to find obstacle-free paths. The rotation occurs until the robot finds a way to move; however, when robot orientation becomes too different with respect to goal direction, and the path is already occluded, the robot starts to rotate towards the opposite direction. This behaviour makes sense, but considering the fact that the action selection made by the agent is based only on present observation vector, the consequence is that in certain situations the robot is blocked in an endless loop where it continues to alternate rotation to a side with rotation to the opposite side.

6.2 Gazebo Simulation

Finally, the objective of the work was to experiment the performance of the developed control algorithm when applied to the target robot. This has been done in simulation, using a Gazebo model of the robot.

6.2.1 Control Algorithm

Gazebo is a 3D robotics simulator, characterized by a robust physics engine and realistic rendering of environments. The simulation has been performed through a MATLAB-Gazebo cosimulation, that allows to connect MATLAB and Gazebo through the ROS interface. The model of the ground robot can be controlled by

publishing messages to the robot (such as desired velocities), and by subscribing to topics that the robot publishes (such as odometry and camera data).

The algorithm implemented in the Gazebo simulation is based on the same hybrid philosophy presented in previous section. In particular, the choice of the policy to adopt for computing the action is based on the same if statement explained in Subsection 6.1.1. Even the construction of the observation vector is the same, with the exception that the range measurements are obtained by converting the depth values provided by the camera through geometric relationships. This is the most heavy procedure of the algorithm from a computational point of view, since a huge number of x-y-z values are converted into distance and angle couples. To ensure a faster process, hence to acquire an higher control frequency, not all the points from the depth image are converted into range findings, but only the ones whose height is in the range $[-0.1\,m, 0.1\,m]$ with respect to the height coordinate of the center of the camera. The lower limit of the range is necessary to avoid to confuse the points of the floor with obstacles, while the upper limit allows to skip useless computations, since the height of the robot is only 14.35 cm. With the adoption of this range, the control algorithm shown a control frequency about 2 Hz; however, it varies a bit since it depends on the number of points detected by the camera in the specific scene.

The pseudocode of the control algorithm is presented in Algorithm 6.2. It has to be mentioned that the odometry data are computed with respect to the robot initial pose when connected in simulation, hence the target position has to be defined considering this reference.

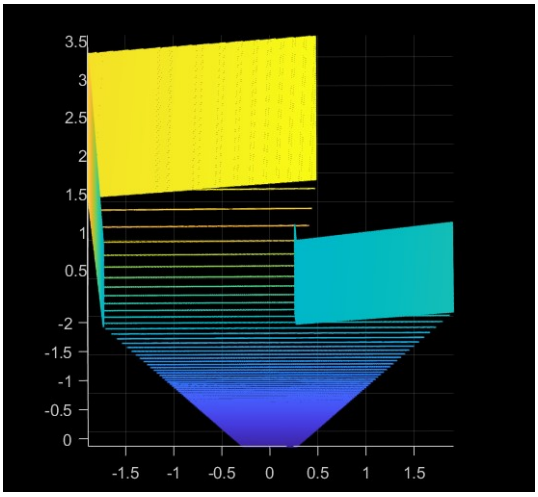


Figure 6.5: Example of depth image obtained from the camera.

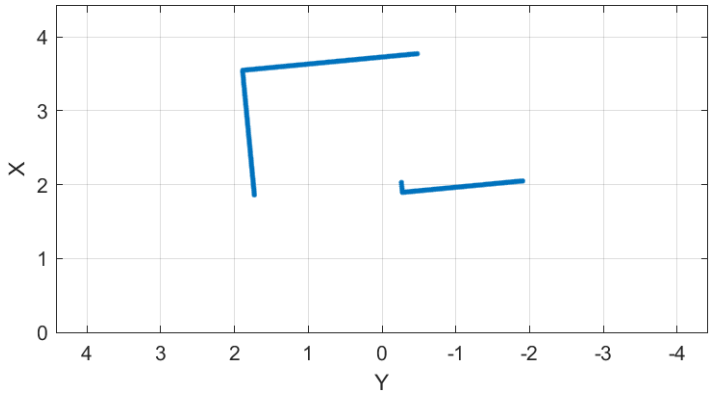


Figure 6.6: Example of converted range measurements.

Algorithm 6.2 Gazebo control algorithm.

```
Input: target position
repeat
    Receive odometry data
    Compute distance from target
    Compute angle error with respect to target
    Receive depth camera data
    Convert depth data into range measurements
    Reduce the number of ranges
    Construct the observation vector
    Choose the policy responsible for making the action
    Compute the action by exploiting the chosen policy
    Command robot action
until distance from target is lower than a threshold or a timeout happens
Command robot to stop
```

6.2.2 Results

The algorithm has been tested in several different and challenging scenarios, and some examples of its performance are presented in Fig. 6.7, Fig. 6.8, Fig. 6.9, and Fig. 6.10. The colors of the trajectories in the figures replicate the choices adopted for the validation results, hence the blue refers to the actions decided by the PPO policy, while the orange is related to the DDPG policy. It has to be noted that while the first Gazebo environment resembles the last PPO training stage, and the second is characterized by the same idea of some validation environments, the third and fourth scenario are very different to the ones that the agent tackled during learning. In particular, the fourth involves quite narrow paths, resembling a maze-like structure. The algorithm shows great performance and generalization abilities, altough, obviously, it presents some limitations. The most challenging situations are related to sharp turns and too occluded scenarios. A huge hardware limitation is represented by the minimum range detected by the depth camera and to its limited

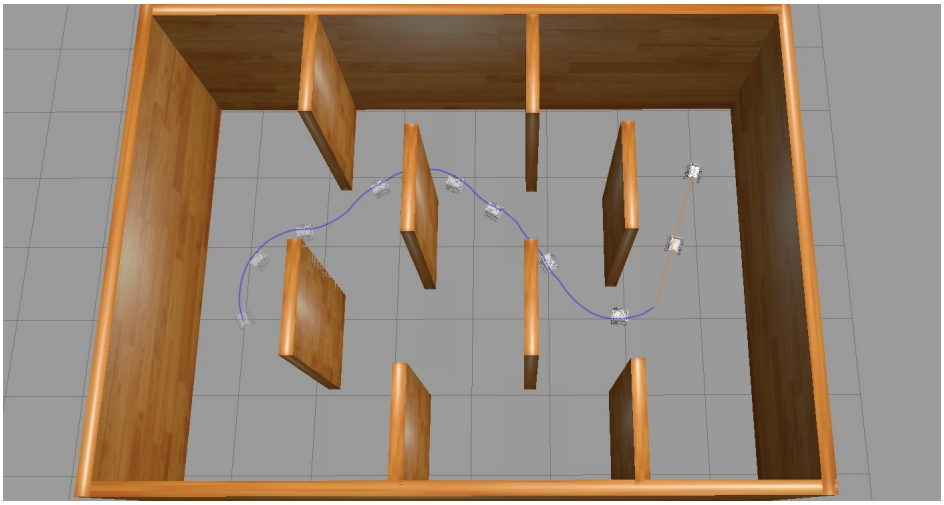


Figure 6.7: First Gazebo environment.

horizontal field of view (86°). Regarding the former, in the Gazebo simulation, when the object is closer to 30 cm to the center of the robot, i.e., about 20 cm with respect to the face of the robot, it cannot sense the object anymore. The hybrid architecture increases partially the efficiency of the trajectory and leads to great results in the target reaching accuracy, since the goal threshold has been maintaned to 10 cm as in MATLAB simulations, and it never happened that the robot overcame the target due to errors in the direction. However, due to aforementioned camera limitations, the hybrid configuration adopoted can sometimes lead to additional crashes, when it happens that the PPO policy brings the robot too close to an object until the presence of the obstacle is not correctly sensed anymore, and hence, the action decision is passed to the DDPG policy that cuts the path by going directly towards the obstacle.

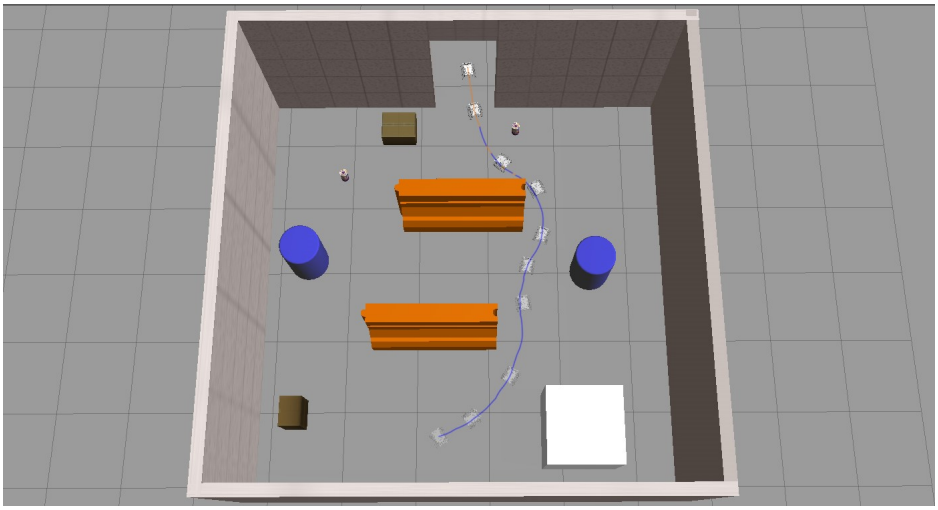


Figure 6.8: Second Gazebo environment.

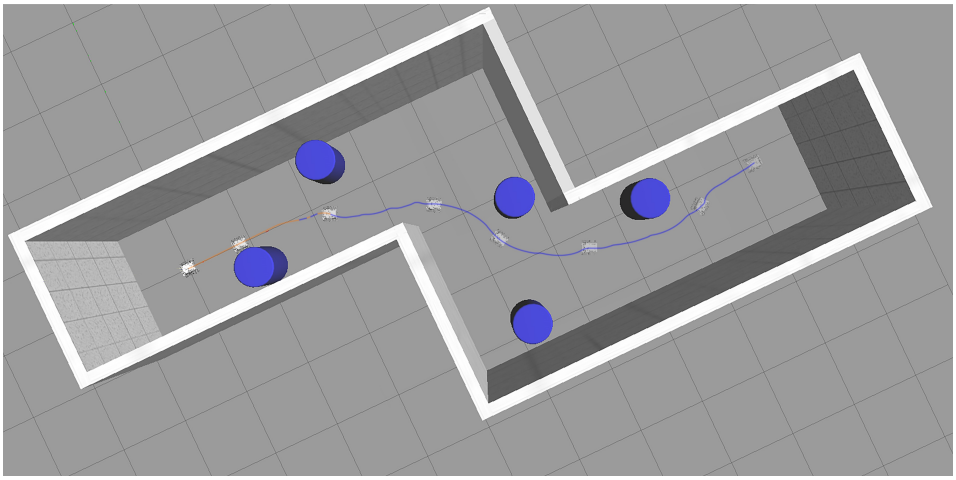


Figure 6.9: Third Gazebo environment.

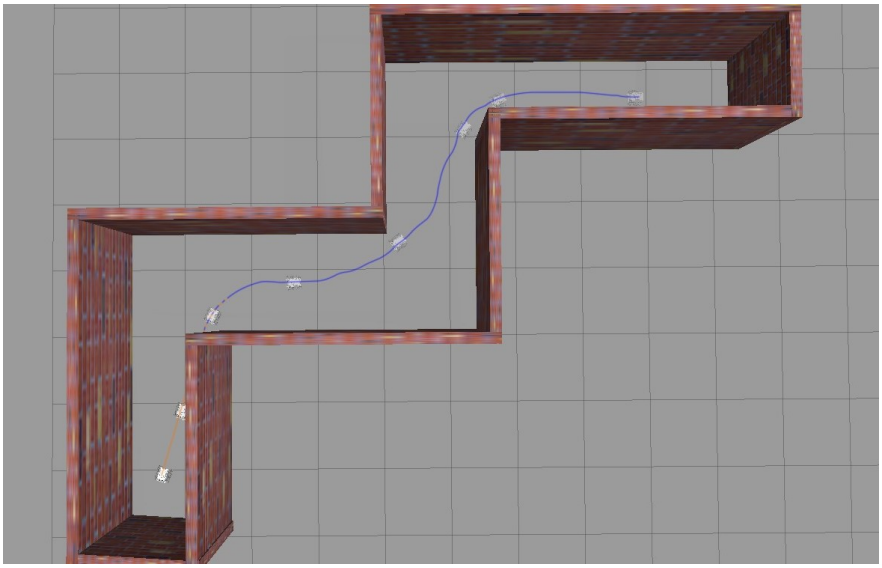


Figure 6.10: Fourth Gazebo environment.

Chapter 7

Conclusions and Future Works

In conclusion, the work of this thesis is based on three parts: the design and training of a reinforcement learning agent through Proximal Policy Optimization algorithm, to command the movements of a ground robot to reach a user-defined target while avoiding collisions with the obstacles present in the path; the design and training of a second agent, with the aim to produce efficient trajectories in specific situations, through Deep Deterministic Policy Gradient algorithm; the development and testing of a control algorithm, that deploys alternatively the two agents.

The stochastic PPO agent has been developed to learn collision avoidance abilities. It has been designed to receive some information about the distance of the obstacles in the area detected by the camera embedded with the robot, as well as the relative target position, as input, and to produce the linear and angular velocity of the robot as output. To ensure reproducibility, the training of the agent, based on five different environments characterized by incremental complexity, has been performed by exploiting multiple training runs with different random seeds, that led to slightly different but satisfactory policies. The most promising policies have been chosen to be deployed and tested into the control algorithm.

The resulting PPO policies showed great collision avoidance abilities; however, they present largely inefficient trajectories in trivial situations. Hence, a deterministic DDPG agent has been devised to provide efficient and precise trajectories when the path towards the target is obstacle-free.

Finally, a control algorithm has been developed to collect robot sensor data, process it, decide what agent to exploit, provide the inputs to it, and command robot velocities according to the decision made by one of the agents. The algorithm has been tested through MATLAB and Gazebo simulations by using a variety of challenging environments, and showed good performances in avoiding static obstacles and a great generalization ability. Nevertheless, its performances decreased when dealing with sharp turns or too occluded scenarios. The adoption of the hybrid architecture has led to more efficient trajectories and enormous accuracy in the

reaching of the target, but with the drawback of a reduction of the safety distance kept by the robot from obstacles.

Some ideas for future works could be:

- Implementation of real-word experiments, considering possible problems that could arise, e.g., noise in the depth camera data that can lead to ghost obstacles.
- Enhancement of the hybrid architecture, by trying to exploit the deterministic policy in a more efficient manner, without reducing the safety distance kept by the stochastic agent.
- Achievment of a continuous and smooth behaviour of the stochastic policy.
- Deployment of an additional vision sensor to increase the perception abilities of the agent.
- Implementation of a more complex observation vector for the collision avoidance agent, that includes both present and past information about the environment, e.g., range measurments in the last n time steps, to try to tackle moving obstacles situations.

Bibliography

[1] Alfons Schuster. *Intelligent Computing Everywhere*. Springer, 2007.

[2] National Geographic. *Intelligenza artificiale - La strada verso la superintelligenza*. 2018.

[3] Wikipedia. Applications of artificial intelligence, Last checked: 08-03-2021. https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence.

[4] Dario Izzo, Marcus Märtens, and Binfeng Pan. A survey on artificial intelligence trends in spacecraft guidance dynamics and control, 2018.

[5] NASA. Nasa takes a cue from Silicon Valley to hatch artificial intelligence technologies, 2019. https://www.nasa.gov/feature/goddard/2019/nasa-takes-a-cue-from-silicon-valley-to-hatch-artificial-intelligence-technologies

[6] ESA. Robots in space, 2020. https://www.esa.int/Enabling_Support/Preparing_for_the_Future/Discovery_and_Preparation/Robots_in_space2.

[7] Neil Abcouwer, Shreyansh Daftry, Siddarth Venkatraman, Tyler del Sesto, Olivier Toupet, Ravi Lanka, Jialin Song, Yisong Yue, and Masahiro Ono. Machine learning based path planning for improved rover navigation (preprint version), 2020. https://authors.library.caltech.edu/107566/1/2011.06022.pdf.

[8] M. Maimone, J. Biesiadecki, E. Tunstel, Y. Cheng, and C. Leger. Surface navigation and mobility intelligence on the Mars Exploration Rovers. In *Intelligence for Space Robotics*, pages 45 – 69, 2006. https://www-robotics.jpl.nasa.gov/publications/Mark_Maimone/05_Chapter3_final.pdf.

[9] Jeffrey Biesiadecki, Chris Leger, and Mark Maimone. Tradeoffs between directed and autonomous driving on the Mars Exploration Rovers. *I. J. Robotic Res.*, 26:91–104, 2007.

[10] Olivier Toupet, Hiro Ono, Tyler del Sesto, Nat Guy, Josh vander Hook, and Mike McHenry. Enhanced Autonav for Mars 2020 rover: Introduction, 2016. https://trs.jpl.nasa.gov/bitstream/handle/2014/48231/CL%2317-3124.pdf?sequence=1&isAllowed=y.

[11] NASA. Nasa's mars 2020 rover completes its first drive, 2019. https://www.nasa.gov/feature/jpl/nasas-mars-2020-rover-completes-its-first-drive.

[12] NASA JPL. Next-Gen Autonav concept, 2017. https://www-robotics.jpl.nasa.gov/tasks/showTask.cfm?FuseAction=ShowTask&TaskID=287&tdaID=700091.

[13] Microsoft Research. What drives Curiosity? Robotics technologies on the Mars science rover, 2014. https://www.microsoft.com/en-us/research/video/what-drives-curiosity-robotics-technologies-on-the-mars-science-rover/?from=http%3A%2F%2Fresearch.microsoft.com%2Fapps%2Fvideo%2Fdl.aspx%3Fid%3D230072.

[14] University of Texas at Austin and Texas Advanced Computing Center. Deep learning will help future Mars rovers go farther, faster, and do more science, 2020. https://www.sciencedaily.com/releases/2020/08/200819120700.htm.

[15] Junli Gao, Weijie Ye, Jing Guo, and Zhongjuan Li. Deep reinforcement learning for indoor mobile robot path planning. *Sensors*, 20(19), 2020.

[16] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification*. 1994.

[17] Giuseppe Bonaccorso. *Machine Learning Algorithms*. Packt, 2017.

[18] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2018.

[20] University of Alberta and Alberta Machine Intelligence Institute. Fundamentals of reinforcement learning. https://www.coursera.org/learn/fundamentals-of-reinforcement-learning.

[21] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and JoellePineau. An introduction to deep reinforcement learning. *Foundations and Trends in Machine Learning: Vol. 11, No. 3-4, 2018*, 2018.

[22] Hao Dong, Zihan Ding, and Shanghang Zhang. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. 01 2020.

[23] Stanford University. Cs231n: Convolutional neural networks for visual recognition, 2020. `https://cs231n.github.io/`.

[24] Wikipedia. Artificial neural network, Last checked: 08-03-2021. `https://en.wikipedia.org/wiki/Artificial_neural_network`.

[25] Wikipedia. Softmax function, Last checked: 04-03-2021. `https://en.wikipedia.org/wiki/Softmax_function`.

[26] Fjodor Van Neen. The neural network zoo, 2016. `https://www.asimovinstitute.org/neural-network-zoo/`.

[27] Kur' Hornik, Maxwell Stinchcombe, and Halber White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2:359–366, 1989.

[28] K. J. Åström. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965.

[29] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[30] Tingxiang Fan, Pinxin Long, Wenxi Liu, and Jia Pan. Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios. *The International Journal of Robotics Research*, 39:027836492091653, 05 2020.

[31] Jing Liang, Utsav Patel, Adarsh Jagan Sathyamoorthy, and Dinesh Manocha. Realtime collision avoidance for mobile robots in dense crowds using implicit multi-sensor fusion and deep reinforcement learning, 2020.

[32] Jun Jin, Nhat M. Nguyen, Nazmus Sakib, Daniel Graves, Hengshuai Yao, and Martin Jagersand. Mapless navigation among dynamics with social-safety-awareness: a reinforcement learning approach from 2d laser scans. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020.

[33] Michael L. Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119–125, 2009. Special Issue: Dynamic Decision Making.

[34] Roland Siegwart and Iliah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. 2004.

[35] Mathworks. Reinforcement learning with MATLAB. `https://it.mathworks.com/content/dam/mathworks/ebook/gated/reinforcement-learning-ebook-all-chapters.pdf`.

[36] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. 2010.

[37] Cheol-Joong Kim and Dongkyoung Chwa. Obstacle avoidance method for wheeled mobile robots using interval type-2 fuzzy neural network. *IEEE Transactions on Fuzzy Systems*, 24(2), 2016.

[38] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization, 2020.

[39] OpenAI. Proximal Policy Optimization, 2017. `https://openai.com/blog/openai-baselines-ppo/`.

[40] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin Riedmiller, and David Silver. Emergence of locomotion behaviours in rich environments, 2017.

[41] Brian Gaudet, Richard Linares, and Roberto Furfaro. Deep reinforcement learning for sixdegree-of-freedom planetary powereddescent and landing. 2018.

[42] Charles E. Oestreich, Richard Linares, and Ravi Gondhalekar. Autonomous six-degree-of-freedom spacecraft docking maneuvers via reinforcement learning, 2020.

[43] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[44] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[45] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[46] Open AI. Spinning Up - Proximal Policy Optimization, Last checked: 17-03-2021. `https://spinningup.openai.com/en/latest/algorithms/ppo.html#pseudocode`.

[47] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[48] MathWorks. Proximal Policy Optimization Agents, Last checked: 17-03-2021. https://it.mathworks.com/help/reinforcement-learning/ug/ppo-agents.html#mw_b971c286-831f-4055-9a6c-6a21e290ed98.

[49] Y. Bengio, JÃ©rÃŽme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. volume 60, page 6, 01 2009.

[50] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[51] Junli Gao, Weijie Ye, Jing Guo, and Zhongjuan Li. Deep reinforcement learning for indoor mobile robot path planning. *Sensors*, 20(19), 2020.

[52] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[53] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[54] MathWorks. Deep deterministic policy gradient agents, Last checked: 22-03-2021. https://it.mathworks.com/help/reinforcement-learning/ug/ddpg-agents.html.

[55] Olivier Sigaud. Reinforcement learning: 7. deep q network, 2018. http://pages.isir.upmc.fr/~sigaud/teach/dqn.pdf.

[56] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930.

[57] Wikipedia. Ornstein-uhlenbeck process, Last checked: 22-03-2021. https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process.

[58] Intel. Intel realsense product family d400 series - datasheet, 2020. https://www.intelrealsense.com/wp-content/uploads/2020/06/Intel-RealSense-D400-Series-Datasheet-June-2020.pdf.

[59] Nicolai A. Lynnerup, Laura Nolling, Rasmus Hasle, and John Hallam. A survey on reproducibility by evaluating deep reinforcement learning algorithms on real-world robots, 2019.

[60] MathWorks. rlDDPGAgentOptions, Last checked: 27-03-2021. https://it.mathworks.com/help/reinforcement-learning/ref/rlddpgagentoptions.html#mw_2875b71d-bfb0-4be4-b0d3-a44592c3cb30_head.