UNIVERSITÀ
DI SIENA
1240

Department of
Information Engineering and Mathematics

M. Sc. Program
ENGINEERING MANAGEMENT

A Reinforcement Learning Approach to Path Planning for Mobile Robots

Supervisor:
Prof. Antonio Giannitrapani

Adjunct Supervisors:
Prof. Andrea Garulli
Dr. Renato Quartullo

Candidate:
Tommaso Van Der Meer
Mat. 111833

Academic Year 2021-2022

Contents

Introduction

Autonomous navigation of mobile robots in an indoor environment is considered a basic capability of intelligent machines. However, such a problem is a complex task that requires a wide range of skills. Notably, the robot is required to locate itself continuously, to have an internal interpretation of its surroundings, and to be aware of the position of its goal at all times.

This thesis considers a sub-problem of autonomous navigation: *path planning*, also known as motion planning. In recent years, advances in artificial intelligence have shown *reinforcement learning* (RL) to be a good option for various similar problems. The objective of this thesis is to train a mobile robot for a short-range path planning using a well-known reinforcement learning algorithm: Proximal Policy Optimization (PPO). The considered robot is assumed to have a differential drive and is equipped with a laser to detect obstacles inside the indoor environment. The training algorithm is tailored to the specific problem to generate a control law that successfully steers the robot from its initial position to the target, while avoiding collisions with any obstacle.

The experiments are conducted using a 3D simulation environment relying on ROS2 and Gazebo. Using a simulation setup provides many advantages, such as the possibility to reduce significantly the training time, which is one of the main drawbacks of RL.

Much of the work is devoted to creating the infrastructure used to train the robot. For this reason, the implementation of the RL algorithm, as well as details on the software component, will be extensively discussed. A brief description of the content of each chapter follows.

- **Chapter 1** provides an overview of path planning problems and some common solutions adopted in the literature.
- **Chapter 2** illustrates the key points of the Proximal Policy Optimization algorithm.
- **Chapter 3** presents the problem setting, how the 3D environment is generated, and how to set up the infrastructure for training.
- **Chapter 4** focuses on the training of the mobile robot.
- **Chapter 5** shows and discusses the results obtained after the training process.
- **Chapter 6** summarizes the findings of the thesis and outlines some directions for future research.

Chapter 1

Path Planning for Mobile Robots

This chapter presents the formulation of path planning problems and a review of path planning techniques for mobile robots. Section 1.1 provides a brief introduction to autonomous navigation, which is the broader framework in which path planning is embedded. In Section 1.2, the problem is formulated. Finally, in Section 1.3, a review of the current solutions for path planning is outlined.

1.1 Autonomous navigation

A mobile robot is a robotic system equipped with locomotive elements (e.g., wheels, propellers, and legs) that allow movement [2]. Mobile robots are distinguished as: *wheeled mobile robots* (WMRs), *legged mobile robots* (LMRs), *unmanned aerial vehicles* (UAVs), and *autonomous underwater vehicles* (AUVs) [3]. Nowadays, mobile robots are used for a wide range of applications and must operate in unstructured and dynamic environments. These applications require autonomous systems capable of selecting appropriate actions from their interaction with the surroundings.

The biggest challenge in achieving autonomy is the **navigation problem**. Autonomous navigation requires the robot to continuously collect information about the environment and perform actions to reach its target position. Figure 1.1 illustrates the main elements involved in autonomous navigation.

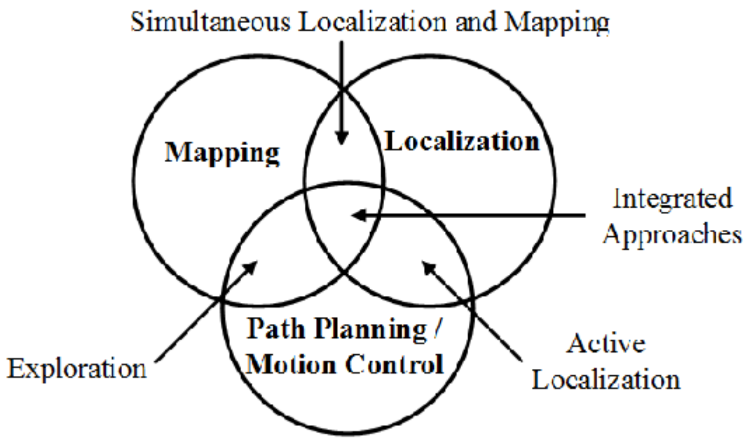

Figure 1.1: Elements of autonomous navigation[1].

As shown, path planning represents only one competence of robotic navigation. Path planning concerns the problem of finding a geometric path from the vehicle's current position to a target location such that it avoids obstacles. Mapping consists in creating a representation of the surrounding environment and localization is the ability to provide the exact position of the mobile robot relative to its internal representation of the environment.

In this thesis, the focus is on a path planning problem for a wheeled mobile robot. The specific problem setting considered in this work is introduced in Section 3.1.

[1]Image by Darmanin, R.N. - Bugeja, M., Jul 29, 2016. *Autonomous Exploration and Mapping using a Mobile Robot Running ROS*, p. 2

1.2 Problem formulation

In a classical path planning problem, a mobile robot is placed in a known environment, composed of free space and obstacles. Given the initial position and orientation of the robot (pose), and the position of the target, the goal is to find a continuous sequence of positions and orientations that allow the robot to reach the target without colliding with obstacles. These problems are often expressed by adopting the configuration space notation. A **configuration** uniquely determines the position and the orientation of the robot inside the working space. Below, a mathematical definition of configuration is given along with its notation [1].

Consider a rigid object $\mathcal{A}$ (the robot) moving in a Euclidean space $\mathcal{W} \in \mathbf{R^N}$ with $N = 2$ or 3. Let the obstacles $\mathcal{B}_1, ..., \mathcal{B}_q$ be closed subsets of $\mathcal{W}$ and let $\mathcal{F}_\mathcal{A}$ and $\mathcal{F}_\mathcal{W}$ be Cartesian systems attached to $\mathcal{A}$ and $\mathcal{W}$, respectively. Since $\mathcal{A}$ is a rigid object, every point $a \in \mathcal{A}$ is fixed with respect to $\mathcal{F}_\mathcal{A}$. Also, every point of the $\mathcal{B}_i$'s obstacles is fixed with respect to $\mathcal{F}_\mathcal{W}$, for $i = 1, ..., q$.

With this notation, a **configuration** $\mathbf{q} = (\mathcal{T}, \Theta)$ of the robot $\mathcal{A}$ is a specification of the position $\mathcal{T}$ and the orientation Θ of $\mathcal{F}_\mathcal{A}$ with respect to $\mathcal{F}_\mathcal{W}$. Since all the points $a \in \mathcal{A}$ are fixed with respect to $\mathcal{F}_\mathcal{A}$, their position in the $\mathcal{F}_\mathcal{W}$ Cartesian system can be derived from $\mathcal{T}$ and Θ.

The set of all the possible configurations of $\mathcal{A}$ is known as the **configuration space** $\mathcal{C}$ and the space occupied by $\mathcal{A}$ at a configuration $\mathbf{q}$ is denoted as $\mathcal{A}(\mathbf{q})$. In a similar way, the space occupied by a point $a \in \mathcal{A}$ is specified as $a(\mathbf{q})$.

The concept of continuity is fundamental to properly define a path in the configuration space. Intuitively, the distance of two configurations $\mathbf{q}$ and $\mathbf{q}'$ has to tend toward zero as the regions $\mathcal{A}(\mathbf{q})$ and $\mathcal{A}(\mathbf{q}')$ get closer to each other. The distance is often measured as follows:

$$d(\mathbf{q}, \mathbf{q}') = \underset{a \in \mathcal{A}}{max} \, ||a(\mathbf{q}) - a(\mathbf{q}')||. \tag{1.1}$$

Therefore, a **path** of $\mathcal{A}$ from $\mathbf{q}_{init}$ to $\mathbf{q}_{goal}$ (the initial and the goal configuration, respectively) is a continuous map $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ where $\tau(0) = \mathbf{q}_{init}$, $\tau(1) = \mathbf{q}_{goal}$, and $\mathcal{C}_{free}$ is the set of all configurations that avoid collisions with obstacles.

1.3 Planning approaches

Many algorithms have been developed to solve classical path planning problems. In this section, some of the most widely used approaches are presented to summarize the current state-of-the-art in motion planning. Clearly, the following review is not intended to be an exhaustive taxonomy of all available planning algorithms.

1.3.1 Artificial Potential Field (APF)

This approach treats the motion of the robot as an electrical charge. The movement is the result of a combination of the attraction exerted by the target and the repulsion coming from obstacles, as shown in Fig. 1.2. The trajectory resulting from the action of these forces is taken as the final path. The APF implementation does not require a high computational effort but often leads to a non-optimal path. In addition, there is the risk of getting trapped into a local minimum of the potential field and never reaching the target.

[2]Image by Anandan, B., 2017, Available at `https://github.com/banuprathap/PotentialFieldPathPlanning`

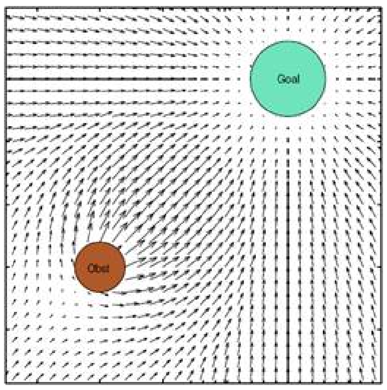

Figure 1.2: Artificial Potential Field forces[2].

1.3.2 Sampling-based algorithms

In a basic sampling-based algorithm, N configurations are sampled from the $\mathcal{C}_{free}$ set. Then, a roadmap (a graph) is built by connecting all the sampled configurations that respect the following condition: two points P and Q are connected in the roadmap only if the segment $\overline{PQ}$ is completely contained in $\mathcal{C}_{free}$. To test whether these segments lie in the free space a local planner is usually employed, like in the *Probabilistic RoadMap algorithm* (PRM) [5]. Once the roadmap is dense enough, the shortest path is computed using Dijkstra's algorithm (Fig. 1.3). Sampling-based approaches are often used in high-dimensional spaces because their computation time scales nicely with the dimensionality of $\mathcal{C}$.

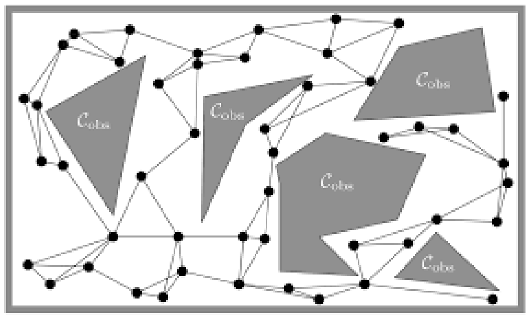

Figure 1.3: Roadmap generated from sampling-based search[3].

[3]Image by Kavraki, L. et al., 1996. *Probabilistic Roadmaps for Path Planning in High Dimensional Configuration Spaces*

1.3.3 Grid-based search

In this planning technique, the configuration space is discretized into a grid, where each square of the grid represents a single configuration. To assess if a configuration can be reached by an adjacent one, each connection is tested. In particular, if the connection is completely contained in $\mathcal{C}_{free}$, then the robot is allowed to move between the two grid squares. The path is then computed using classical search algorithms, such as A^* [7] or D^* [8]. A simple grid discretization is shown in Fig. 1.4 for a better understanding. However, since the dimension of the grid grows exponentially with the dimension of the configuration space, this approach is not suitable for high-dimensional problems.

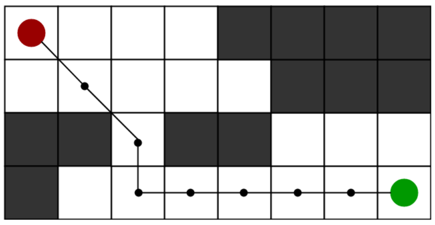

Figure 1.4: Grid discretization of the configuration space[4].

[4]Image by geeksforgeeks.org, 2023, Available at `https://www.geeksforgeeks.org/a-search-algorithm/`

Chapter 2

Reinforcement Learning and Proximal Policy Optimization

This chapter serves the purpose to introduce the reader to the main concepts, the terminology, and the notation of reinforcement learning along with a more detailed presentation of the PPO algorithm. These notions are useful to better understand the training phase carried out in this thesis. Section 2.1 introduces the main concepts of reinforcement learning. In Section 2.2, the reinforcement learning problem is formulated and other useful notions are presented. Finally, Section 2.3 provides an explanation of the PPO algorithm.

2.1 Main concepts of reinforcement learning

Reinforcement learning is a term that encompasses all machine learning methodologies that teach a given task to an agent by trial and error. The idea is to reward behaviors that you want the agent to repeat so that they are more likely to be reproduced in the future. The following review follows [9]. The interested reader is referred to the book [6] for a comprehensive description of RL techniques.

Reinforcement learning has become famous for the incredible results achieved by training bots to play games like Dota, Go, and many arcades of the Atari collection. The trained agents were able to beat experienced players and in some cases break historical records. These methods are being extensively applied in the robotics field to teach machines to accomplish complicated tasks.

The main components in RL are the **agent** and the **environment**. The agent collects partial (or complete) observations of the state of the environment and chooses the action to take from a predefined set. The state changes due to the effect of the action performed by the agent or, in some applications, even autonomously. The agent is then rewarded according to how good its behavior is deemed.

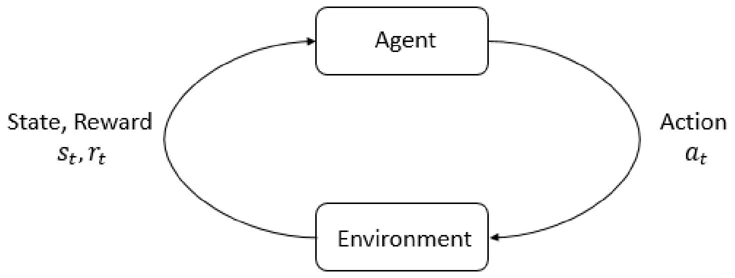

Figure 2.1: Agent-environment loop [9].

The goal is to find a policy (i.e., a rule that tells the agent which action to take for each state) that maximizes the cumulative reward. RL algorithms differ in the method by which the agent learns from its past experiences.

For a deeper understanding of the components involved in RL, additional information on the terminology that will be used is useful.

- A **state** s contains all the information to fully describe the state of the environment.

- An **observation** o contains partial or full information about the state of the environment. It omits all the data that is not useful for the agent to complete the task[1]. State and observation will be used interchangeably.
- The **action space** defines all the possible actions that the agent can perform in the given environment. The action space can be discrete (the set of actions is finite) or continuous (the actions are real-valued vectors), and its definition is critical as some RL algorithms cannot work with both types of space.
- A **policy** is a rule used by the agent to determine which action to take given the state of the environment. If the policy is deterministic, it is denoted as follows:

  $$a_t = \mu(s_t),$$

  where a_t is the action taken at time t if the state is s_t and policy μ is used. A stochastic policy instead is modeled as a probability distribution function (PDF) π, depending on s_t, from which the action a_t is sampled:

  $$a_t \sim \pi(\cdot|s_t).$$

  In the so-called *Deep RL*, a policy is essentially a neural network that takes as input the current state of the system and returns the action the agent has to take (or the mean and the variance of the action PDF π, if the policy is stochastic). The output of a neural network is determined by the weights (or parameters) assigned to each synapse. For this reason, policies in Deep RL

[1]When full information is available, the terms state and observation will be used interchangeably.

are called **parametrized policies** and are denoted as:

$$a_t = \mu_\theta(s_t),$$

$$a_t \sim \pi_\theta(\cdot|s_t).$$

where θ is the vector containing the parameters of the policy.

- An **episode** (or trajectory) τ is a sequence of actions a_t performed by the agent and states s_t of the environment over a given period of time.

  $$\tau = ((s_0, a_0), (s_1, a_1), (s_2, a_2), ...).$$

  In general, the end of an episode can be determined by three events:

  – the agent completes the task with success;

  – the agent does not complete the task within a predefined number of timesteps. In this situation, the episode is said to be truncated;

  – the agent fails to accomplish the task and reaches a terminal state.

  The state at the beginning of an episode is sampled from the **initial-state distribution** ρ_0.

  $$s_0 \sim \rho_0(\cdot).$$

  The environment changes its state according to its natural laws and also according to the last action performed by the agent. The state transitions can be deterministic or stochastic:

$$s_{t+1} = f(s_t, a_t),$$

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

- The **reward function** R is used to compute the instant reward of the agent at each timestep. It generally depends on the current state, the most recent action, and the next state:

  $$r = R(s_t, a_t, s_{t+1}).$$

  The reward function is crucial in reinforcement learning and needs to be carefully designed to make the training successful.
  The objective of the agent is to maximize the cumulative reward over an entire episode τ. This quantity is commonly known as *return* and is denoted as $R(\tau)$. Two types of return are used in RL algorithms:

  – **Finite-horizon undiscounted return**: the sum of all the rewards obtained by the agent over a finite trajectory.

  $$R(\tau) = \sum_{t=0}^{T} r_t.$$

  – **Infinite-horizon discounted return**: the sum of all the rewards obtained by the agent over an episode discounted by a factor $\gamma \in (0, 1)$.

  $$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

  The discount factor ensures the convergence of the sum and values recent rewards higher with respect to future rewards.

2.2 The reinforcement learning problem

As aforementioned, the goal in RL is to learn a policy that maximizes the **expected return**. The expectancy term is added to account for the stochastic behaviour of the environment.

Consider an environment with a stochastic state transition function and a stochastic policy. To formulate the optimization problem, the definition of the probability of a certain trajectory τ of T-steps is given:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t). \tag{2.1}$$

The expected return, which will be denoted as $J(\pi)$, is then equal to:

$$J(\pi) = \underset{\tau \sim \pi}{\mathbb{E}}[R(\tau)] = \int_\tau P(\tau|\pi)R(\tau). \tag{2.2}$$

Hence, the **RL optimization problem** can be formulated as follows:

$$\pi^* = \arg\max_\pi J(\pi). \tag{2.3}$$

In the case of a **parametrized policy**, the optimization problem is:

$$\theta^* = \arg\max_\theta J(\pi_\theta). \tag{2.4}$$

Notice that the structure of the policy (e.g., in the case of a parametrized policy, the number of neurons, synapses, and layers) is given. The problem is to find the parameters θ^* that maximize the expected return.

2.2.1 Value functions

The notion of **value function** is used in almost any RL algorithm. Value functions are often used to establish a baseline on the value of the current state of the environment using the current policy. For this reason, they represent the expected return if the agent starts in a certain state or a certain state-action pair and acts according to its policy for the rest of the time. The four most commonly used functions are:

- The **On-Policy Value Function**, denoted as $V^\pi(s)$, represents the expected return if the agent starts in the state s and acts according to policy π.

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.5}$$

- The **On-Policy Action-Value Function**, commonly known as **Q-value** and denoted as $Q^\pi(s, a)$, is the expected return if the agent starts in a state s, takes the action a (which could also be not compliant with the current policy), and then acts according to policy π for the rest of the time.

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.6}$$

- The **Optimal Value Function**, $V^*(s)$, stands for the expected return if the agent starts in state s and acts according to the optimal policy at each further step.

$$V^*(s) = \mathop{max}_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.7}$$

- The **Optimal Action-Value Function**, indicated as $Q^*(s, a)$, represents the expected return if the agent starts in state s, takes the action a, and then acts according to the optimal policy.

$$Q^{\pi}(s,a) = \underset{\pi}{max} \underset{\tau \sim \pi}{\mathbb{E}} [R(\tau)|s_0 = s, a_0 = a] \tag{2.8}$$

2.2.2 Advantage

Relative advantage is another notion commonly used in RL algorithms. It gives a quantitative measure of how much an action is better than all the other actions on average, when the agent is in a certain state. The relative advantage can be computed using the advantage function $A^{\pi}(s,a)$. More precisely, it represents the preference index of taking an action a in state s instead of sampling an action a' from the current policy π, assuming the agent acts according to π after that. Intuitively, the equation of the relative advantage is:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s), \tag{2.9}$$

where $Q^{\pi}(s,a)$ and $V^{\pi}(s)$ are respectively the on-policy action-value function and the on-policy value function defined in (2.6) and (2.5).

2.2.3 Policy optimization

The aim of this section is to introduce the reader to the mathematical foundations of policy optimization algorithms. These algorithms are a sub-class of reinforcement learning methods of which proximal policy optimization is also a part [10].

Consider a stochastic parametrized policy π_θ and a finite horizon undiscounted return $R(\tau)$. As previously mentioned, the problem is to find a set of parameters θ^*

that maximizes the expected return $J(\pi_\theta)$ as:

$$\theta^* = \arg\max_\theta J(\pi_\theta), \quad J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)]. \tag{2.10}$$

Note that the function to be maximized can be different for distinct algorithms. The parameters optimization can be achieved by gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k}. \tag{2.11}$$

The parameter α is called **learning rate** and controls the impact of the policy gradient step on the new set of parameters. Its value is critically important because if it is set too large, the step taken could be so big that it brings over the optimal region for the parameters. If, on the other hand, it is too small, the RL algorithm could need a large number of episodes to converge to the optimal region or, it could even get stuck in a non-optimal area.
The element $\nabla_\theta J(\pi_\theta)$ is called **policy gradient** and determines the direction along which to change the parameters in order to maximize the expected return. It can be proven that the policy gradient, in this case, is equal to:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \ \log \pi_\theta(a_t|s_t) R(\tau)\right]. \tag{2.12}$$

Since this is an expectation, this value can be estimated by collecting a set of episodes $D = \{\tau_i\}_{i=1,\dots,N}$ where the agent acts according to π_θ and finally compute:

$$\hat{g} = \frac{1}{|D|}\sum_{\tau\in D}\sum_{t=0}^{T} \nabla_\theta \ log \pi_\theta(a_t|s_t) R(\tau). \tag{2.13}$$

This a simpler expression of policy gradient (more recent RL algorithms use more advanced versions). Summarizing, to perform an update step on the current policy:

1. Collect a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$ using the current policy π_θ.
2. Compute the estimator $\hat{g}$ for the policy gradient using the equation (2.13).
3. Update the policy parameters according to (2.11).

Reinforcement learning algorithms relying on policy optimization are almost always run **on-policy**, meaning that at each step of the algorithm, the most recent policy is used to collect episodes.

2.3 Proximal Policy Optimization (PPO)

PPO has been designed to prevent the policy update step from generating a new policy too far from the previous one [11], [4]. This is advantageous because sometimes the parameters of the current policy are changed so much that the performance of the model drops dramatically and it may take many timesteps to return to the previous situation.

PPO is an on-policy algorithm that admits both discrete and continuous action spaces. Two variants of the optimization problem exist: **PPO-clip** and **PPO-penalty**. In this section, the focus is only on the first variant. The policy update step equation of PPO-clip is:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathbb{E}} \left[L(s, a, \theta_k, \theta) \right] \tag{2.14}$$

Notice that the expectation is over the state-action pair (a,s) where a is the action sampled from the previous policy. A single update is performed using **Stochastic Gradient Descent** (SGD), which is an alternative to classic policy gradient descent that highly reduces the computational effort by taking multiple smaller steps to update policy parameters. For each small step, the gradient is computed using a mini-batch (a smaller sample) instead of using all the sampled trajectories. Below, the L function is defined:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon\right) A^{\pi_{\theta_k}}(s, a)\right). \tag{2.15}$$

All the elements in (2.15) are explained below:

- π_θ is the new parametrized policy using θ as parameters. Hence, $\pi_\theta(a|s)$ outputs the sampled action from the new policy when the agent is in state s;
- π_{θ_k} is the old policy, i.e., the policy used to collect the trajectories at the k-th step of the algorithm;
- Considering the two previous points, $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ is essentially the **probability ratio** of taking the action a in state s with respect to the old policy;
- $A^{\pi_{\theta_k}}(s, a)$ represents the **advantage** (2.9) of taking action a in state s instead of sampling an action from the old policy π_{θ_k}, assuming that, after this, the agent always acts according to π_{θ_k};
- ϵ is a small parameter (usually $\epsilon \in [0, 0.3]$) defining how far the new policy is allowed to go with respect to the old policy;
- *clip*$(\cdot)$ is a function limiting the output value of the first argument passed within the given range. In this case, the limiting range is $[1 - \epsilon, 1 + \epsilon]$.

To better understand the idea behind this function, two cases are analyzed: positive advantage and negative advantage.

2.3.1 Positive advantage: $A^{\pi_{\theta_k}}(s, a) > 0$

If the advantage of a state-action pair (a, s) is positive with respect to the old policy π_{θ_k}, it is desirable to increase the probability of that action when in state s. So doing, the probability ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ will be greater then one.

The second term of the min function in (2.15) establishes a bound on the growth of the probability ratio of the state-action pair:

- If the ratio goes over $1 + \epsilon$, the clipped objective is selected as it is lower than the un-clipped one. Hence, the increase in the action likelihood will be bounded.
- If, instead, the new parameters θ generate a policy that lowers the probability ratio (which is not a desirable situation when the advantage is positive), even below $1 - \epsilon$, the un-clipped term is selected. This makes sense as such a situation will lead to a lower value of L, which goes in the opposite direction of the algorithm maximization goal (2.14). That is why, in this case, the descending probability ratio is not limited.

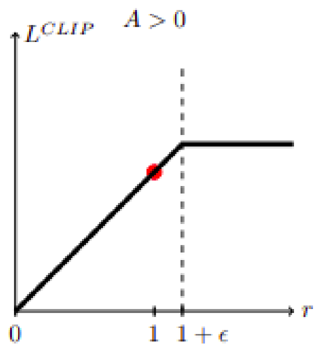

Figure 2.2: L-function with respect to the probability ratio $r = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ when the advantage A is positive.[2]

[2]Image by Schulman et al., 2017, p. 3 [4]

2.3.2 Negative advantage: $A^{\pi_{\theta_k}}(s, a) < 0$

When the relative advantage is negative, reducing the probability of the action a in state s is convenient.

Also in this case, the clipped objective limits the decrease in the likelihood of the state-action pair:

- If the probability ratio of the new policy is below $1 - \epsilon$, the second term of the *min* function will be selected resulting in a bounded decrease of the action likelihood.
- On the other hand, if the ratio is positive, the un-clipped term will be selected so that the penalization of such a policy will not be bounded. Remember that, since the advantage is negative, raising the likelihood of the action is undesirable.

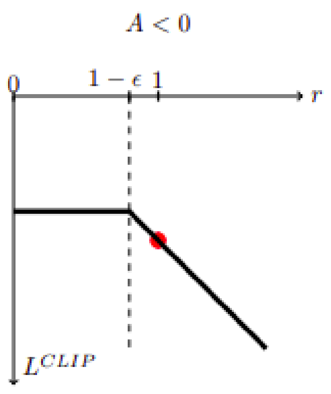

Figure 2.3: L-function with respect to the probability ratio $r = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ when the advantage A is negative.[3]

[3] Image by Schulman et al., 2017, p. 3 [4]

2.3.3 Exploration in PPO

To ensure that the agent explores new states of the environment, an **entropy** bonus is added to the loss function L. The impact of this element can be regulated by adjusting the entropy coefficient. Basically, the higher the coefficient, the greater the probability that the agent prefers a random action rather than sampling it from the current policy. The entropy bonus decreases as training goes ahead, ensuring that the agent behavior converges to its current policy over time.

2.3.4 PPO algorithm

A policy update requires the knowledge of the on-policy value function (2.5). Being an expectation, it can be computed using an appropriate estimator. In this regard, PPO employs a parameterized estimator whose parameters ϕ are updated at each step of the algorithm. Below, the pseudocode of the algorithm is presented [11].

Algorithm 1 PPO-Clip

```
1: Define initial policy parameters $\theta_0$ and initial value function parameters $\phi_0$.
2: for k=0,1,2,... do
3:     Collect a set of trajectories $D_k = \{\tau_i\}$ using the current policy $\pi(\theta_k)$.
4:     Compute the rewards $R(\tau_i)$ for each episode $i$.
5:     Compute advantage estimates $\widehat{A}^{\pi_{\theta_k}}$ with the current value function $V_{\phi_k}$ (using
       any advantage estimation method).
6:     Update policy parameters by maximizing the Eq. 2.14 using Stochastic
       Gradient Descent.
7:     Update value function parameters $\phi$ by regression on mean-squared error
       (using some gradient descent algorithm).
8: end for
```

Chapter 3

Software Architecture

This chapter provides the tools and methodologies used to set up the training infrastructure. Initially, in Section 3.1, the setting of the problem analyzed in this thesis is illustrated. In Section 3.2, the general infrastructure of the application developed is presented along with a brief description of all the software involved. Section 3.3 focuses on implementing control on the simulated robot and retrieving data from its sensors. Lastly, Section 3.4 illustrates the RL environment designed to train the desired agent.

3.1 Problem setting

This section presents the reference scenario for the motion planning problem under study. Following the approach proposed in [5], the goal is to train a reinforcement learning agent capable of reaching short-distance goals without knowing the large-scale topology of the space. Such an agent could be used as a local planner for existing path planning algorithms, like sampling-based approaches. In this way, the agent could be used not only to join pairs of close points, but also to control the motion of the robot once the planning is finished and the path is defined. A properly trained RL agent can avoid obstacles that were not present in the environment during the planning phase and satisfy all the task constraints involved in the actual motion control. In fact, the final model generates feasible control commands (e.g., linear and angular speed) that the mobile robot executes to follow the given sequence of waypoints[1].

Figure 3.1 presents the 3D environment used. The wheeled mobile robot is placed on the floor of a generic hospital furnished with various objects and even some people. The software and methods used to set up the environment will be discussed in the next sections.

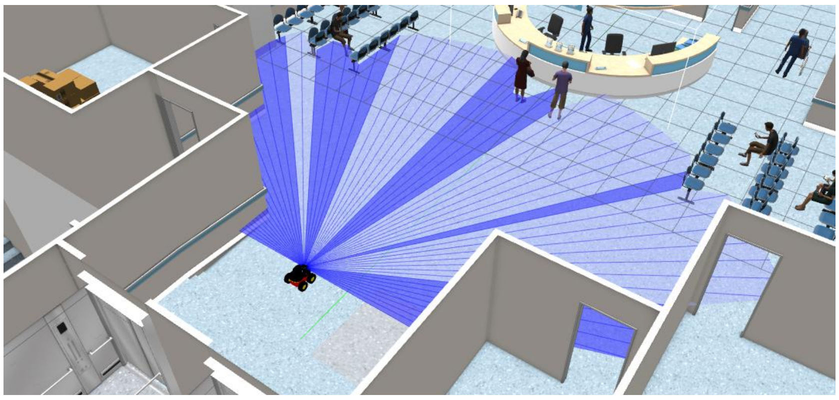

Figure 3.1: Overview of the 3D environment used in the study.

The robot employed is a Pioneer 3AT with a 4-wheel differential drive and a 180° laser for obstacle detection. The latter collects 61 distance measurements that can range from 0.08 to 10 meters. As mentioned above, the robot does not know the large-scale topology of the environment. However, through simulation, it is possible to capture the robot's position in space (with respect to the simulation Cartesian system $\mathcal{F}_{\mathcal{W}}$) at any instant, and moving actions can be instructed by specifying the desired angular and linear speed. In the next figure, the complete top view of the

[1]Intermediate points on a route used to outline the path

space can be observed.

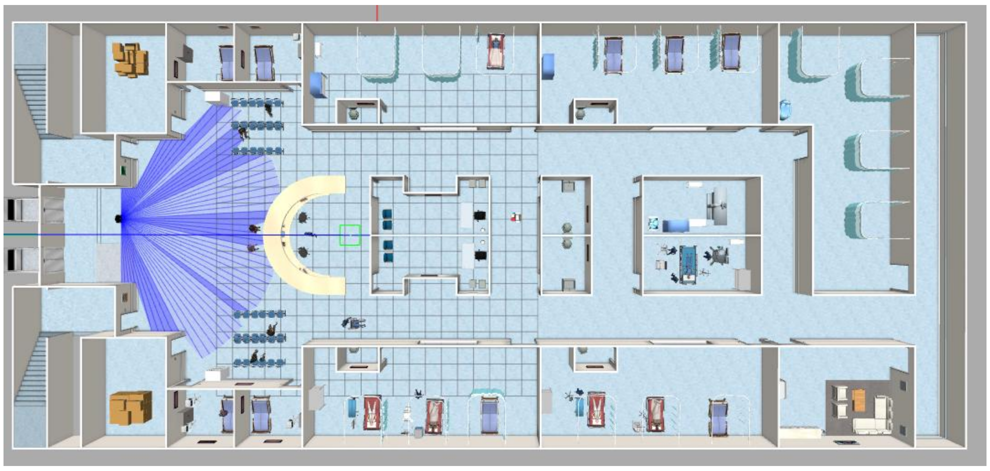

Figure 3.2: Top view of the 3D environment used in the study.

The trained RL agent is required to reach any short-range goal (up to 10 meters far away) in the free space without colliding with obstacles in a reasonable amount of time. Specifications chosen for this task are listed in the following:

- The angular speed of the robot is limited within the range [-1,1] rad/s while the linear speed is restricted within the range [0,1] m/s. Backward motion is not allowed.
- Since the laser is not positioned at the edge of the robot, a collision with an obstacle is detected whenever any laser sample has a value below 0.25 meters. This limit accounts for the geometry of the robot with respect to the laser position.
- The target is defined as a single point in the (x, y) space. The task is considered successfully completed whenever the distance from the center of the robot to the target is below 0.40 meters. This value considers the position of the laser relative to the center of the robot (it is positioned 0.14 meters ahead). If the

goal is near a wall, the robot must be able to reach the target before identifying a collision with the wall. Thus, the final value is given by the sum 0.25 m + 0.14 m + 0.1 m, where the last element represents a small additional margin.

3.2 General infrastructure

The entire Python code written to set up the simulation, train the robot, and evaluate the trained model is available on GitHub[2]. Figure 3.5 shows a diagram of the overall architecture.

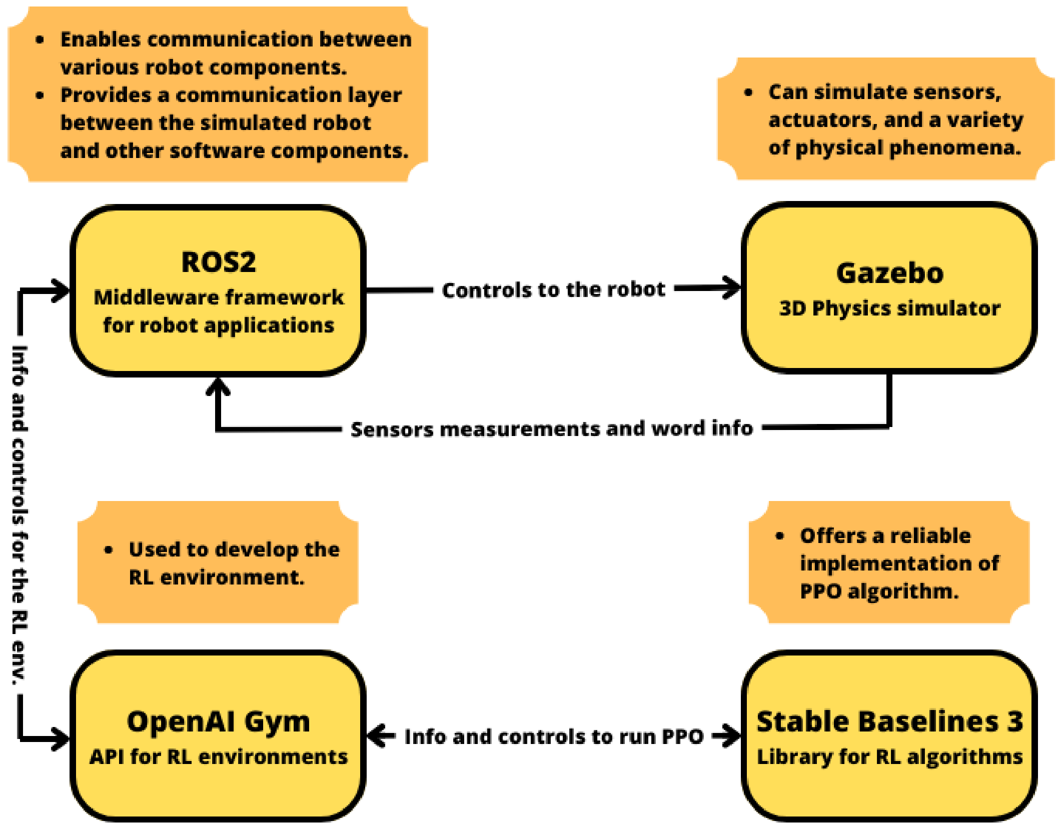
• Enables communication between various robot components.
• Provides a communication layer between the simulated robot and other software components.
• Can simulate sensors, actuators, and a variety of physical phenomena.
ROS2
Middleware framework for robot applications
Controls to the robot
Gazebo
3D Physics simulator
Sensors measurements and word info
Info and controls for the RL env.
• Used to develop the RL environment.
• Offers a reliable implementation of PPO algorithm.
OpenAI Gym
API for RL environments
Info and controls to run PPO
Stable Baselines 3
Library for RL algorithms

Figure 3.3: Scheme of the overall architecture.

[2]Van Der Meer, T. (Author) (2023), *Hospitalbot-Path-Planning*, `https://github.com/TommasoVandermeer/Hospitalbot-Path-Planning`.

The main framework used is **ROS2 Humble**, a set of tools and libraries to build robot applications [16]. Basically, ROS2 works with the concept of **nodes**. A node is a process that performs a specific task, such as reading sensor data, processing that data, or sending commands to actuators. Nodes can communicate with each other by subscribing or publishing to a specific topic, or by using services. Whenever a new message is received on a topic, all the nodes subscribed to that topic will execute a user-specified callback function to process that message. This structure enables communication between different robot components and also provides a communication layer between the robot and other software components.

Gazebo is employed to simulate the environment and the robot, it comes with a set of libraries that allow integration with ROS2 [17]. Gazebo typically interacts with ROS2 by creating a node that can publish and subscribe to topics outside the simulation environment. For example, the Gazebo node can publish data coming from a simulated LIDAR to a specific topic. A ROS2 node can then subscribe to that topic to execute a callback whenever new LIDAR data is published. Similarly, the Gazebo node can subscribe to a topic that receives velocity commands from a ROS2 node to adjust the speed of the simulated robot.

OpenAI Gym is an open-source toolkit for developing and comparing reinforcement learning algorithms [18]. It includes the base class *env* with all the functions needed for an RL environment (step, reset, render, close) and with tools to define action and observation spaces. The *env* class can be modified to develop a customized RL environment specific to the user's needs.

Stable Baselines 3 is a Python package compatible with OpenAI Gym that offers various implementations of the most popular RL algorithms, including Deep

Q-networks (DQN), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC) [14].

The 3D models of the hospital and the robot used in this thesis were taken from the Github repositories [12] and [13], respectively. Only the original robot model was slightly modified to add the 180° laser.

3.3 Robot controller

For the considered application, it is necessary to send velocity commands to the simulated robot, process the laser readings coming from the LIDAR, and get the position and the orientation of the robot to compute the distance from the goal. In addition, at the beginning of each training episode, the agent must be placed back in its initial position.

To implement all these functions, a dedicated ROS2 node has been developed. The node includes a publisher to issue commands to the robot in the simulation, a subscriber that executes a callback whenever new LIDAR readings are available, and another subscriber to get information about the current robot position and orientation. Moreover, the node includes a client linked to a Gazebo service that moves the robot back to its initial position at the beginning of each episode.

3.3.1 Action publisher

The standard packages integrating the functionalities between Gazebo and ROS2 come with an additional plugin (`libgazebo ros diff drive`) that can be included in the SDF file (Simulation Descriptor Format) of the robot to simulate a differential drive.

The plugin creates a ROS2 topic to which the Gazebo node is subscribed: when a new speed command is published on the topic, it is transmitted to the robot drive and is executed.

To correctly employ the plugin, the characteristics of the wheels have been specified in the SDF file along with the plugin tag and the name of the new topic (*/demo/cmd_vel*). Then, it suffices to add a publisher to the robot controller node that sends commands to the */demo/cmd_vel* topic.

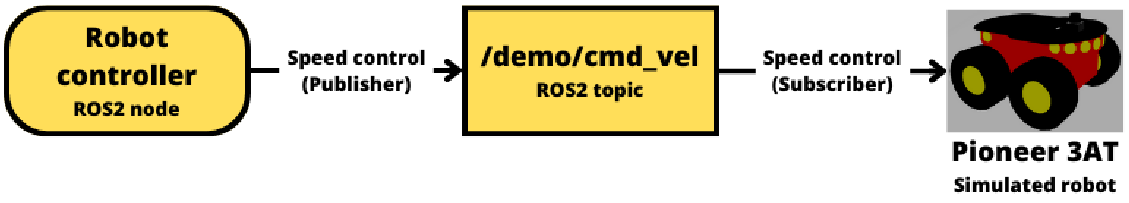

Figure 3.4: Scheme of the velocity command flow.

3.3.2 Laser and pose subscribers

Retrieving laser readings can be achieved by adding the plugin `libgazebo ros ray sensor` to the robot SDF file. As with the previous plugin, this one creates the topic */demo/laser/out* where the LIDAR readings are published at the specified frequency, in this case, 10 Hz. The robot controller node is subscribed to that topic to finally receive the laser information.

In the same fashion, odometry[3] data can be fetched from the simulated robot with the plugin `libgazebo ros diff drive` already introduced. This plugin also simulates an IMU sensor (Inertial Measurement Unit) to compute the odometry infor-

[3]The measurement of a robot's position change relative to a known location using motion sensors

mation of the robot. Only the robot base frame and the updating frequency (100 Hz) have been manually added to the code. The information is first published to the */demo/odom* topic and then retrieved from the robot controller node through a subscription.

Notice that, by default, the IMU sensor includes noise in its readings thus making the simulation more realistic. No further noise has been added to all other sensors and actuators.

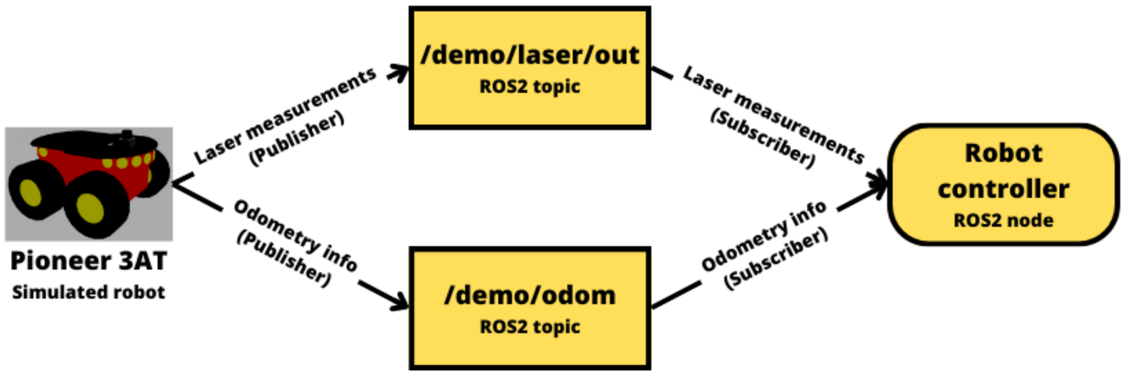
/demo/laser/out
ROS2 topic
Laser measurements (Publisher)
Laser measurements (Subscriber)
Robot controller
ROS2 node
Pioneer 3AT
Simulated robot
Odometry info (Publisher)
Odometry info (Subscriber)
/demo/odom
ROS2 topic

Figure 3.5: Scheme of the sensors readings flow.

3.3.3 Set entity state client

At the beginning of each episode, the robot must be moved to a new initial position according to the initial-state distribution $\rho_0(\cdot)$. The fastest way to do this, in a Gazebo simulation, is to use the Gazebo plugin `libgazebo ros state`. Once added to the world file, it enables to change the state of all the models inside the simulation by specifying the new pose and the initial velocity.

The plugin creates a ROS2 service (*/demo/set_entity_state*) that processes incoming client requests to set the new model state. At this point, it is sufficient to build a client inside the robot controller node that eventually sends a request to change the agent state whenever a new episode begins.

3.4 Gym environment

The reinforcement learning environment has been developed using OpenAI Gym. When creating a custom environment, the following elements must be specified:

- The **observation space**, which must contain enough information to allow the agent to select the best action.
- The **action space**, which represents the set of all the actions that the agent can perform.
- The **reward function**, which is necessary to compute the instant reward at the end of each step.
- The **step method**, where it is essential to specify the criterion by which an episode ends (if there is one).
- The **reset method**, which is used to initiate a new episode.

After defining all these elements, it is possible to use Stable Baselines 3 to train the RL agent.

3.4.1 Environment initialization

First of all, the customized environment created is a Python class that inherits from the robot controller node class. In this way, all methods created to send actions to the robot, receive sensors data, and set the new agent state, can be used inside the RL environment.

The observation space has been defined as a dictionary containing the polar coordinates $(r, \theta^T_{\mathcal{F}_\mathcal{A}})$ of the goal relative to the robot's Cartesian system $\mathcal{F}_\mathcal{A}$, and the laser samples o_i with $i = 1, ..., 61$ of the 180° LIDAR with up to 10 meters depth. Polar coordinates limits have been defined as $[0, 60]$ for the distance (60 meters is the maximum distance reachable on the hospital floor) and $[-\pi, \pi]$ for the angle (which is measured in radians).

The action space is a 2-dimensional array where the first element is the linear speed and the second element is the angular speed. The two actions are continuous and are limited as mentioned in Section 3.1.

A good practice when training an RL agent is to normalize both its observation and action spaces. This ensures better results during training and faster convergence to the optimal policy. For this reason, all the elements of the observation space have been normalized between 0 and 1 whereas the elements of the action space have been normalized between -1 and 1.

3.4.2 Step method

The step method includes most of the natural laws of the environment. It takes an action as input, computes the new state after applying the action, calculates the reward, and checks if the episode is finished or not. Below the step method developed in this thesis is reported:

```
    def step(self, action):
        # De-normalize the action to send the command to robot
        action = self.denormalize_action(action)
        # Apply the action
```

```
5          self.send_velocity_command(action)
6          # Spin the node until sensors samples are updated
7          self.spin()
8          # Compute the polar coordinates
9          self.transform_coordinates()
10         # Update the robot location and laser readings
11         observation = self._get_obs()
12         # Update infos
13         info = self._get_info()
14         # Compute Reward
15         reward = self.compute_rewards(info)
16         # Check if the episode is terminated
17         done = (info["distance"] < self._minimum_dist_from_target) \
18         or (any(info["laser"] < self._minimum_dist_from_obstacles))
19
20         return observation, reward, done, info
```

Before executing the step method, the Gym environment samples an action from the current policy which is structured according to the action space given. Since the action space is normalized, before sending the action to the simulated robot, it has to be de-normalized as in row 3 of the above code.

After applying the action it is necessary to check the new state of the system, i.e., the LIDAR samples and the polar coordinates of the robot must be retrieved (see Section 3.3.2). In order for a ROS2 node to execute callback functions when a new message is received on a subscribed topic, it must be spinning. Essentially, spinning a node means allowing it to keep running and check for events on subscribed topics and service calls. Nevertheless, the classical ROS2 *spin* function is a blocking operation in the code, which is not desired in this case. For this reason, a customized `spin()` function was developed: the node is spun until it catches the first event and executes the first callback with the *spin_once* method; if both the laser readings and the odometry information are not updated, the *spin_once* method is called again.

That said, it is possible to calculate the approximate maximum distance the robot can travel in a single step. Assume that the action taken includes the maximum linear speed (1 m/s) and no angular speed and assume that there is no delay due to code execution. Since the lower sensor update rate is 10 Hz (LIDAR update rate), each action is executed for 0.1 seconds. With a linear speed of 1 m/s, the maximum distance traveled by the robot is 0.1 meters.

The odometry information coming from the sensor gives only the pose of the robot relative to the Cartesian system of the simulation environment $\mathcal{F}_{\mathcal{W}}$. The position is given as a 3-tuple $(x_{\mathcal{F}_{\mathcal{W}}}, y_{\mathcal{F}_{\mathcal{W}}}, z_{\mathcal{F}_{\mathcal{W}}})$ and the orientation as a quaternion (q_0, q_1, q_2, q_3). Therefore, the given coordinates must be transformed to be consistent with the definition of the observation space (see Section 3.4.1).

The coordinates of the target $(x^T_{\mathcal{F}_{\mathcal{W}}}, y^T_{\mathcal{F}_{\mathcal{W}}})$ with respect to the $\mathcal{F}_{\mathcal{W}}$ system are given and the robot can only move in the (x, y) plane (and can only rotate with respect to the Z-axis), then, the radius r is:

$$r = \sqrt{(x^T_{\mathcal{F}_{\mathcal{W}}} - x_{\mathcal{F}_{\mathcal{W}}})^2 + (y^T_{\mathcal{F}_{\mathcal{W}}} - y_{\mathcal{F}_{\mathcal{W}}})^2}. \tag{3.1}$$

Then, to compute the angle $\theta^T_{\mathcal{F}_{\mathcal{A}}}$, the coordinates $(x^T_{\mathcal{F}_{\mathcal{A}}}, y^T_{\mathcal{F}_{\mathcal{A}}})$ of the target relative to the robot system $\mathcal{F}_{\mathcal{A}}$ and the rotation angle $\theta_{\mathcal{F}_{\mathcal{W}} \to \mathcal{F}_{\mathcal{A}}}$ between the Z-axes of the two systems are necessary. The quaternion for a given Z-axis rotation is:

$$q_0 = 0, \quad q_1 = 0, \quad q_2 = \sin(\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}}/2), \quad q_3 = \cos(\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}}/2). \tag{3.2}$$

Therefore, the rotation angle can be computed as:

$$\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}} = 2\ atan2(q_2, q_3). \tag{3.3}$$

Thus, the coordinates $(x^T_{\mathcal{F}_\mathcal{A}}, y^T_{\mathcal{F}_\mathcal{A}})$ can be calculated as:

$$\begin{aligned} x^T_{\mathcal{F}_\mathcal{A}} &= cos(-\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}})(x^T_{\mathcal{F}_\mathcal{W}} - x_{\mathcal{F}_\mathcal{W}}) - sin(-\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}})(y^T_{\mathcal{F}_\mathcal{W}} - y_{\mathcal{F}_\mathcal{W}}), \\ y^T_{\mathcal{F}_\mathcal{A}} &= sin(-\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}})(x^T_{\mathcal{F}_\mathcal{W}} - x_{\mathcal{F}_\mathcal{W}}) + cos(-\theta_{\mathcal{F}_\mathcal{W} \to \mathcal{F}_\mathcal{A}})(y^T_{\mathcal{F}_\mathcal{W}} - y_{\mathcal{F}_\mathcal{W}}). \end{aligned} \tag{3.4}$$

Finally, the angle $\theta^T_{\mathcal{F}_\mathcal{A}}$ of the goal relative to the robot Cartesian system $\mathcal{F}_\mathcal{A}$ can be computed as:

$$\theta^T_{\mathcal{F}_\mathcal{A}} = atan2(y^T_{\mathcal{F}_\mathcal{A}}, x^T_{\mathcal{F}_\mathcal{A}}). \tag{3.5}$$

At this point, the polar coordinates $(r, \theta^T_{\mathcal{F}_\mathcal{A}})$ are known. The method _get_obs() normalizes the new observations before they are returned to the RL algorithm. The _get_info () method, instead, saves some un-normalized information about the state of the system.

Subsequently, the reward is computed (more information about the reward function can be found in Section 3.4.3) based on the system information saved. Finally, a control is performed to check if the episode is terminated.

There are two criteria by which the episode can end:

- The robot has reached the target: $r < 0.4$ meters;
- A collision has been detected from the LIDAR: $\min_i o_i < 0.25$ meters.

3.4.3 Reward function

A sparse reward has been employed for this study. The reward r_t at time t is computed as follows:

$$r_t = \begin{cases} 1 & \text{if } r < 0.4 \quad \text{(goal reached)} \\ -1 & \text{if } \min_i o_i < 0.25 \quad \text{(obstacle collision)} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

In this way, policies that lead the robot to a collision are penalized whereas behaviors that guide the robot to the target are rewarded with the exact opposite value. Such a reward function should produce a risk-neutral policy.

3.4.4 Reset method

The reset method is called before starting a new episode. Below, the implementation of the reset method is reported:

```python
def reset(self, seed=None, options=None):
    # Get the new pose of the robot
    pose2d = self.randomize_robot_location()
    # Call the set robot position service
    self._done_set_rob_state = False
    self.call_set_robot_state_service(pose2d)
    while self._done_set_rob_state == False:
        rclpy.spin_once(self)
        # Randomize target location
        self.randomize_target_location()
        # Compute the initial observation
        self.spin()
        self.transform_coordinates()
        # Update state and additional info
        observation = self._get_obs()
        info = self._get_info()

        return observation
```

First, the initial pose of the robot in the new episode is computed by calling the randomize_robot_location() method. The new pose is then used to call the */set_entity_state* service which changes the position of the robot in the simulation.

Optionally, also the target position can be changed to avoid overfitting problems on the trained model. In the end, the initial observation is computed as in the *step* method.

Chapter 4

Training of the Agent

In this chapter, the training process of the agent is illustrated. The first section shows how the simulation speed has been increased in order to reduce the training time. Section 4.2 presents the method used to tune the hyperparameters for the PPO algorithm. Section 4.3 shows the training of the agent on a simplified task to check the reliability of the customized Gym environment and to verify the effectiveness of the chosen reward function. Finally, in Section 4.4, the training process of the final agent is presented.

4.1 Increasing simulation speed

Once a working Gym environment is available, a reinforcement learning model can be trained using very few lines of code through Stable Baselines 3. However, RL is very sample inefficient in general (i.e., a many time steps are required for the convergence to the optimal policy). For this reason, it is advisable to speed up the simulation time as much as possible. In this way, the training algorithm can process more time steps in the same amount of time. In Gazebo, the simulation time can be adjusted by changing two parameters in the world file:

- *max_step_size* (default value: 0.001 seconds), determining the time in the simulation to be simulated in one step;
- *real_time_update_rate* (default value: 1000 Hz), which determines the minimum time period after which the next simulation step is triggered.

The multiplication between these two parameters determines the *real_time_factor* (i.e., the simulation time divided by the real time). However, this parameter cannot be increased indefinitely. Clearly, a limit is imposed by the computational effort that the machine on which the algorithm runs can withstand.

In this work, the real time factor for training has been increased by raising *max_step_size* to 0.01 seconds and the *real_time_update_rate* to 5000 Hz, resulting in a training speed 50x faster. In addition, to reduce the computational effort on the machine, the graphics output was turned off during training using a dedicated launch file for Gazebo.

The machine used to run the training algorithm mounts an *Intel(R) Core(TM) i9-9900 CPU @ 3.10GHz*, it has 62 GBs of RAM, and has *Ubuntu 22.04 LTS* installed as its operating system. Notice that the training times mentioned later refer to the use of this machine.

4.2 Tuning the hyperparameters

Before training an RL agent, it is important to properly tune the hyperparameters, as the default ones are not necessarily suitable for every environment. To do so, the Python library `Optuna` was employed.

`Optuna` allows the user to run a desired number of relatively small training trials, each with different parameters, and compare the results at the end. Clearly, the more the trials conducted, the more likely it is to find the optimal hyperparameters.

Stable Baselines 3 enables the user to change many parameters [15], a good portion of which has been left at their default value. The ones that were optimized through Optuna are:

- **learning rate** (default: 0.0003): the gradient update step size α in (2.11);
- **n steps** (default: 2048): the number of steps to run per policy update;
- **gamma** (default: 0.99): discount factor γ to compute the infinite-horizon discounted return;
- **clip range** (default: 0.02): clipping parameter ϵ of the loss function in (2.15);
- **gae lambda** (default: 0.95): factor to regulate the bias-variance trade-off for Generalized Advantage Estimator;
- **ent coef** (default: 0.0): used to compute the entropy bonus;
- **vf coef** (default: 0.5): value function coefficient for the loss calculation.

As it will be shown next, good results are very sensitive to parameter optimization. That is why, for each different task, hyperparameter tuning is performed.

4.3 Training on a simplified environment

At the beginning of the training phase, to evaluate the effectiveness of the chosen reward function, a simplified environment in which the LIDAR samples are not taken into consideration was analyzed.

The purpose of such a simplified setup is to debug undesirable behaviors in the Gym environment and verify that the chosen reward function leads to the desired policy. Later, greater complexity can be added to the problem so that more sophisticated agents can be trained. The changes made for the simplified environment are listed below.

- The observation space contains only the polar coordinates of the target $(r, \theta^{T}_{\mathcal{F}_{\mathcal{A}}})$ relative to the robot.
- An episode ends if the robot reaches the target $(r < 0.4)$ or if it goes 3.5 meters farther than the target $(r > 3.5)$, creating a navigation limit.
- The reward function only returns 1 when the robot reaches the target, in all the other cases it returns 0.

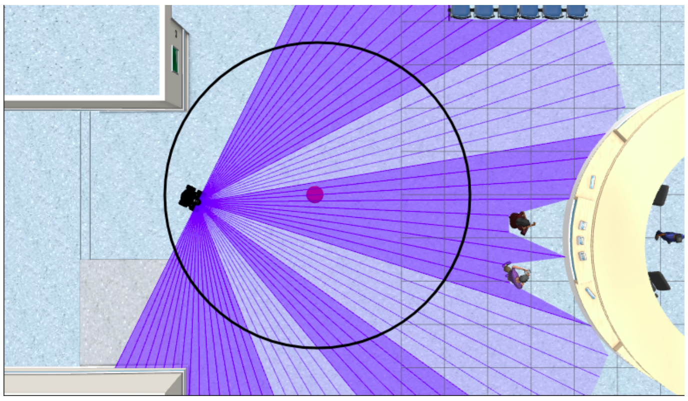

Figure 4.1: Overview of the training setting on the simple task (the black circle is the navigation limit and the red dot is the target). LIDAR measurements are displayed but not used for training.

The robot initial location and target location are fixed for each episode. Obviously, because LIDAR readings are not considered, the robot is placed in an area where there are no obstacles. Also, given the simple task to accomplish, the default hyperparameters of the PPO algorithm are used.

Before starting the training process, the maximum number of steps after which an episode is truncated must be set, in this case, a maximum number of 1000 timesteps was chosen. Given that the agent can travel a distance up to 0.1 meters in a single step and considering that, at the beginning of an episode, the agent is placed 3 meters away from the obstacle, the robot could complete the task in a minimum of 30 steps. Hence, 1000 steps per episode are more than enough to successfully complete the task.

In Figure 4.2 the mean episodic training reward (averaged over 100 episodes) is shown (the X-axis represents the training steps). Only 250.000 timesteps and 20 minutes of training were necessary to achieve a 100% success rate in the simplified environment. Notice that the mean episodic reward never reaches the value of 1 due to the effect of the discount factor.

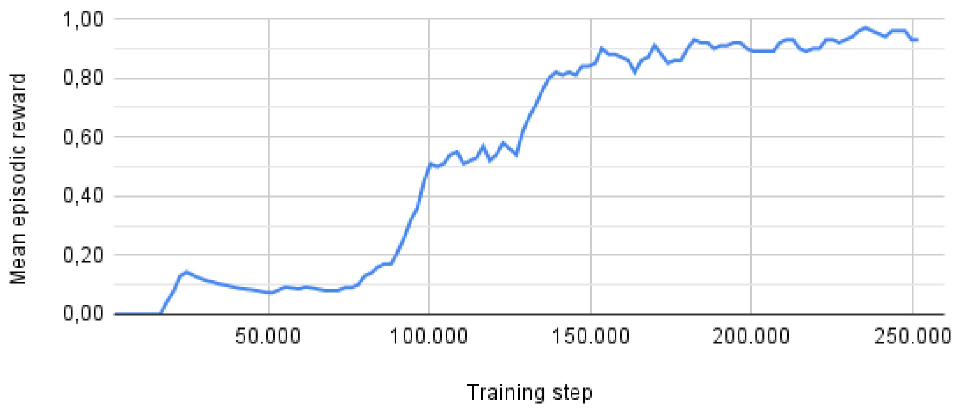

Figure 4.2: Mean episodic reward (averaged over 100 episodes) for the simplified task.

As a further piece of evidence, the evolution of the average episode length (in time steps) is shown in Figure 4.3. The number of steps needed to reach the goal gradually decreases over time. This is due to the discounted return: a reward achieved now is more valuable than a reward that will be gained in the future. That is why the episode length drastically decreases and the agent is encouraged to reach the goal as soon as possible.

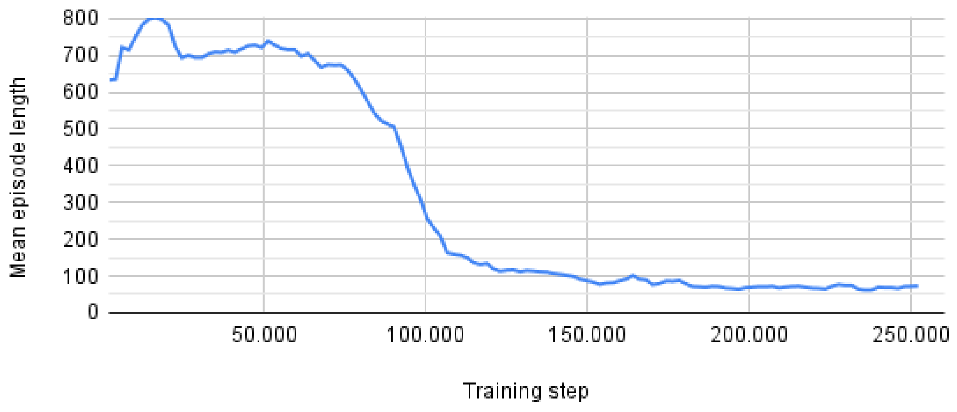

Figure 4.3: Mean episode length measured in steps (averaged over 100 episodes) for the simplified task.

Noticed that, the behavior of the trained robot is in agreement with the intuition. In fact, the agent points straight to the target and reaches it without changing direction. These are all pieces of evidence that the reward function chosen can effectively lead to the desired policy. The next step is to train an agent by including laser readings in the state.

4.4 Training for the standard task

Now, the goal is to train an agent capable of completing the task introduced in Section 3.4, where also laser readings are included in the state, and a negative reward is given if a collision is detected. Again, the process is divided into several stages in which the complexity of the task progressively increases. Doing so ensures a better understanding of the training process and facilitates the identification of possible flaws.

4.4.1 First-generation agent

To get off to a simple start, the robot is placed in an area almost entirely free of obstacles (remember that laser readings are now included). To ensure more generalizability of the final policy, at the beginning of each episode, the initial position of the robot and the target are slightly modified by adding a small random component to their coordinates. In all cases, the agent is placed 3-6 meters away from the goal.

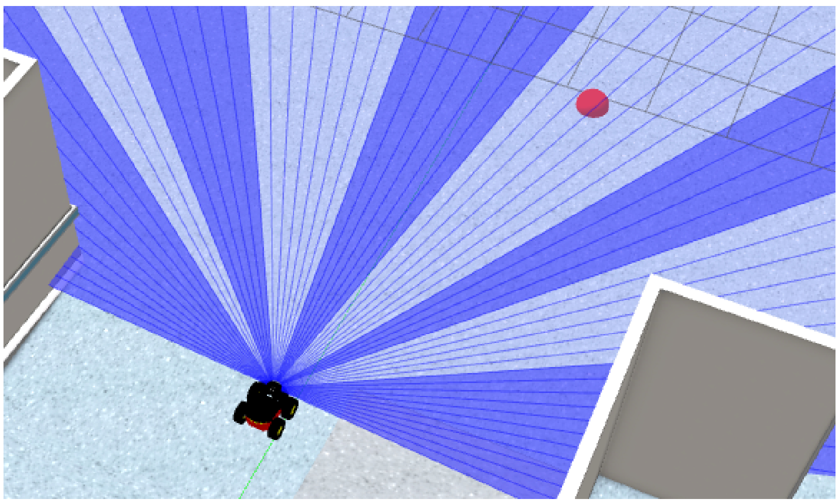

Figure 4.4: Overview of the training setting (the red semi-sphere is the goal).

Given the increased complexity of the task, hyperparameter tuning should be performed before training the agent. After 30 trials of 100.000 steps each, the parameters chosen were the following:

- **learning rate** = 0.0001177 (default: 0.0003);
- **n steps** = 2279 (default: 2048);
- **gamma** = 0.9880615 (default: 0.99);

- **clip range** = 0.1482 (default: 0.02);
- **gae lambda** = 0.9435888 (default: 0.95);
- **ent coef** = 0.0000968 (default: 0.0);
- **vf coef** = 0.6330533 (default: 0.5).

Ultimately, the maximum number of timesteps was reduced to 300 since that is still more than enough.

Greater complexity increases the number of training steps required to develop a policy with a high success rate. In fact, training an agent in this setting required more than 3.5 million steps and nearly 4 hours of training. Figure 4.5 shows the trend of the episode reward mean during training.

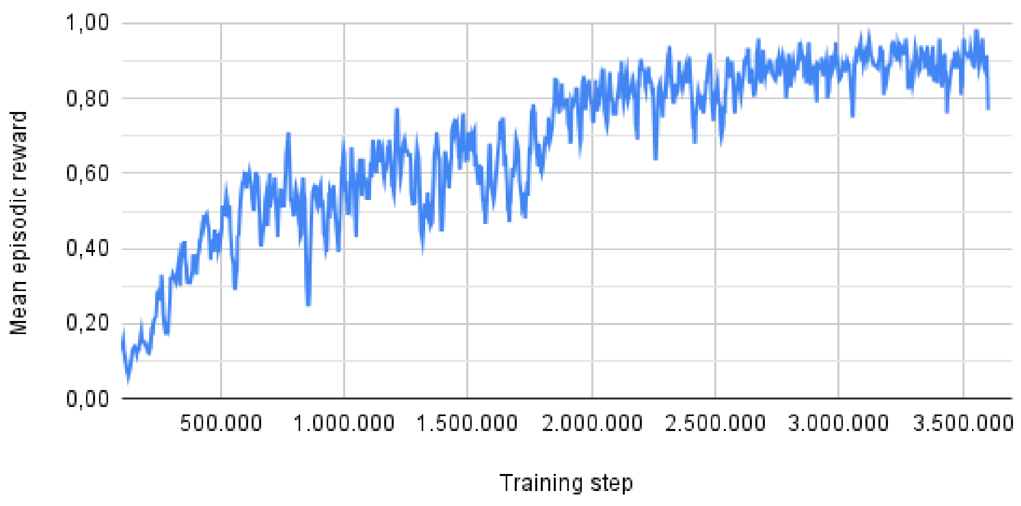

Figure 4.5: Mean episodic reward (averaged over 100 episodes) during first-generation agent training.

As expected, the average reward is much more volatile since the position of the robot and target is slightly changed at the beginning of each episode. However, the trend remains positive throughout training. In Figure 4.6, the mean episode length over training can be observed.

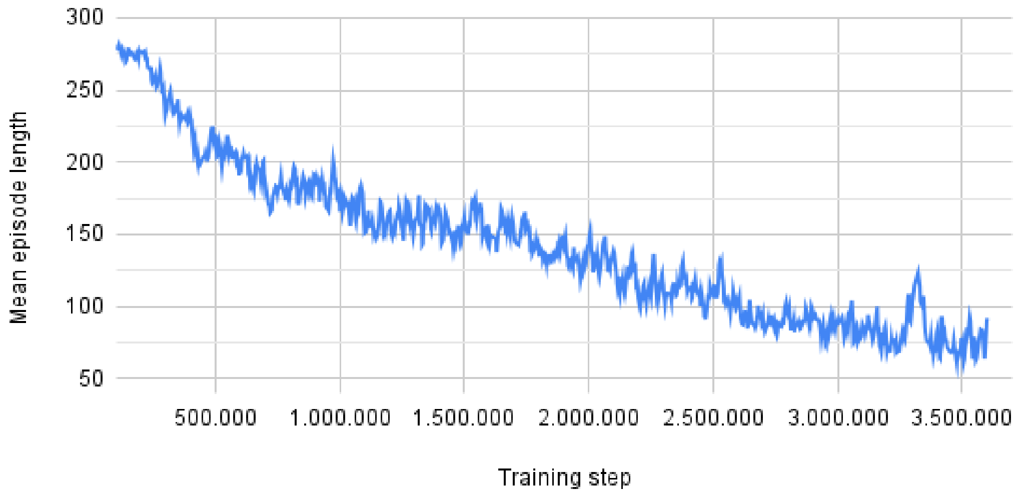

Figure 4.6: Mean episode length measured in steps (averaged over 100 episodes) during first-generation agent training.

The trained agent shows very similar performance to the trained agent for the simplified task. It is also capable of reaching more distant targets, but only if those targets are placed far away from obstacles and the path is mostly clear.

The training was performed in an almost free-obstacle area (see Figure 4.4), for this reason, the agent is able to change its orientation to direct itself toward the target and can approach the target if the path is clear. However, as soon as an obstacle is detected by the laser, the robot stops its motion, even if the obstacle is not so close. The trained agent is greatly overfitted on examples where the obstacles are not close by.

Clearly, for the purpose of this thesis, the agent is required to accomplish more complex behaviors. It should recognize different obstacle patterns and generate a sequence of commands that allows the robot to avoid the obstacle by following the fastest path to the goal.

4.4.2 Second-generation agent

To train an agent that recognizes different patterns of obstacles through its laser readings, the training process is performed on a more heterogeneous set of episodes. Many different locations are defined at the initialization of the Gym environment by specifying the coordinates of the initial robot position and of the target.

When a new episode starts, a new location is chosen randomly and both the robot and the target are positioned in that location. Each location has a different feature. For instance, in one location the robot has to pass through a door. In another one, it has to reach a goal very close to a wall. In addition, better generalizability was achieved by slightly varying the initial position of the robot and the position of the goal in each episode, as done for the first-generation agent. Figure 4.7 shows some of the locations used during the training process inside the hospital world. Notice that there are multiple robots in the image just for the sake of illustration. In reality, only one agent at a time is used during training.

As usual, the hyperparameters were adjusted before training was carried out. The best parameters found to run the training algorithm with this setting are listed below.

- **learning rate** = 0.00005 (default: 0.0003);
- **n steps** = 20480 (default: 2048);

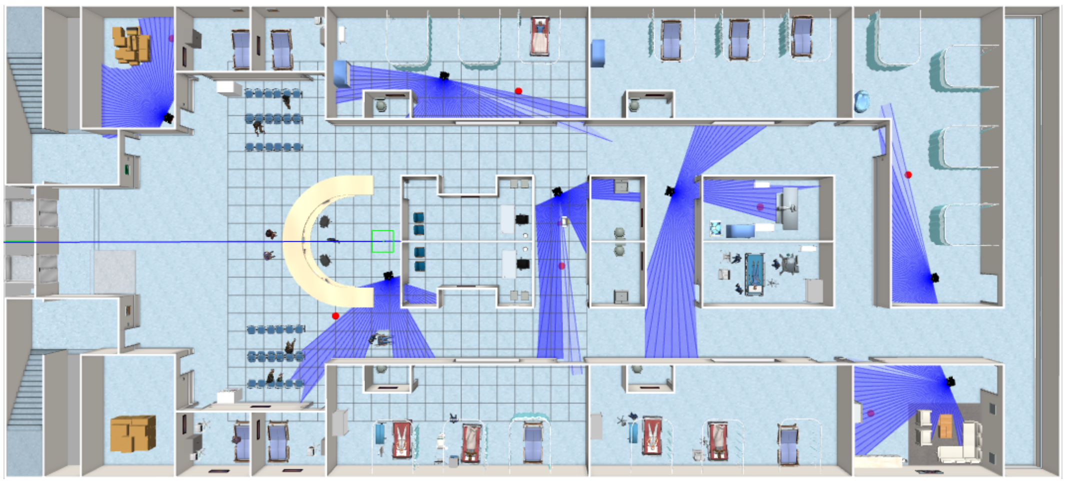

Figure 4.7: Overview of the second-generation training setting. Different locations correspond to different pairs of initial robot positions and target positions.

- **gamma** = 0.9880615 (default: 0.99);
- **clip range** = 0.1482 (default: 0.02);
- **gae lambda** = 0.9435888 (default: 0.95);
- **ent coef** = 0.0004 (default: 0.0);
- **vf coef** = 0.6330533 (default: 0.5).

A big change can be seen in the number of steps to perform per update (n steps), the parameter has been increased by 10 times from the default value. This makes sense because now the episodes could be very different from each other. If the policy is updated on a small set of episodes (a low number of timesteps), the extrapolated information may not take into account features that are present in different episodes. Thus, the new policy might perform very well on the examples already seen, but poorly on others. Setting a large *n steps* ensures more balanced policy updates.

After more than 50 million steps and 60 hours of training, the agent has learned to reach the target while avoiding collision most of the time. Figure 4.8 illustrates the mean episodic reward during training.

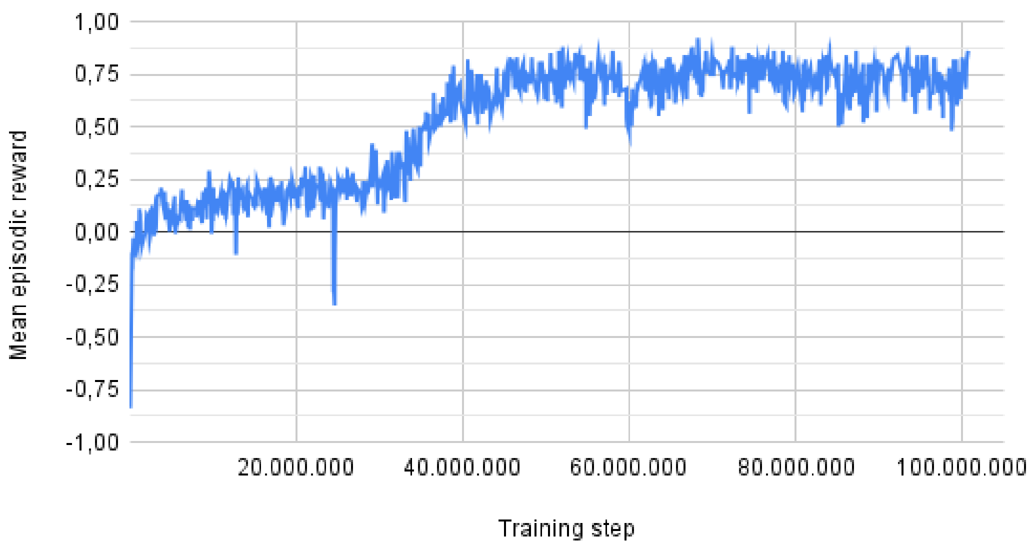

Figure 4.8: Mean episodic reward (averaged over 100 episodes) during second-generation agent training.

After the 50 million steps threshold, the agent performance does not improve anymore. The same applies to the mean episode length, reported in Figure 4.9.

Because of the increased complexity of this task, the success rate is slightly decreased and the robot's movement is very sensitive to laser readings. This results in continuous changes in the orientation of the robot, even if the target is right in front of it.

On the other hand, the trained agent is much more capable of generalizing. The robot can reach targets in places that have not been explored before, it is able to pass through doors, it can overcome obstacles right in the middle of the path, and it is able to stop in front of walls if it cannot find a viable way to reach the target.

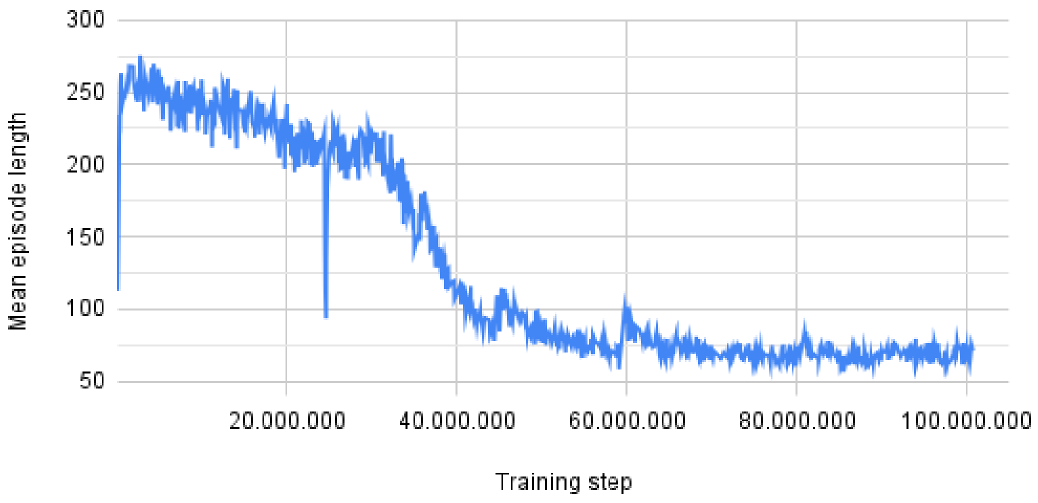

Figure 4.9: Mean episode length measured in steps (averaged over 100 episodes) during second-generation agent training.

4.4.3 Risk-seeker agent

In order to make the robot less sensitive to far away obstacles, another agent was trained by using a different reward function, as follows:

$$r_t = \begin{cases} 1 & \text{if } r < 0.4 \quad \text{(goal reached)} \\ -0.1 & \text{if } \min_i o_i < 0.25 \quad \text{(obstacle collision)} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Because the reward for reaching the target is now 10 times greater than the penalty for colliding with an obstacle, the agent should be encouraged to move toward the target even if LIDAR readings indicate that an obstacle is in its proximity.

The training setting was left as for the second-generation agent, only the reward function was changed. Good results were obtained after 40 million training steps and 42 hours. The curves of the mean episodic reward and the mean episode length during training are pretty similar to the ones of the second-generation agent.

As expected, the resulting agent is less sensitive to laser readings. If the target is directly in front of the robot, the robot will move toward it without repeatedly changing direction. On the other hand, such an agent shows worse performance for more challenging tasks, such as moving through narrow passages (e.g., a door).

In Figure 4.10, the performance of the second-generation agent and the risk-seeker agent on 1000 episodes are shown. To obtain these data, the two agents were tested on the same set of examples used during training, which means that in each episode the robot and target were placed in the same areas used for training. The first-generation agent performance are not shown because it was trained on a different set of examples. Note that an episode is considered successful if the robot reaches the goal, while it is considered failed if the robot collides with an obstacle. As already mentioned, an episode is truncated if the robot does not reach the target in a predefined number of steps.

Both agents show a high success rate with the risk-seeker agent performing slightly better. Moreover, the risk-seeker agent leads to zero truncated episodes. This is understandable because, since the reward for achieving the goal is much higher than the penalty for failing, the agent prefers to move and take the risk of hitting an obstacle instead of ending up in a truncated episode with zero reward.

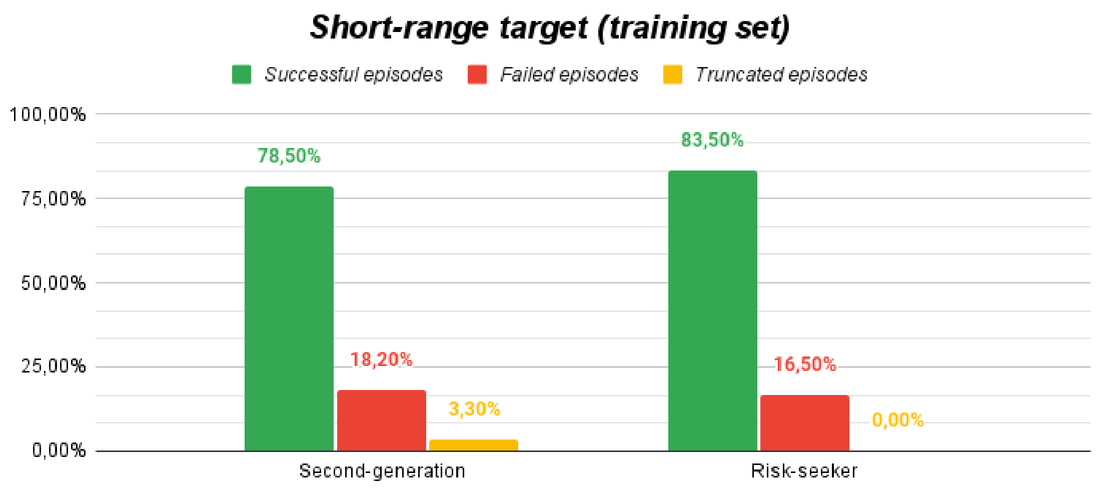

Figure 4.10: Agents performance on the same examples used for training

Chapter 5

Numerical Results

This chapter presents a numerical validation of the agents developed during the study: first-generation, second-generation, and risk-seeker. In Section 5.1, the results obtained on a series of episodes with short-range targets are discussed. Section 5.2 presents simulations on a set of episodes with long-range targets.

5.1 Short-range target

First, the agents were tested on the same task for which they were trained. This time, however, different areas are used to position the robot and the target with respect to the areas used for training. In this way, agents can be tested on never-before-seen examples (test-set) and their generalizability can be evaluated. In all cases, the target is placed no more than 10 meters away from the robot to assess its ability to reach goals at a short distance.

The three agents were tested on a set of 1000 episodes with a maximum length of 300 steps. Figure 5.1 shows the results obtained.

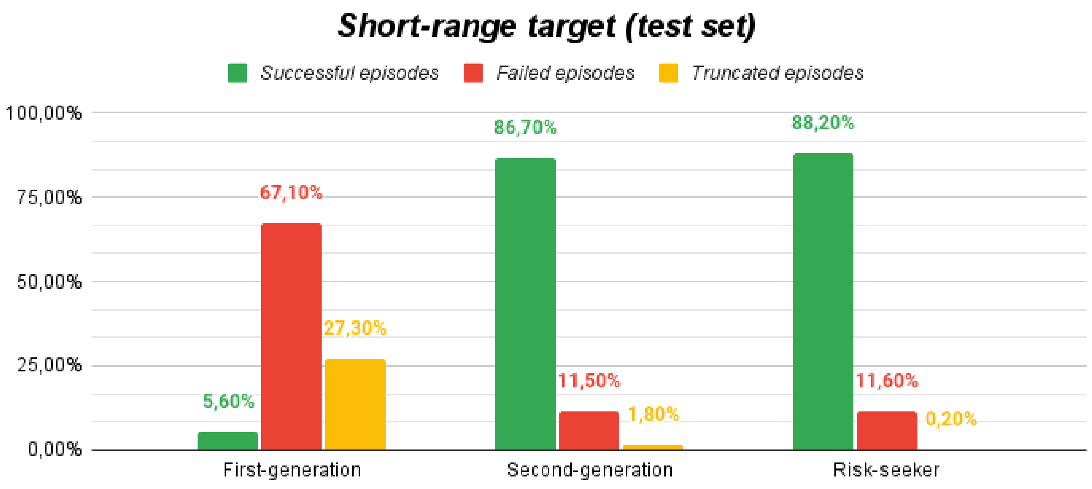

Figure 5.1: Agents performance on a new set of examples with short-range targets (<10m)

The first-generation agent shows poor performance with a success rate just above 5 percent. Although it performs very well on the examples it was trained on (see Section 4.4.1), it demonstrates low generalizability and is unable to avoid obstacles.

In contrast, the other two agents succeed more than 80% of the time, while they fail in almost all other cases. In this evaluation, there are no significant differences between the second-generation agent and the risk-seeking agent.

5.2 Long-range target

Now, the agents are tested on a series of examples where the target is placed farther away from the robot, up to 20 meters distant. At the beginning of each episode, the robot and the target are placed in one of 5 different areas of the hospital. As in training, a random contribution is added to the initial robot and target coordinates to make the episodes different from each other.

Again, each agent was tested on a set of 1000 episodes. Figure 5.2 illustrates the result obtained.

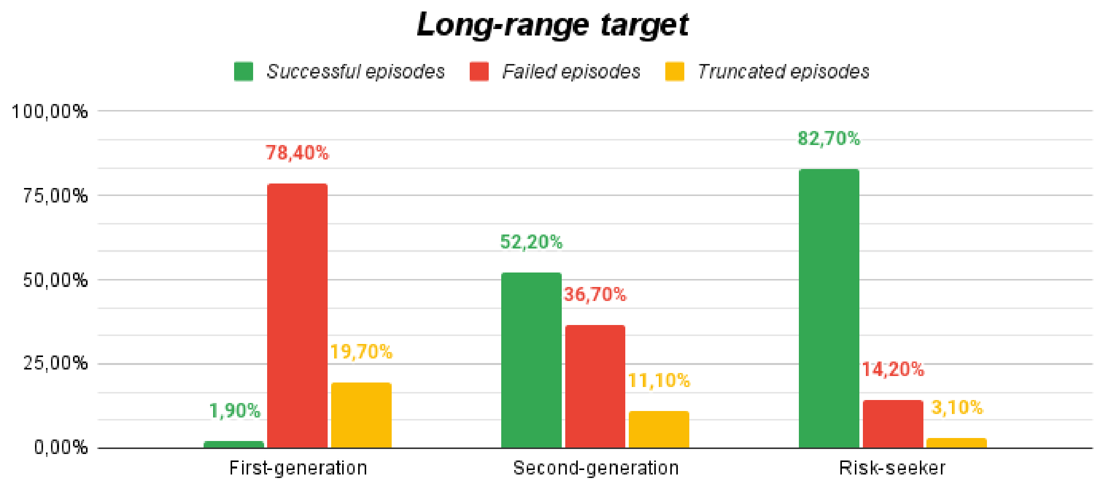

Figure 5.2: Agents performance on a set of examples with long-range targets (>10m)

The first-generation agent once again has a low success rate but, in this case, there is a significant difference between the second-generation agent and the risk-seeker agent. The latter performs much better and maintains the same rates obtained in the test with short-range targets.

The second-generation agent might perform worse because it is overfitted on examples with short-distance targets. It was trained for more than 100 million steps, but the average episodic reward did not increase after the 50 millionth step (see Section 4.4.2).

Chapter 6

Conclusions and Future Work

The goal of this study was to train a reinforcement learning agent capable of accomplishing a path planning task in a 3D environment. Such an agent could be employed in common path planning algorithms to connect adjacent configurations (local planner), but it could also be used during the execution phase to generate a feasible sequence of motion controls that would lead the robot toward the goal without hitting any obstacle.

The obtained results show that reinforcement learning is a viable option when it comes to developing fairly reliable local planners. Trained agents are able to reach targets at short-to-medium distances while avoiding obstacles more than 80% of the time.

In addition, the study showed how ROS2, Gazebo, OpenAI Gym, and Stable Baselines 3 could be used together to develop a reinforcement learning infrastructure and a 3D simulated environment to train agents for robotic applications.

Future work concerns the development of different reward functions and the integration of the RL agent in a hierarchical planning architecture.

In general, agent performance could be improved by shaping a more specific reward function or by including an even more varied set of episodes in the training setting. However, due to the limitations of the current study, further research is needed to identify a better learning environment setting.

In a hierarchical planning architecture (see Sections 1.3.2 and 1.3.3), a set of waypoints is generated from the free space. Then, a graph is built by establishing connections between the waypoints and, finally, the shortest path to reach the final target is computed. In such a framework, RL agents can be used to establish the connection between pairs of adjacent waypoints. For instance, consider two waypoints a and b. To connect these points in the graph, the robot is placed on a, while b is considered as the target. At this point, the RL agent is employed to lead the robot toward b. If the robot reaches the target without hitting any obstacle, an edge between a and b is placed into the graph. By using this method, the resulting graph will only include edges between waypoints that the RL agent is able to connect.

References

[1] Latombe, J.C. (Dec 31, 1990), *Robot Motion Planning.*

[2] Garaffa, L.C. - Basso, M. - Konzen, A.A. - de Freitas, E.P. (Nov 12, 2021), *Reinforcement Learning for Mobile Robotics Exploration: A Survey.*

[3] Tzafestas, S.G. (2014), *Introduction to Mobile Robot Control*

[4] Schulman, J. - Wolski, F. - Dhariwal, P. - Radford, A. - Klimov, O. (Aug 28, 2017), *Proximal Policy Optimization Algorithms*, Available at `https://arxiv.org/abs/1707.06347v2`.

[5] Faust, A. - Oslund, K. - Ramirez, O. - Francis, A. - Tapia, L. - Fiser, M. - Davidson, J. (May 25, 2018), *PRM-RL: Long-range Robotic Navigation Tasks by Combining Reinforcement Learning and Sampling-based Planning.*

[6] Sutton, R.S. - Barto A.G. (Nov 13, 2018), *Reinforcement Learning: An Introduction, second edition*

[7] Duchoň, F. - Babinec, A. - Kajan, M. - Beňo, P. - Florek, M. - Fico, T. - Jurišica, L. (Dec 27, 2014), *Path Planning with Modified a Star Algorithm for a Mobile Robot*, Available at `https://www.sciencedirect.com/science/article/pii/S187770581403149X`.

[8] Raheema, F.A. - Hameed, U.I. (Dec 11, 2018), *Path Planning Algorithm using D* Heuristic Method Based on PSO in Dynamic Environment*, Available at `https://core.ac.uk/display/235050716?utm_source=pdf&utm_medium=banner&utm_campaign=pdf-decoration-v1`

[9] Spinningup.openai.com (2018), *Key concepts in RL*, (Access date: Mar 10, 2023), `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html`.

[10] Spinningup.openai.com (2018), *Intro to Policy Optimization*, (Access date: Mar 10, 2023), `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`.

[11] Spinningup.openai.com (2018), *Proximal Policy Optimization*, (Access date: Mar 10, 2023), `https://spinningup.openai.com/en/latest/algorithms/ppo.html`.

[12] AWS Robotics (Jul 29, 2021), *aws-robomaker-hospital-world*, (Access date: Dec 15, 2022), `https://github.com/aws-robotics/aws-robomaker-hospital-world`

[13] Wonnacott, D. (Nov 13, 2018), *ros-pioneer3at*, (Access date: Dec 15, 2022), `https://github.com/dawonn/ros-pioneer3at`

[14] Stable Baselines 3 (2022), *Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations*, (Access date: Dec 15, 2022), `https://stable-baselines3.readthedocs.io/en/master/index.html`

[15] Stable Baselines 3 (2022), *PPO*, (Access date: Dec 15, 2022), `https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html`

[16] Open Robotics (2023), *ROS 2 Documentation*, (Access date: Dec 15, 2022), `https://docs.ros.org/en/humble/index.html`

[17] Open Source Robotics Foundation (2014), *Installing gazebo_ros_pkgs (ROS 2)*, (Access date: Dec 15, 2022), `http://classic.gazebosim.org/tutorials?tut=ros2_installing&cat=connect_ros`

[18] Farama Foundation (2022), *Gymnasium documentation*, (Access date: Dec 15, 2022), `https://gymnasium.farama.org/`

Acknowledgements

I would like to thank *Prof. Giannitrapani*, *Prof. Garulli*, and *Dr. Quartullo* for continuously guiding and supporting me in this final stage of my academic journey.

I would like to thank my whole family, especially my parents, for teaching me the arduous discipline of always achieving the goal and without whom I could not have even started my college career. I thank my brother *Mattia* for always being a source of inspiration, guidance, and a great rapper in his spare time. I thank my sister *Sara* and *Bruno* for giving a wonderful addition to the family, little *Caterina*.

I would like to thank all my lifelong friends for all the good times we have shared together and for all those yet to come. Thank you for standing by me, each in their own way, on this intense and exciting journey.

I would like to thank *Ilaria* for randomly coming to that party on December 27 three years ago in Ville di Corsano, because otherwise, I would not have met her. I thank her for the incredible empathy she shows me every day, because she supports me in everything I do, and because she always pushes me to do my best.

Finally, I would like to thank all my fellow students who made this adventure even more fun and even more interesting.