POLITECNICO DI TORINO

Master's Degree in Mechatronics Engineering

Master's Thesis

VR simulation of mobile robots for social behavior learning

Supervisors
Prof. Marcello Chiaberge

Candidate
Asia Ferri

Academic Year 2024-2025

Acknowledgements

Thanks to the PIC4SeR team, for the continuous support and for sharing their knowledge and experience during the development of this project.

To my family and friends, for being by my side throughout the highs and lows of this journey.

Abstract

In robotics, one of the main areas of interest for the research community is the creation of environments where machines and humans can safely coexist. This brings to two main issues: training and testing of robots in a controlled but realistic setting, and their correct navigation in social spaces, while causing no harm to people. Virtual reality allows the simulation of such environments, ensuring safety and allowing the training of robots in areas where errors are controlled and confined. This way, robot behavior can be supervised and demonstrated by the operator, able to enter the training environment with the use of a Virtual Reality headset.

This thesis explores this type of simulation by training a ROS-based four-wheel robot model tasked with navigating a laboratory-like area to reach a specified target, respecting social constraints. This process is carried on using human behavior as a reference, implementing Imitation learning using behavioral Cloning (BC) and Generative Adversarial Imitation Learning (GAIL), and confronting it with Reinforcement learning approaches, in particular Proximal Policy Optimization (PPO). Additionally, a hybrid learning pipeline that combines demonstrations with reinforcement learning is considered. Unity is used as the virtual development platform, alongside ML-Agents toolkit for Neural Network training. The objective is to assess which learning strategy achieves safer and more efficient navigation, and to evaluate the potential benefits of combining human demonstrations with autonomous exploration. The VR framework realized for this thesis can be adapted for future applications and different tasks, that can integrate human robot collaboration or more complex environments.

Contents

Chapter 1

Introduction

The development of robots capable of carrying out tasks autonomously is one of the main subjects of study and research in modern robotics. Robots are increasingly required to operate without humans intervening on their actions to perform complex tasks and to adapt them to different environments. Learning algorithms such as Reinforcement Learning (RL) and Imitation Learning (IL) are widely used to train agents to behave in a desired manner, allowing them to learn from experience or imitate expert demonstrations. These algorithms require training environments where the agent can interact and learn through several trials, receiving feedback on its actions. The specific environment where the robot has to operate is often unknown, not reachable or too dangerous for humans. In these cases, it is fundamental to create safe and controlled environments that allow the developers to test, train and get insights on the robot behavior. For this purpose, simulated environments are widely used in robotics, since they allow creating virtual scenarios where the robot can interact and learn without risks or restrictions.

This thesis focuses on the creation of an environment for training robots using Unity ML-agents, a framework that integrates the Unity game engine with machine learning algorithms. The main goal is to analyze the potential of the algorithms provided by the framework, such as Proximal Policy Optimization (PPO), Behavioral Cloning (BC), and Generative Adversarial Imitation Learning (GAIL), in training a four-wheeled robot to navigate in a simulated social environment. The combination of reinforcement learning and imitation learning techniques is also explored, integrating prior training with IL algorithms followed by further training with RL algorithms. The performance of the various algorithms is evaluated through a series of tests that measure their ability to enable effective robot navigation with obstacles and people avoidance.

Following an introduction to the related work on the use of Unity for robot training in Chapter 2, this thesis analyzes in Chapter 3 the algorithms used for training the robot and the tools provided by Unity ML-agents. Chapter 4 describes the components of the training environment with the settings thereof. Finally, Chapter 5 shows the results obtained from the training sessions, analyzing the performance of the different algorithms according to a certain set of tests. Other configurations that have been first tested and

then discarded during the development of the environment are reported in the Appendix 7 in order to show the evolution of the project and the performance with different settings.

Chapter 2

Related work

2.1 Introduction to the topic

Simulation plays a crucial role in both robot programming and testing, as well as in integrating robots into social contexts. Modern software tools such as game engines allow researchers to create virtual environments that can recreate settings as close to reality as possible, providing safe and flexible training grounds for robotic systems. In this thesis, Unity game engine is employed to simulate and test a robotic system operating in a social environment. The work combines the capabilities of ROS2, a widely used framework for robotic control, with Unity's powerful simulation features. The goal is to train and assess models capable of exhibiting socially acceptable behavior within realistic virtual scenarios. Developing such training environments is essential for the safe and effective integration of robots into everyday human settings. Simulations allow testing complex behaviors without physical risk, accelerating development, and improving the reliability of human-robot interactions. The use of Unity as a simulation platform has gained significant attention in recent years due to its flexibility, visual realism, and compatibility with robotic frameworks. The following section reviews key studies exploring how Unity has been employed to create virtual environments for robot training, testing, and behavior validation.

2.2 Unity as a Simulation Platform

The main challenge in creating simulation environments is achieving a high degree of realism, which often leads to complex and cumbersome interfaces. Many existing tools provide only approximate physical simulations and offer limited flexibility in modelling diverse environments, such as lunar landscapes or underwater scenarios. A more modern perspective suggests that game engines are particularly suitable for research and robot development, as they provide user-friendly interfaces, customizable physics parameters, built-in animation tools, texture options, and other features that facilitate realistic and adaptable simulations.

Juliani et al. [1] disclose Unity as a suitable platform for the simulation and training of intelligent agents, highlighting its training capabilities and flexibility for AI research.

As they note, "many of the existing environments provide either unrealistic visuals, inaccurate physics[..]. We argue that modern game engines are uniquely suited to act as general platforms". Among the advantages of using Unity, the most important are the ability to create multi-agent scenarios that allow developers to test multiple models in the same context, and the support for VR/AR applications, which enables interactive engagement between users and agents.
Unity also allows implementing the concept of "social complexity" referred to by Juliani et al. [1], achieved through the interaction of AI agents in shared virtual spaces. These environments provide the conditions for the emergence of complex behaviors and the study of how agents learn to react appropriately in social or collaborative situations. It provides a foundation for research focused on human-robot interaction and socially aware robotic systems, through both the use of human animations to simulate human presence and virtual reality applications that enable real user interaction within the simulated scene. Cleaver et al. [2] followed this approach by developing the HAVEN framework, a Unity-based environment designed to study human-robot interaction using augmented and virtual reality, allowing users to directly engage with virtual robots in socially meaningful scenarios.
Moreover, Unity's flexibility enables the replication of a wide variety of environments with specific physical properties and environmental conditions. For example, Seyyedhasani et al. [3] used Unity-based virtual reality to simulate vineyard terrains for evaluating the navigation and performance of agricultural robots in complex natural settings. Similarly, El-Müftü and Gür [4] developed a Unity-based underwater simulation framework that allows testing robotic systems under fluid dynamics and low-visibility conditions. Finally, Franchini et al. [5] suggested a comprehensive virtual reality framework for lunar exploration that integrates Unity and ROS2, thus enabling both human and robotic agents to perform mission-oriented tasks and cooperative behaviors in simulated extraterrestrial environments.

Collectively, these works highlight the versatility of Unity as a simulation platform capable of accurately reproducing a wide range of diverse and realistic environments. They also demonstrate its growing importance as a research tool for robotics, not only for motion and control but also for studying perception, learning, and social interaction in complex, dynamic scenarios.

2.3 Unity and ROS/ROS2 integration

Another characteristic that makes Unity a suitable environment for research and robotic development is its ability to integrate with the Robot Operating System (ROS) and its successor, ROS2. ROS provides a standardized framework for controlling robotic systems, managing communication between sensors, actuators, and control nodes. Integrating ROS with Unity allows virtual robots to be operated using the same software employed on physical robots, enabling testing and debugging in a safe environment. Robotic models can be designed in detail using external modelling software, such as Blender, then imported into Unity. ROS or ROS2 control nodes can run either locally or on an external device and connect to the simulation through dedicated communication protocols.

This setup enables researchers to simulate realistic robotic behavior and test robots in environments that are difficult or impossible to access under real-word conditions.

The HAVEN environment [2] is an example of such an integration. It employs a Turtle-Bot2 robot model connected to a ROS control system that uses Unity's NavMesh for autonomous navigation and interaction with the user in collaborative tasks. Likewise, Franchini et al. [5] used a Jackal robot model to simulate a lunar rover capable of performing autonomous missions such as navigating towards a target or following a user avatar within a virtual environment.
El-Müftü and Gür [4] integrated a ROS system in the URSULA robot simulator, where "each of the four soft robotic limbs in URSULA is represented as a series of connected rigid objects to simulate continuous flexible motion. The position and rotation of the virtual limb segments are continuously updated on the basis of data received from ROS via a WebSocket connection." This illustrates how ROS-Unity integration supports the real-time synchronization of physical models and simulated kinematics.
Ye et al. [6] further validated the Unity-ROS2 integration by reproducing behaviors tested in simulation on a physical robot platform. Their results show that behaviors trained within the Unity environment can be successfully transferred to real-world scenarios. This sim-to-real capability highlights the reliability of Unity-based simulations for developing and validating robotic behaviors prior to deployment.

In brief, the integration of ROS and ROS2 with Unity provides a powerful infrastructure for the simulation, testing, and training of softwares, transferable to real robotic systems.

2.4 ML-Agents for robotic training

The Unity ML-Agents Toolkit provides a robust framework for training intelligent agents within simulated environments. It enables robotic systems to acquire complex behaviors through repeated interaction with their surroundings, using learning algorithms such as Reinforcement Learning (RL) and Behavioral Cloning (BC). When integrated with ROS2, ML-Agents allows robots to learn through trial and error in safe and controlled environments, reducing the risks and costs associated with physical testing.
Ye et al. [6] analyzed the use of ML-Agents to show how the Unity's package can be employed to train multiple different robotic behaviors. In their work, many reinforcement learning policies were trained in simulation using Unity's physics and perception models, and then deployed through ROS2 interfaces to real-world robots. The consistency observed between virtual and physical executions validates the reliability of simulation-based reinforcement learning and confirms Unity's potential as a platform for simulation-to-reality applications. Ye et al. [6] utilized Unity's multi-agent capabilities to simulate human-robot interactions, where agents learned appropriate social responses through reward-based adaptation. This framework not only supports learning navigation and manipulation skills but also extends to context-aware behaviors, essential for human-robot interaction research. Beyond robotic tasks, game engine-based reinforcement learning also finds application in a wide range of other domains. For example, Murdivien and Um [7]

used a Unity simulation to train agents for a three-dimensional bin packing task, demonstrating how reinforcement learning can solve spatial logistics problems. While their focus is not strictly on robotics, their work highlights how ML-Agents-like approaches can be used for tasks that require spatial reasoning, physics interactions and environment dynamics, which features are shared by robotic simulation.
Overall, the ML-Agents toolkit is a bridge between simulation and real-world deployment. By combining reinforcement learning capabilities with the ROS2 communication layer, it allows for efficient, safe, and repeatable training of robotic agents before their deployment in physical and social environments.

2.5 Thesis Positioning

Considering all the studies discussed above, it becomes apparent that Unity, in combination with ROS2 and the ML-Agents Toolkit, offers a robust environment for robotic simulation, training, and validation. Grounded on these premises, this project aims to address a Unity-ROS2 framework designed to train and evaluate robotic agents capable of exhibiting socially acceptable behaviors within human-centered virtual environments. Both Reinforcement Learning and Imitation Learning algorithms from the ML-Agents toolkit are implemented and analyzed in scenarios where human presence is either simulated or incorporated through virtual reality interactions.

Chapter 3

Theory

3.1 Learning Algorithms

Learning algorithms in machine learning are sets of rules that allow an agent or model to assimilate a behavior towards a task based on experience or data. Unlike traditional programming where instructions are provided for every possible scenario, machine learning enables systems to discover patterns and strategies from the data received. These methods can be classified as follows:

- **Supervised Learning**, where models learn from labelled datasets given by the user;
- **Unsupervised Learning**, which aims to identify patterns in unlabeled data;
- **Reinforcement learning**, where an agent interacts with an environment to maximize cumulative reward over time;
- **Imitation Learning**, where an agent learns to replicate expert behavior from demonstration data, without manually designed reward signals.

In particular, Reinforcement Learning (RL) is suitable for robot learning and training. It allows learning how complex tasks can be carried out without the user's manual control, but it requires an environment in which the agent can explore and learn through the interaction with its surroundings. Virtual simulation fits well into this approach since it allows safe trial and error learning and realistic replication of any type of environment.

In RL, the agent observes a state s_t, chooses and takes action a_t, receives the corresponding reward r_t and then transitions to a new state s_{t+1}. The objective of the agent is to learn a policy $\pi(a|s)$ that maximizes the expected (average) cumulative reward $J(\pi)$, expressed as

$$J(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{3.1}$$

where γ is the discount factor. This value determines how much future rewards are important compared to immediate rewards, and takes a value between 0 and 1.

In order to determine the quality of a state or action during training, four functions are defined in RL:

- **Value function $V^{\pi}(s)$:** is the expected benefit starting from state s and following policy π

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t \;\middle|\; s_0 = s\right]. \tag{3.2}$$

- **Action-Value Function $Q^{\pi}(s,a)$:** is the expected benefit of choosing action a being in state s and following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t \;\middle|\; s_0 = s,\, a_0 = a\right]. \tag{3.3}$$

- **Optimal Value Function $V^{*}(s)$:** is the expected benefit from starting from state s and always acting according to the optimal policy

$$V^{*}(s) \;=\; \max_{\pi} \; \mathbb{E}_{\pi}\big[\, R(\tau) \mid s_0 = s \,\big] \tag{3.4}$$

- **Optimal Action-Value Function $Q^{*}(s,a)$:** is the expected benefit from starting from state s, taking action a, and after that action always acting according to the optimal policy

$$Q^{*}(s,a) \;=\; \max_{\pi} \; \mathbb{E}_{\pi}\big[\, R(\tau) \mid s_0 = s,\; a_0 = a \,\big] \tag{3.5}$$

The value functions need to satisfy the Bellman equations, which define a recursive structure to make the function consistent step by step. The concept underlying the Bellman equations is that the value of the starting point needs to be at any time the reward expected from staying in that point, increased by the value of the point where the agent will move next. The Bellman equations are defined as

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(\cdot|s),\, s'\sim P(\cdot|s,a)}\big[\, r(s,a) + \gamma\, V^{\pi}(s') \,\big], \tag{3.6}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a),\, a'\sim\pi(\cdot|s')}\big[\, r(s,a) + \gamma\, Q^{\pi}(s',a') \,\big]. \tag{3.7}$$

while the optimal Bellman equations are defined as

$$V^{*}(s) = \max_{a} \, \mathbb{E}_{s'\sim P(\cdot|s,a)}\big[\, r(s,a) + \gamma\, V^{*}(s') \,\big], \tag{3.8}$$

$$Q^{*}(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a)}\big[\, r(s,a) + \gamma \, \max_{a'} Q^{*}(s',a') \,\big]. \tag{3.9}$$

The difference between the Bellman equations and their optimal counterpart is that in the second ones the maximum over actions is taken. This is justified since the agent needs to take the action that leads to the highest result in order to act optimally.

Types of RL algorithms are Proximal Policy Optimization algorithm (PPO)(3.1.1) and Soft Actor-Critic algorithm (SAC) [8].

On the other hand, Imitation Learning (IL) aims to learn by observing demonstrations given by experts. It offers an alternative to reinforcement learning when designing a reward function is complex, and instead a demonstration by an expert is available.
IL uses a demonstration dataset collected from an expert policy π_E, in the form of trajectories of state–action pairs. Its objective is to find a policy $\pi(a|s)$ that from state s brings to action a, matching expert's policy $\pi_E(a|s)$.

Assume $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ to denote a trajectory generated by the expert policy $\pi_E(a|s)$ interacting with the environment according to the transition dynamics $P(s_{t+1}|s_t, a_t)$. The imitation learning problem can be defined as minimizing the difference between the trajectory distributions of the expert and that learned by agent:

$$\min_{\pi_\theta} \mathcal{L}(\pi_\theta, \pi_E) \quad \text{such that} \quad \mathcal{L}(\pi_\theta, \pi_E) = D_{\text{div}}(p_{\pi_\theta}(\tau) \,\|\, p_{\pi_E}(\tau)) \tag{3.10}$$

where $D_{\text{div}}(\cdot\|\cdot)$ is a statistical divergence measure such as Kullback-Leibler divergence, Jensen-Shannon divergence, or Wasserstein distance. This formulation expresses IL as a distribution matching problem, where the goal for the learned policy is to visit similar state-action visitation frequencies as the expert.

The advantage of IL lies in its ability to add prior knowledge in the form of expert demonstrations, reducing the amount of exploration required compared to reinforcement learning. This is particularly useful in domains where random exploration is unsafe, expensive, or time consuming, such as robotics, autonomous driving, or human-robot interaction. By directly imitating demonstrated behaviors, an agent can quickly achieve optimal performance without the need to autonomously discover reward structures.

However, IL introduces several theoretical and practical challenges. Since demonstration data are typically collected under the expert policy, the learner is trained on a fixed distribution of states $p_{\pi_E}(s)$. When the learned policy π_θ deviates from this distribution, it may encounter unseen states, leading to prediction errors. This phenomenon is known as covariate shift or distributional shift. In order to mitigate this issue, many modern IL approaches integrate reinforcement learning or adversarial training components that allow the agent to refine its behavior through interaction with the environment and exploration of unknown states. Types of IL algorithms are behavioral Cloning (BC)(3.1.2)and Generative Adversarial Imitation Learning (GAIL)(3.1.3).

Deep Learning extends traditional machine learning by integrating deep Neural Networks (NN) as function approximators. NN can approximate complex inputs such as images, big data, or joint positions to actions or value estimates. Using NN, the learning process can handle continuous structured environments and therefore high complexity observations. All algorithms implemented by ML-agents package use Neural Networks, with characteristics that can be modified and tailored around a specific training (as described in chapter 3.3.1).

3.1.1 PPO - Proximal Policy Optimization Algorithm

PPO (Proximal Policy Optimization) is a family of Reinforcement Learning (RL) algorithms based on policy-gradient methods. They are model-free reinforcement learning

algorithms, i.e., they learn a policy by interacting with an environment and gathering data, rather than trying to first build a model of the environment dynamics. PPOs try to improve the policy $\pi_\theta(a|s)$ by tuning parameters θ in the direction that increases the expected reward. The base algorithm is an on-policy actor-critic method, i.e., it simultaneously learns a policy (the actor) and a value function (the critic) using trajectories generated by the current policy, rather than from previously stored experiences. The main feature of the PPOs algorithms is that they improve stability by limiting the deviation between distributions of updated and old policies.
PPO derives from TRPO [9], Trust Region Policy Optimization. The common objective of these two types of algorithms is to find the biggest improvement on a policy using the data collected and without causing performance collapse. TRPO and PPO are on-policy algorithms, i.e., all pieces of data used to update the policy are collected using the same policy.

TRPO aims to maximize an objective function, considering a size constraint on the policy update in order to avoid the new policy from being too distant from the previous one. For this purpose, it needs to solve a constrained optimization problem, too complex and not suitable for noisy systems or policies that share parameters with auxiliary tasks. PPO keeps the reliability and data efficiency of TRPO using only first-order optimization, making the algorithm simpler, faster, and more scalable. There are two main variants of the PPO algorithm: the PPO-Penalty and PPO-Clip.

PPO-Penalty algorithm keeps the constraint of TRPO, but instead of making it a hard constraint it defines a soft, variable constraint to keep the distance between policies near a target value. It is referred to as a penalty method since it regulates the penalty coefficient to scale the statistical distance (KL-divergence) in the objective function.

On the other hand, PPO-Clip completely removes the update constraint and bases its algorithm on clipping the objective function to keep the new policy from moving too far away from the old one. ML-Agents package implements PPO-Clip, which will generally be referred to as PPO hereafter in this thesis.

The PPO clipped surrogate objective is defined as

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\, \hat{A}_t,\ \text{clip}(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)\, \hat{A}_t \right) \right] \tag{3.11}$$

where:

- θ - policy parameters, weights of the neural network that defines the policy
- $\hat{\mathbb{E}}_t$ - empirical expectation over timesteps, i.e., the average of this expression over all the timesteps in the collected batch.
- $r_t(\theta)$ - probability ratio between new and old policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{3.12}$$

  It is an index of how much the new policy differs from the old one, i.e., how much the probability of taking an action a_t in state s_t has changed.

- $\hat{A}_t$ - estimated advantage function at time step t, representing how much better (or worse) an action a_t was compared to the expected value. It is calculated as the difference between the actual reward ($\hat{R}_t$) and the expected value of the value function V_{ϕ_k} computed starting form state s_t

$$\hat{A}_t = \hat{R}_t - V_{\phi_k}(s_t) \tag{3.13}$$

  When multiplied by $r_t(\theta)$, it forms the normal policy gradient term ($r_t(\theta)\hat{A}_t$), that indicates the scaled term to update the policy.

- ϵ - hyperparameter controlling the clipping range (usually 0.1 or 0.2). It limits the change in $r_t(\theta)$ preventing large policy updates that could destabilize learning.

- The clip function in PPO is defined as:

$$\text{clip}(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon) = \begin{cases} 1+\epsilon, & \text{if } r_t(\theta) > 1+\epsilon, \\ 1-\epsilon, & \text{if } r_t(\theta) < 1-\epsilon, \\ r_t(\theta), & \text{otherwise.} \end{cases} \tag{3.14}$$

  Clipping aims to limit the extent to which the probability ratio $r_t(\theta)$ is allowed to change. The clip function enforces the constraint:

$$1-\epsilon \leq r_t(\theta) \leq 1+\epsilon \tag{3.15}$$

  preventing updates that would move the new policy too far from the old policy.

Since PPO chooses the minimum between the policy gradient term and its clipped version, the update is limited by the clipping range, preferring small updates on big ones. Expanding the expectation calculation and starting from the initial policy parameters (θ_0) and the initial value function parameters (ϕ_0) as inputs, the pseudocode is

Algorithm 1 PPO-Clip

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Run the policy $\pi_k = \pi(\theta_k)$ into the environment and collect the trajectories

$$\mathcal{D}_k = \{\tau_i\}$$

3: Compute the discounted rewards

$$\hat{R}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

.

4: Compute advantage estimates

$$\hat{A}_t = \hat{R}_t - V_{\phi_k}$$

5: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(r_t(\theta)\, \hat{A}_t,\ \text{clip}(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)\, \hat{A}_t\right),$$

typically via stochastic gradient ascent with Adam [10].

6: Update the value function V_{ϕ_k} by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm. [11]

7: **end for**

Since expression 3.11 is complex to implement and to read, an easier simplified version has been designed:

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\, \hat{A}_t,\ \ g(\epsilon, \hat{A}_t)\right) \tag{3.16}$$

where

$$g(\epsilon, \hat{A}_t) = \begin{cases} (1+\epsilon)\hat{A}_t, & \hat{A}_t \geq 0, \\ (1-\epsilon)\hat{A}_t, & \hat{A}_t < 0. \end{cases} \tag{3.17}$$

By analysing the sign of the advantage, when $\hat{A}_t > 0$, the agent has taken a beneficial action. Increasing $r_t(\theta)$ is desirable, since it indicates the probability ratio of the new policy with respect to the old one. However, if $r_t(\theta)$ becomes greater than $1 + \epsilon$, the clipped objective prevents further improvement of the loss.

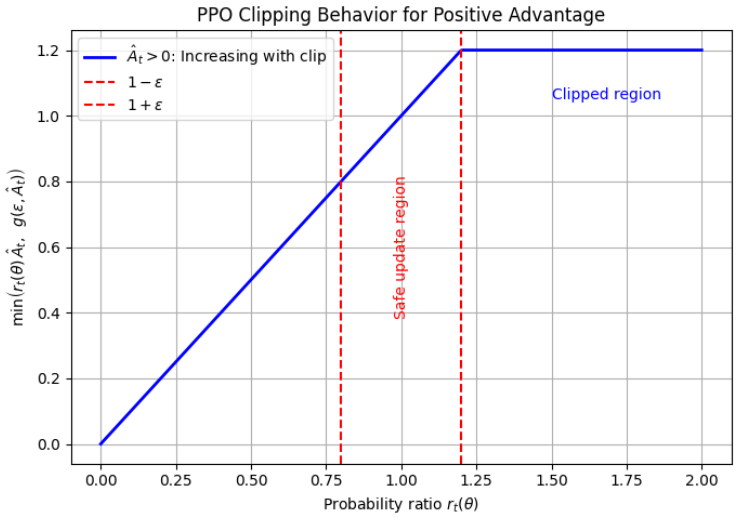


Figure 3.1: PPO-Clip positive advantage

On the other hand, if $\hat{A}_t < 0$, the action performed worse than expected and the optimization decreases $r_t(\theta)$. Once $r_t(\theta)$ falls below $1 - \epsilon$, the objective again is stopped providing additional benefit, preventing an overly aggressive reduction in probability.

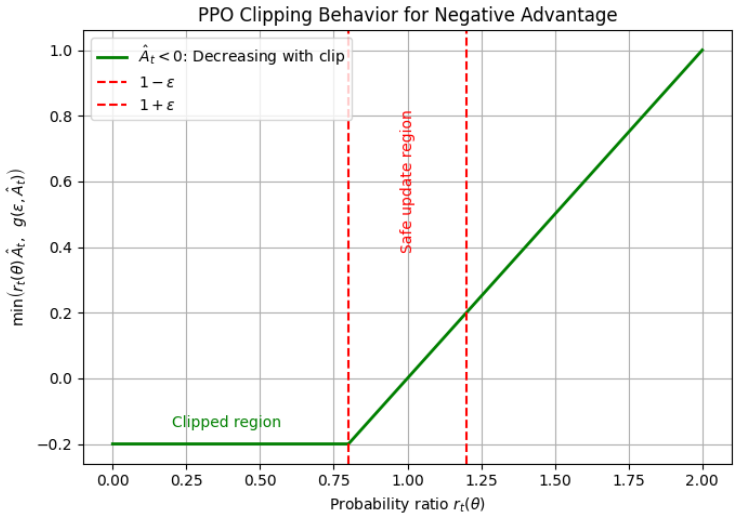


Figure 3.2: PPO-Clip negative advantage

In summary, the clipping term shapes the objective so that policy updates are encouraged when they remain within a trust region around the old policy, but their influence is capped once they exceed a safe deviation. At each timestep, the actions chosen by the agent are sampled on their probability based on the output of the policy distribution.

The agent can thus explore different actions to discover other potentially better strategies, instead of following fixed trajectories. The randomness of the actions is influenced by the agents initial conditions, the type of training performed, and the magnitude of the updates on the policy.

Since with the PPO algorithm the policy gradually becomes more deterministic, it has the risk to converge to local optima, with no more additional positive exploration. To solve this problem, parallel environments and multiple experiences are a good strategy, since they increase the diversity of the explored trajectories, reducing the probability to be trapped in suboptimal regions. Overall, PPO has become one of the leading reinforcement-learning algorithms due to its positive performance, easy implementation and tuning, and its ability to handle complex environments.

3.1.2 BC - Behavioral Cloning Algorithm

Behavioral Cloning (BC) is one of the basic applications of imitation learning, in which a policy is learnt taking expert demonstrations as an example. A dataset $\mathcal{D} = \{(s_i, a_i)\}_{i=1}^{N}$ is collected by the expert executing trajectories into the environment, and then is used by the agent as a supervised learning set of state-action pairs. The goal is to find the policy parameters θ that minimize the difference between the predicted actions and the expert actions. BC typically solves:

$$\max_{\theta} \ \mathbb{E}_{(s,a)\sim\mathcal{D}}\big[\log \pi_\theta(a \mid s)\big] \tag{3.18}$$

or equivalently minimizes the negative log-likelihood (NLL) loss:

$$\mathcal{L}_{\mathrm{BC}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log \pi_\theta(a_i \mid s_i) \tag{3.19}$$

Similarly, BC can be seen as minimizing a surrogate cost function:

$$\min_{\theta} \ \mathbb{E}_{(s,a)\sim\mathcal{D}}[c(s,a)] \tag{3.20}$$

where $c(s,a)$ is a 0-1 cost (i.e. 1 if $\pi_\theta(s) \neq a$, 0 otherwise) or a more smooth loss.

In the case of full-state feedback, where the full environment state s_t is observable, the optimal policy can be represented as a function of the current state:

$$a_t = \pi_\theta(s_t) \tag{3.21}$$

This corresponds to a supervised regression or classification problem, depending on whether the action space is continuous or discrete.

Although BC provides an effective framework for the implementation of imitation learning, its training only based on the expert demonstration leads to a distributional shift, also referred to as the covariate shift problem. When the learned policy π_θ is deployed, it may visit different states from those seen during training, leading to great

performance losses between the training and test distributions. The small errors made by the agent take it to other states than those of the expert, causing a loss in the quality of the agent actions. Ross et al. [12] formalized this issue by showing that if the per-step classification error of the learned policy is ϵ, then the total expected error after a horizon T can grow linearly as $\mathcal{O}(T\epsilon)$ under BC. This is because each error can push the agent into unseen regions of the state space, increasing the probability of further mistakes.
In practical terms, distributional shift occurs as a gradual divergence of the agent behavior from that of the expert, particularly in long-horizon or safety-critical tasks. The agent may behave correctly for a few steps but quickly accumulates deviations that lead to unstable trajectories or failure to complete the task. This limitation formed the basis for the development of interactive imitation learning methods, such as the Dataset Aggregation (DAgger) algorithm [12], which iteratively collects expert corrections on states visited by the learned policy, aligning the training and on-policy state distributions:

$$\mathcal{D}_{t+1} = \mathcal{D}_t \cup \{(s, \pi_E(s)) \mid s \sim p_{\pi_\theta}(s)\} \tag{3.22}$$

By augmenting the dataset with these additional samples, DAgger reduces the covariate shift and ensures more robust policy generalization.

More recently, the sample complexity of BC has been revisited; for example, Foster et al. [13] show that BC's worst-case dependence on horizon can be quadratic in horizon in some settings. Thus, BC is often insufficient per se in tasks requiring long-horizon generalization; many practical implementations combine BC with other techniques (e.g., DAgger [12], GAIL (3.1.3), or RL fine-tuning). In the context of the Unity ML-Agents Toolkit, BC is typically employed as a pre-training stage or as an auxiliary objective in combination with reinforcement learning. This hybrid setup mitigates distributional shift to some extent since reinforcement learning allows the policy to refine its behavior in unseen states beyond the expert demonstrations.

The pseudocode for BC is:

Algorithm 2 Behavioral Cloning

Require: Expert dataset $\mathcal{D} = \{(s_i, a_i)\}$, learning rate α, number of training epochs E, batch size B

Initialize policy parameters θ
for epoch $= 1$ to E **do**
  Shuffle dataset $\mathcal{D}$
  for each mini-batch $\mathcal{B} \subset \mathcal{D}$ of size B **do**
    Compute loss

$$\ell = -\frac{1}{|\mathcal{B}|} \sum_{(s,a)\in\mathcal{B}} \log \pi_\theta(a \mid s)$$

    $\theta \leftarrow \theta - \alpha \nabla_\theta \ell$
  end for
end for
return learned policy π_θ

This simple algorithm treats the imitation problem as supervised learning. BC is

equivalent to training a classifier or regressor that maps states to actions:

$$f_\theta : s \mapsto a.$$

For discrete actions, the network outputs a categorical probability distribution over all possible actions where the target is the expert action. For continuous actions, the output layer represents either the mean or the parameters of a Gaussian distribution, allowing the network to model stochastic policies. In both cases, the optimization seeks to minimize the expected discrepancy between predicted and expert actions.

3.1.3 GAIL - Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning (GAIL) [14] is a model-free imitation learning algorithm inspired by Generative Adversarial Networks (GANs). It directly learns a policy from expert behavior without the need to retrieve a cost function as this is done in Inverse Reinforcement Learning (IRL) [15]. In GAIL, a discriminator network is trained to distinguish expert state-action pairs from those produced by the agent, while the agent policy is simultaneously trained to maximize the reward signal. The objective of the agent policy is to cause the discriminator to detect its actions as those of the expert. The theoretical strength of GAIL lies in its minimization of the Jensen-Shannon divergence between the expert and agent occupancy measures (augmented by an entropy regularization term). This leads to the potential to achieve close imitation in large action-space domains.

Given the expert trajectories $\tau_E = \{(s_t, a_t)\}$, sampled from an expert policy $\pi_E(a|s)$, the objective is to learn a policy $\pi_\theta(a|s)$ such that the occupancy measure $\rho_{\pi_\theta}(s, a)$ approximates the one of the expert:

$$\rho_{\pi_\theta}(s, a) = \pi_\theta(a|s) \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\pi_\theta) \tag{3.23}$$

The occupancy measure is the expected number of times a state-action pair is visited under a specific policy, calculated as a probability distribution. In traditional IRL, we first recover a cost function $c(s, a)$ such that the expert minimizes the expected cumulative cost. GAIL bypasses this step by directly solving:

$$\min_{\pi_\theta} \max_{D_\omega} \mathbb{E}_{\pi_E}[\log D_\omega(s, a)] + \mathbb{E}_{\pi_\theta}[\log(1 - D_\omega(s, a))] - \lambda H(\pi_\theta) \tag{3.24}$$

where

- **Outer Optimization** $\min_{\pi_\theta}$: π_θ is the policy to be learned, parametrized by neural network weights θ.
- **Inner Optimization** $\max_{D_\omega}$: D_ω is the discriminator network parametrized by weights ω. It is a binary classifier that tries to distinguish between the expert behavior ($D_\omega(s, a) \approx 1$) and the learner behavior ($D_\omega(s, a) \approx 0$). It maximizes the objective function with respect to discriminator parameters ω to accurately classify which state-action pairs come from expert and which from the learner.

- **Expert Policy** π_E: The expert's policy that we want to imitate.

- **First Expectation** $\mathbb{E}_{\pi_E}[\log D_\omega(s,a)]$: Expected log-probability of the discriminator identifying correctly the expert state-action pairs. When discriminator sees expert data, the best outcome is $D_\omega(s,a) \approx 1$, since $\log(D_\omega(s,a) = 1) = 0$ is the maximum value.

- **Second Expectation** $\mathbb{E}_{\pi_\theta}[\log(1 - D_\omega(s,a))]$: Expected log-probability of the discriminator identifying the learner state-action pairs as non-expert. When discriminator sees learner data, the best outcome is $D_\omega(s,a) \approx 0$, since $\log(1 - D_\omega(s,a) = 1) = 0$ is the maximum value

- **Entropy Regularization** $-\lambda H(\pi_\theta)$: $H(\pi_\theta)$ is the entropy of the policy π_θ and it measures the randomness in the policy's action selection. It is calculated as

$$H(\pi_\theta) = \mathbb{E}_{\pi_\theta}[-\log \pi_\theta(a|s)]. \tag{3.25}$$

  High entropy indicates that the policy is more stochastic (explores more), while a low entropy indicates that the policy is more deterministic (exploits more).

- **Regularization Term** λ: $\lambda > 0$ is the entropy coefficient or temperature parameter. During the minimization, subtracting entropy encourages higher entropy and thus more exploration. Large λ indicates strong exploration and the policy stays stochastic longer, while small λ indicates less exploration, with the policy becoming deterministic faster. $\lambda = 0$ indicates no entropy regularization and pure imitation.

The 3.24 formulation directly minimizes the Jensen-Shannon divergence between expert and learner occupancy measures. The discriminator acts as a learned reward function:

$$r(s,a) = -\log(1 - D_\omega(s,a)) \tag{3.26}$$

which encourages the policy to visit states and actions similar to those of the expert.

The GAIL training loop alternates between updating the discriminator and the agent policy. The discriminator is trained using samples from both the expert and the policy, while the policy is updated using any reinforcement learning algorithm (e.g., PPO or SAC) with the discriminator-derived rewards.

Algorithm 3 Generative Adversarial Imitation Learning (GAIL)

1: Initialize policy $\pi_\theta(a|s)$ and discriminator $D_\omega(s,a)$
2: **for** each iteration **do**
3: Sample trajectories τ_i using π_θ
4: Update discriminator by ascending its gradient:

$$\nabla_\omega \Big(\mathbb{E}_{\pi_E}[\log D_\omega(s,a)] + \mathbb{E}_{\pi_\theta}[\log(1 - D_\omega(s,a))] \Big)$$

5: Compute pseudo-rewards for policy updates:

$$r(s,a) = -\log(1 - D_\omega(s,a))$$

6: Update policy π_θ using RL on $r(s,a)$, maximizing:

$$\mathbb{E}_{\pi_\theta} \left[\sum_t \gamma^t (r(s_t,a_t) - \lambda \log \pi_\theta(a_t|s_t)) \right]$$

7: **end for**

3.2 ROS2 System and the Jackal

In this project, the simulated robot is the *Jackal* [16], a four-wheeled ground vehicle developed by Clearpath Robotics. The Jackal is equipped with a LIDAR sensor that provides 360-degree environmental scanning for navigation and obstacle avoidance. In the simulated environment, this sensor data is emulated within Unity and transmitted to ROS2 through the TCP/IP interface, ensuring a realistic representation of perception and movement within the virtual scene.

The Robot Operating System 2 (ROS2) is an open-source robotics middleware framework designed for the development of robot applications [17]. Unlike its predecessor ROS, ROS2 offers improvements in real-time capabilities, security, and cross-platform support. ROS2 allows the creation of robotic nodes that use a publish/subscribe communication paradigm. In this project, ROS2 is one of the main components of the training environment: the simulated robot nodes subscribe to control commands received by an agent perceiving the virtual scene, enabling the integration of Unity simulation with actual robotic systems.

Figure 3.3: Jackal robot model

The Jackal UGV is a compact, rugged mobile robot platform intended for research and development. According to Clearpath Robotics [16], the Jackal features an onboard PC, GPS/IMU integration, and out-of-the-box ROS compatibility. The external dimensions are approximately 508 mm × 430 mm × 250 mm, with a weight of about 17 kg, and maximum payload of about 20 kg. The maximum speed is approximately 2.0 m/s in typical usage scenarios. The Jackal operates as a differential drive (or skid steered) vehicle which simplifies integration of velocity commands (linear and angular) in ROS2 topic interfaces. From an application perspective, the Jackal UGV has been used in autonomous exploration, multi-objective planning, terrain navigation, and human-aware robotics. For example, a team at Massachusetts Institute of Technology (MIT) outfitted a Jackal to navigate among crowds of pedestrians and perform socially-aware motion [18].

The robot model is imported in the simulation using a URDF file. The *Unified Robot Description Format* (URDF) is an XML-based format used by ROS to describe the physical and kinematic structure of a robot. A URDF file defines geometry, joints, sensors, and other components of a robot, enabling the visualization and simulation of its behavior in a virtual environment. By importing a URDF into Unity, a copy of the physical robot can be created.

3.3 ML-Agents

ML-Agents (Machine Learning Agents) [19] is a toolkit developed by Unity Technologies that allows developers to integrate machine learning into Unity environments. To cite directly the ML-Agents package documentation: "ML-Agents enables games and simulations to serve as environments for training intelligent agents in Unity" [19]. It allows using various machine learning frameworks, along with the training of behaviors through reinforcement learning, imitation learning, or other AI techniques. By connecting the simulated scene to a learning algorithm, ML-Agents allows virtual entities referred to as

Agents to learn different behaviors, applying suitable actions to achieve the best outcome. The ML-Agents Toolkit has five high-level components:

- **The Learning Environment**, which contains the training scene and all simulation elements. The configuration of the scene should be built on the type of behavior the agent needs to learn. The ML-Agents Toolkit includes an ML-Agents Unity SDK (com.unity.ml-agents package [20]) that allows the user to transform any Unity scene into a learning environment. By virtue of this SDK, agents and behaviors can be defined and customized, and attached to the game objects of the scene.

- **The Python Low-Level API**, which contains a low-level Python interface that makes interaction and manipulation of the learning environment possible. This element works outside Unity and communicates with the project through the Communicator. It is contained in a dedicated Python package referred to as *mlagents_env*, and it is used by the Python training process to control the Academy during the training (further details on the Academy will be given below).

- **The External Communicator**, which connects the Learning Environment and the Python API. It forms part of the Learning Environment and is created in Unity.

- **The Python trainer**, which contains the machine learning algorithms that can be implemented to carry out the training of the agents. ML-Agents package has a single command utility *mlagents-learn*, that supports all the training methods supported by the package. The trainer communicates with the Python API only and is outside Unity.

- **The Gym Wrapper and the PettingZoo Wrapper**, which allow the users to work with their python Training algorithms within the ML-Agents package. These elements will not be used in this thesis since all the additional code will be directly defined in *C#* and inside the Unity project.

Multiple components are defined within the learning environment. The first one is the Academy, a *C#* class part of the ML-Agents package that serves as a controller for the learning components inside the Unity simulation. It initializes the simulation environment, controls the training steps, manages the observations, and tracks the episode completion, rewards, and environment resets. The Academy calls the methods *OnEpisodeBegin()*, *CollectObservations()*, *OnActionReceived()* and *OnEpisodeBegin()* for each Agent in the scene. In this thesis, since a single agent is present, the methods to reset the environment are defined inside the Agent script without modifying the Academy class.

In the ML-Agents framework, an Agent is the main learning entity. Once the script defining the Agent is attached to a GameObject, this element interacts with the environment getting information, taking actions, and receiving feedback in the form of rewards. The combination of this data sets the way for the creation of the Agent behavior. The Agent can apply different learning algorithms that can be specified by the user, together with the various parameters of the specific algorithm. It is based on three major entities:

- **Observations:** The observations are the data received from the agent about the environment. They can be numeric or visual, measuring what the Agent can perceive about the surrounding scene. The observations are a subset of the so-called environment state, which contains all information on all object present in the scene. Observations can be associated with the data obtainable by a physical robot with the use of sensors, such as a camera or laser.

- **Actions:** The actions that the Agent can take. They can be discrete or continuous, in accordance with the type of environment and complexity of the task. They can be mapped to be direct forces applied in the Unity environment, or commands to be sent to the robot to move or act.

- **Rewards:** A reward is a scalar value received by the agent that can indicate the quality of the action performed or the general quality of the behavior exhibited. The reward function needs to be set so that the maximization of the total reward in the episode generates the desired behavior.

Once these entities are defined, the Agent can be trained through several trials within the environment, where it will learn a behavior (policy) that maps observations to actions. Each trial corresponds to an episode that ends when a win or loss condition occurs.
In the code, an agent is defined by creating a class that inherits from the Agent base class provided by the ML-Agents API. In this class there are key methods that define the agent learning loop:

- The `OnEpisodeBegin()` method is called at the start of each training episode and is used to reset the environment and agent state, ensuring that each episode begins under controlled or randomized conditions.

- The `CollectObservations()` method is responsible for collecting information about the environment. These observations are then passed to the neural network to guide the decision making process.

- The `OnActionReceived()` method defines what happens when the agent performs an action. It interprets the output from the policy model and applies it to the environment, for example by causing robot movement, rotating a robot joint, or adjusting a parameter.

- The `Heuristic()` method is used for the definition of the heuristic actions. It allows the user to manually define actions for the agent by mapping keyboard or joystick inputs to action values. When the Agent's behavior is set to *Default*, in the presence of a model its actions are used. If the model is absent, the control switches to the heuristic commands. When the behavior of the agent is set to *Heuristic only*, the actions from the NN-model are ignored and only the actions from the user are received by the agent. This feature is the base for the recording of demonstrations since it allows the user to utilize the Agent action directly to show the correct behavior in the various states.

The other major component for the ML-Agents package is the Behavior. It defines the characteristics of the Agent, such as the type and number of actions it can take. Each Behavior is identified by a *Behavior Name* field. The Behavior receives the observations and rewards of the Agent and returns actions. The Behavior can be

- Learning: The Behavior is still to be trained and is not defined yet.
- Heuristic: The Behavior is defined by deterministic actions defined in the code.
- Inference: The Behavior includes a NN model and its actions are defined by the latter. An Inference Behavior is a trained Learning Behavior.

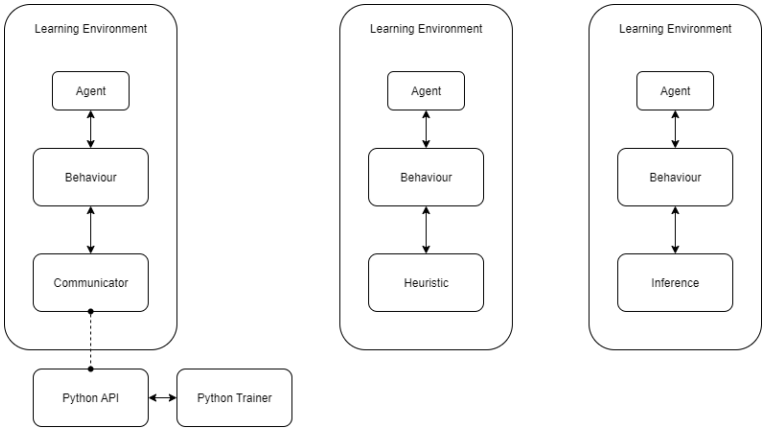


Figure 3.4: Learning, Heuristic and Inference Behavior structure

Rewards play a central role for the agent learning process. After an action or at the resolution of a certain event, the agent receives a value through reward signals that indicate whether its behavior was correct or not. Positive rewards encourage repeating similar actions, while negative rewards discourage them. The reward value is typically defined by the environment, where reaching the goal corresponds to a positive instance, while crashing or failing corresponds to a negative one. Rewards can also be defined outside of the environment to encourage certain behaviors or help learning complex tasks. While the rewards inside the environment are referred to as extrinsic, those defined externally are referred to as intrinsic. GAIL algorithm (3.1.3) is used as a intrinsic reward mechanism, introducing a discriminator $D_\phi(s, a)$. The agent receives a reward therefrom, which is proportional to the extent to which the agent actions match those of the expert. On the other hand, BC (3.1.2) is not an intrinsic reward but an auxiliary supervised objective and adds a loss term to the policy optimization.
The four reward signals provided by Ml-Agents are

- *extrinsic*: enabled by default, they represent the rewards received into the environment and are specified by the developer.
- *gail*: the reward signal received by the GAIL discriminator.

- *curiosity*: together with *rnd*, it is defined inside the Curiosity module. It is an intrinsic reward signal that encourages exploration in sparse-reward environments.
- *rnd*: Random Network Distillation (RND) [21] is defined inside the Curiosity module and is an intrinsic reward signal that encourages exploration in sparse-reward environments.

For the definition of the reward function for this project, various rewards have been taken into account:

- **Social distance:** a reward has been designed for keeping distance from people in the scene. The reward is calculated as

$$R_p = -\frac{(\text{minSafeDistance} - \text{minPersonDist})}{\text{minPersonDist}} \tag{3.27}$$

where *minSafeDistance* is a parameter indicating the safety distance from people and objects, while *minPersonDist* is the minimum current distance from people measured by the robot. R_p is then clamped to the range

$$R_p = \begin{cases} -2, & \text{if } R_p < -2, \\ R_p, & \text{if } -2 \leq R_p \leq 2, \\ 2, & \text{if } R_p > 2. \end{cases} \tag{3.28}$$

The reward for people distance has weight c_p, that is set to 1 to highlight the importance of people distance and social behavior.

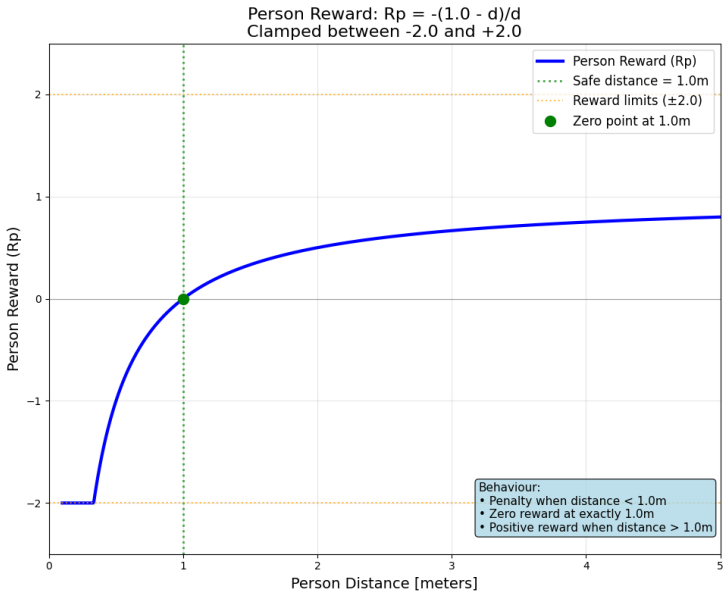


Figure 3.5: Social Distance Reward with $minSafeDistance = 1$

- **Obstacle distance:** the same concept for keeping distance from people has been applied to obstacles. The reward is calculated as

$$R_o = -\frac{(\text{minSafeDistance} - \text{minObstacleDist})}{\text{minObstacleDist}} \tag{3.29}$$

where *minSafeDistance* is a parameter indicating the safety distance from people and objects, while *minObstacleDist* is the minimum current distance from objects measured by the robot. R_o is then clamped to the range

$$R_o = \begin{cases} -2, & \text{if } R_o < -2, \\ R_o, & \text{if } -2 \leq R_o \leq 2, \\ 2, & \text{if } R_o > 2. \end{cases} \tag{3.30}$$

The reward for obstacle distance has weight c_o that has been kept below one during testing, usually set to a value of 0.8.

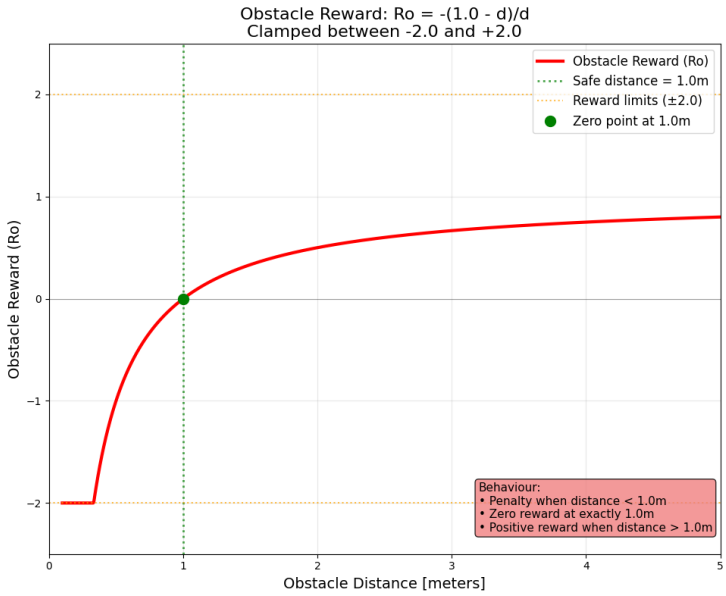


Figure 3.6: Obstacle Distance Reward with $minSafeDistance = 1$

- **Wall distance:** since a good social behavior for robots is to move along walls and not in the middle of the room, a reward for keeping a certain distance from the wall has been designed. It is calculated as

$$d_e = |\text{minWallDist} - \text{optimalWallDistance}| \tag{3.31}$$

$$G = e^{-4d_e^2} \tag{3.32}$$

The reward R_w is then defined as

$$R_w = \begin{cases} -(1-G), & \text{if minWallDist} < \text{optimalWallDistance}, \\ G, & \text{otherwise.} \end{cases} \tag{3.33}$$

Finally, the reward is clamped to the range

$$R_w = \begin{cases} -1, & \text{if } R_w < -1, \\ R_w, & \text{if } -1 \leq R_w \leq 1, \\ 1, & \text{if } R_w > 1. \end{cases} \tag{3.34}$$

The reward for wall distancing has weight c_w less than 1 since the target could be anywhere in a room and not always close to a wall.

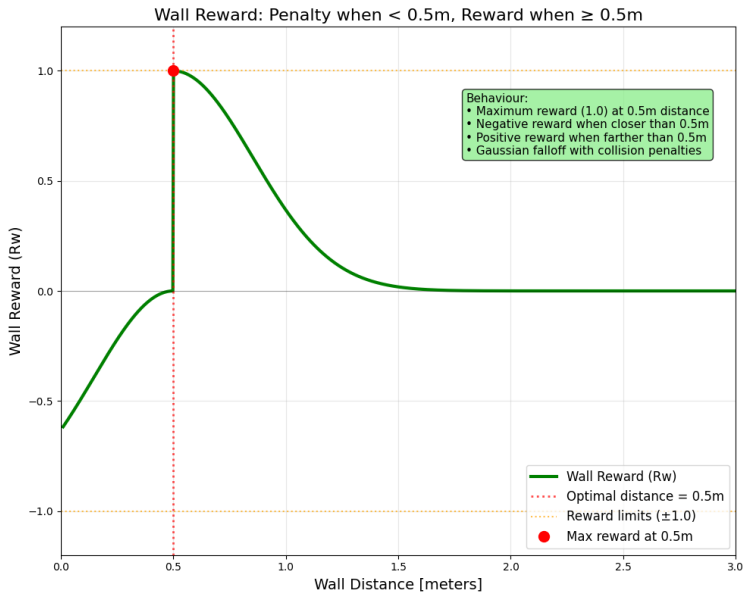


Figure 3.7: Wall Distance Reward with $optimalWallDistance = 0.5$

- **Velocity reward:** a reward for keeping a target linear velocity. This reward can help the robot to complete the task more quickly. It is calculated as

$$R_{\text{speed}} = 1.0 - \text{Clamp}_{[0,1]}\left(\frac{|v_{\text{current}} - v_{\text{target}}|}{v_{\text{target}}}\right) \tag{3.35}$$

where v_{current} is the current speed of the robot, v_{target} is the desired target speed, and $\text{Clamp}_{[0,1]}(x)$ is a clamping function that restricts the value of x to the range [0,1]. The weight of the velocity reward is c_v, usually less than 1 since the robot may need to stop to avoid collisions with people passing nearby.

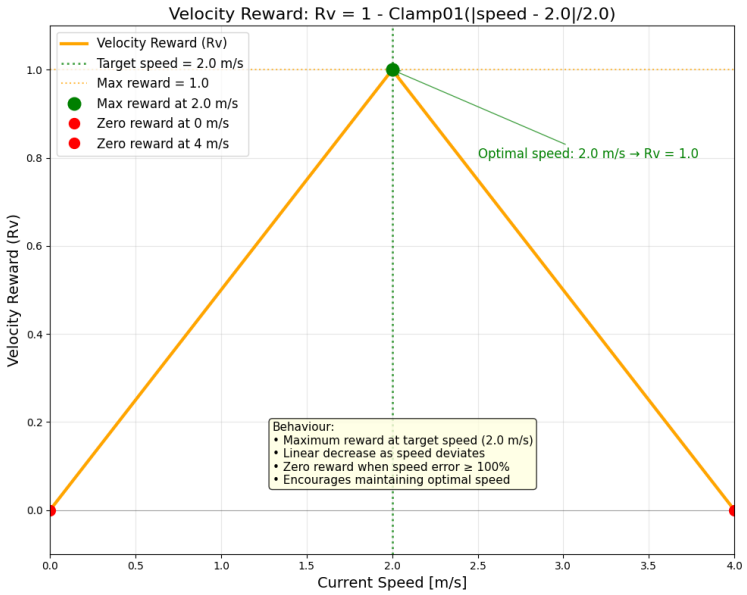


Figure 3.8: Velocity Reward with $targetSpeed = 2.0$

- **Heading reward:** a reward for heading in the right direction towards the target. It is calculated as

$$\theta = \text{Angle}(\text{robotForward}, \text{toTarget}) \cdot \frac{\pi}{180} \tag{3.36}$$

where θ is the angular difference (in radians) between the forward direction of the robot and the target direction. The reward is then calculated as

$$R_h = 1 - 2\sqrt{\left|\frac{\theta}{\pi}\right|} \tag{3.37}$$

so that the agent receives the maximum reward ($R_h = 1$) when perfectly aligned with the target ($\theta = 0$), and a minimum reward ($R_h = -1$) when facing directly away from it ($\theta = \pi$). The heading reward has weight c_h, that usually is lower than 1 since the robot could need to turn to avoid obstacles for reaching its target.

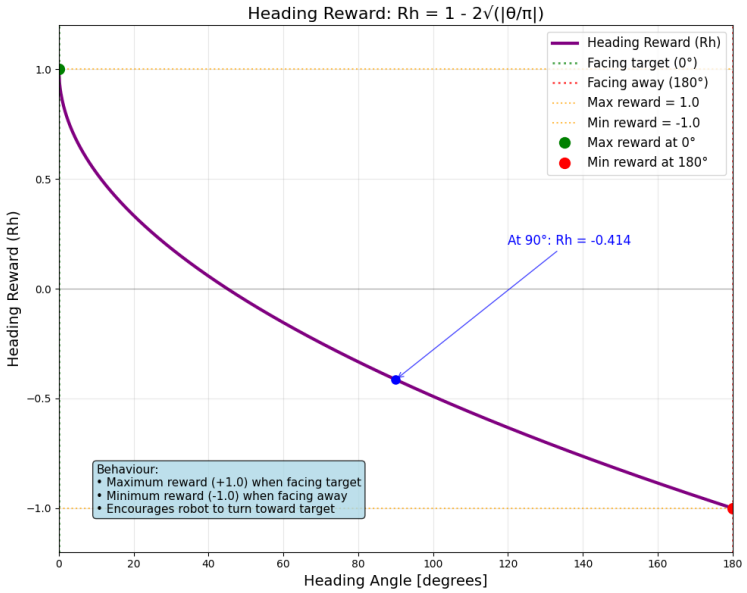


Figure 3.9: Heading Reward

- **Distance reward:** The distance to the target is one of the most important rewards to be designed. Two main configurations have been designed. In the first one, the distance reward is calculated as

$$\Delta d = d_{\text{prev}} - d_{\text{target}} \tag{3.38}$$

$$R_d = 20 \frac{\Delta d}{inital Distance} \tag{3.39}$$

Since the movement for each action is very small, the 20 factor makes it more consistent. The value is then normalized with respect to the starting distance of the episode. The final reward is then clamped to the range $[-1, 1]$ as follows:

$$R_d = \begin{cases} -1, & \text{if } R'_d < -1, \\ R'_d, & \text{if } -1 \leq R'_d \leq 1, \\ 1, & \text{if } R'_d > 1. \end{cases} \tag{3.40}$$

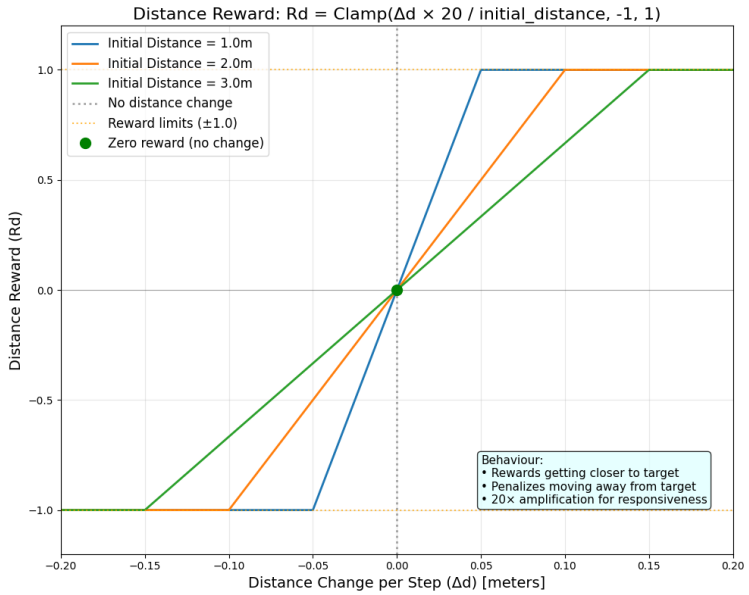


Figure 3.10: Examples of first type of Distance Reward with different initial distances

The second form for the distance reward is normalized with respect to the maximum distance from the target between the starting point and the end point of the movement. It is calculated as:

$$R_d = \frac{d_{prev} - d_{current}}{Max(d_{prev}, d_{current})} \tag{3.41}$$

The distance reward has weight c_d. Tuning this parameter poses a significant challenge, since higher values encourage faster task completion, whereas lower ones may prompt the robot to take alternative routes rather than the straight one to the target, which are necessary to avoid obstacles.

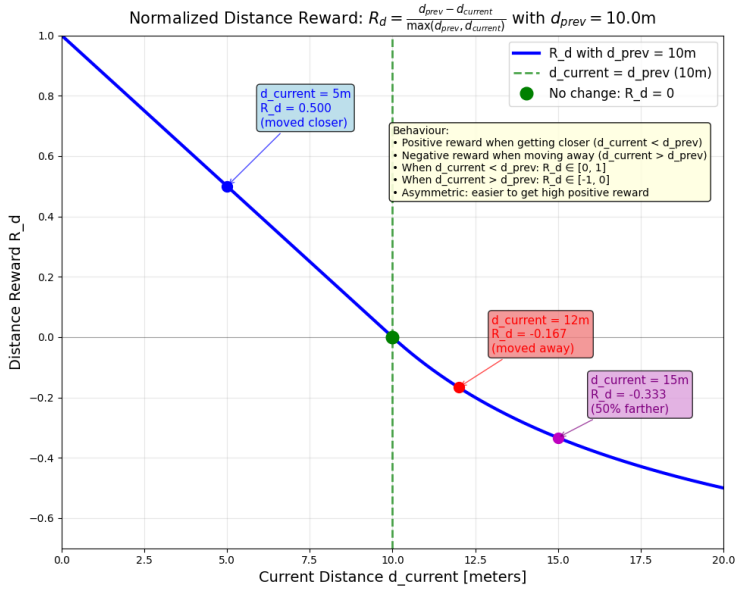


Figure 3.11: Second type of Distance Reward with previous distance equal to 10

- **Proximity reward:** this reward is assigned to the robot when it enters a certain range from the target. In curriculum training, the required distance from the goal for earning this reward gradually decreases as the robot performs the task successfully. This reward has a fixed value, usually

$$Rprox = 1 \tag{3.42}$$

that can be tuned using its weight c_{prox}.

- **Time penalty:** a time penalty has been set to cause the robot to complete its task as quickly as possible. It is calculated as

$$R_t = -\frac{t_{\text{elapsed}}}{t_{\max}} R_{\max} \tag{3.43}$$

where t_{elapsed} is the current episode time, $t_{\max}$ is the episode timeout, and $R_{\max}$ represents the maximum penalty value (set to 2). The time penalty has weight c_{time}, which approaches 1 as the robot learns how to perform its task.

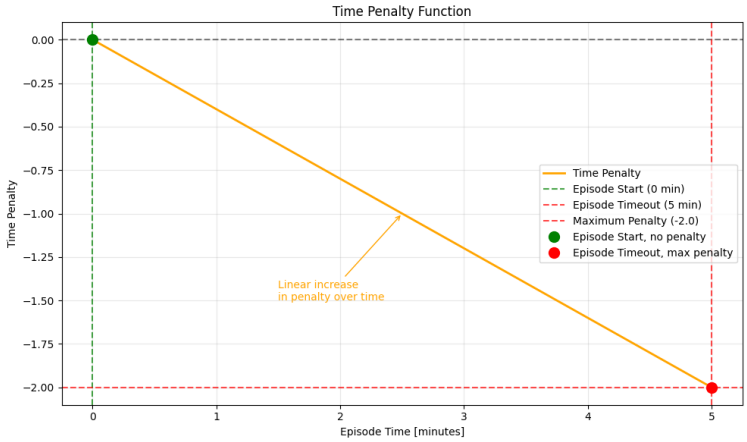


Figure 3.12: Wall Distance Reward with $timeout = 5\,minutes$

- **Event-based rewards and penalties:** if the robot collides with an object or reaches the target, a fixed value reflecting the event is added to the reward calculation.
  - – If the robot has an unobstructed line of sight to the target (`blocked = false`), a small positive reward is added (e.g., $reward = 1$)
  - When the robot reaches the target, a large reward is added (e.g., $reward = 20$)
  - Collisions with obstacles or walls cause a small penalty (e.g., $reward = -5$)
  - Collisions with people result in a bigger penalty, to highlight the importance of social correct behavior (e.g., $reward = -10$)
  - If the episode ends because of a timeout, a small penalty is applied (e.g., $reward = -5$)

When the learning objective for the current episode is achieved or failed, the function `EndEpisode()` is called. This method terminates the current episode and starts a new one, allowing the agent to learn through multiple trials.

3.3.1 ML-Agents and Algorithm Configuration

Within the context of the Unity ML-Agents Toolkit, both reinforcement learning (Proximal Policy Optimization PPO, 3.1.1) and imitation learning (Behavioral Cloning BC 3.1.2 and Generative Adversarial Imitation Learning GAIL 3.1.3) methods are supported. These algorithms can be combined to train agents efficiently in simulation environments. In order to start and customize the training, the creation of a configuration file *.yaml* is required. Multiple parameters can be defined, along with their variation during training to tune the exploration level and the learning speed of the Agent, for example. ML-agents integrates TensorBoard, a visualization tool for monitoring and analyzing machine learning experiments. TensorBoard provides insights into the training process, allowing users

to track metrics such as rewards, losses, and other performance indicators. By visualizing these metrics, developers can better understand how their agents are learning and make decisions about hyperparameter tuning and model adjustments. The full list of parameters is available in the Ml-Agents Documentation [19]. In particular, the parameters used in this thesis are:

- *trainer_type*: It allows specifying the type of trainer to be used. It can be *ppo*, *sac* or *poca*, with default value *ppo*.

- *time_horizon*: It indicates the number of experiences that the agent needs to collect before adding its experience to the buffer. When the horizon limit is reached before the end of an episode, the overall expected reward is predicted using value estimation. This indicates that a long-time horizon causes a higher variance but a less biased estimate, while a smaller horizon causes a less varied but more biased one. The choice of the time horizon needs to balance this trade-off, considering that bigger values can sequentially capture the important behavior of the agent actions, while a smaller number can help with frequent rewards and longer episodes. The default value is 64, but it typically ranges from 32 to 2048.

- *summary_freq*: It indicates how many steps must be collected before adding the performance to the graphs in TensorBoard.

- *max_steps:* It specifies the maximum number of steps that the agent takes within the environment during training. Once this limit is reached, the training is stopped and the model will be created.

- *keep_checkpoints*: It indicates the maximum number of checkpoints to be held during training. By virtue of the creation of checkpoints, if the training crashes or performance drops, the training can be resumed from the last checkpoint instead of starting from the beginning. If *even_checkpoints* is false, a checkpoint is saved every *checkpoint_interval* steps. Once the maximum number of checkpoints is reached, the oldest checkpoint is deleted to make space for the new one.

- *even_checkpoints*: It specifies whether checkpoints need to be saved every *checkpoint_interval* steps (when set to false) or whether they must be evenly distributed during training (when set to true). By default, such a parameter is set to false.

- *checkpoint_interval*: It indicates the number of steps before a checkpoint is saved.

- *hyperparameters: batch_size*: It specifies the number of experiences required in each iteration of gradient descent optimization. It should be multiple times smaller than the buffer size in order to collect experience before updating the policy model.

- *hyperparameters: buffer_size*: It indicates the number of experiences collected before the policy model is updated. In order to have a more stable training, updates are usually obtained using bigger buffer sizes.

- *hyperparameters: learning_rate*: It specifies the initial learning rate for gradient descent. It is usually denoted as α before the policy model is updated. In order to have a more stable training, updates are usually obtained using bigger buffer sizes. Its value ranges between $1e-5$ and $1e-3$, with default value as $3e-4$.

- *hyperparameters: learning_rate_schedule*: It determines how the learning rate changes during training. It can be constant or linear, in which case it decays over time.

- *hyperparameters: hidden_units*: It indicates the number of hidden layers in the neural network. It should be increased as the complexity of the observations increases. It typically ranges from 32 to 512.

- *hyperparameters: num_layers*: It specifies the number of hidden layers in the neural network. It increases in the case of complex models. It usually ranges from 1 to 3.

- *hyperparameters: normalize*: It indicates if normalization is applied to the observation vector. Normalization can be helpful in the case of complex and high variance observations to avoid values that are large in magnitude from dominating the learning signal. The normalization is applied as

$$x_{norm} = \frac{x - \mu}{\sigma}$$

  where μ is the running mean of the observation (x) dimension, and σ is the running variance.

When a certain trainer is chosen, additional configuration parameters can be added. The PPO-specific ones are:

- *hyperparameters: beta*: It specifies the strength of the entropy regularization term to encourage exploration while penalizing overly deterministic policies. Increasing β increases randomness in the policy, while decreasing it causes greater exploitation. This value needs to be tuned so that entropy decreases slowly while the reward increases. It ranges from $1e-4$ to $1e-2$.

- *hyperparameters: beta_schedule*: It determines how the value of beta decays during training. It can be constant or linear.

- *hyperparameters: epsilon*: It controls the clipping range in the surrogate loss function limiting the extent to which the new-policy probability ratio $r_t(\theta)$ can deviate from 1. Smaller ϵ values maintain conservative updates for stability, while larger values allow for a faster policy evolution but risk instability. Its value typically ranges from 0.1 to 0.3.

- *hyperparameters: epsilon_schedule*: It determines how the value of epsilon decays during training. It can be constant or linear.

- *hyperparameters: lambd*: Used in computing the Generalized Advantage Estimation (GAE) [22], it balances bias and variance in the advantage estimator:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}, \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

  When $\lambda \to 1$, the estimator heavily relies on long-term rewards (low bias, high variance); when $\lambda \to 0$, it depends mostly on immediate value predictions (high bias, low variance). Typical values are between 0.9 and 0.95.

- *hyperparameters: num_epoch*: It indicates the number of times needed to go through the experience buffer before performing gradient descent optimization.

Tuning such parameters is fundamental for enhancing the performance and stability of training using reinforcement learning.

For the integration of imitation learning, Unity ML-Agents provides a game object component referred to as *Demonstration recorder*, enabling developers to record expert demonstrations within a Unity scene. The demonstrations consist of state-action trajectories stored in `.demo` files, which serve as training data for the agent. It can be recorded by playing the scene in the Unity editor, and giving the robot commands using a controller or the computer keyboard. Imitation learning can be configured via the *.yaml* trainer configuration file, where reward signals and behavioral cloning settings are specified. As highlighted above in chapter 3.3, imitation learning is implemented in different ways for GAIL (3.1.3) and BC (3.1.2). GAIL algorithm is used as an intrinsic reward mechanism, while BC is an auxiliary supervised objective.

Since the very first versions of the package, ML-Agents has been removing offline BC training, forcing the imitation learning to be integrated with RL algorithms such as PPO or SAC. However, in the configuration files, it is possible to set the strength of the influence of BC to the maximum 1 to see its effect on the training. BC remains effective when demonstration data covers most of the possible state distributions; otherwise, distributional shift may still occur if the policy encounters unseen states during training or deployment [23]. In the Unity ML-Agents framework, BC operates under the full-state feedback assumption that the agent receives complete numerical observations such as positions, velocities, and target coordinates that represent the full environment state. The policy network learns to map these observations directly to actions by minimizing the BC loss on the demonstration dataset. In practice, BC is implemented as an auxiliary objective combined with reinforcement learning updates:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{RL}} + \lambda_{\text{BC}} \, \mathcal{L}_{\text{BC}},$$

where λ_{BC} is the parameter that controls the strength of the imitation objective. In order to implement Behavioral Cloning, the addition of a *behavioral_cloning* section to the configuration *.yaml* file is required. The parameters used to tune the BC implementation are:

- *demo_file*: It specifies the path to the *.demo* file where the demonstration is stored.

- *strength*: It correspond to λ_{BC} and indicates how much the BC learning influences the policy.
- *steps*: It defines the amount of steps on which the BC influence is active during training. It can be used to create hybrid trainings, with first experience and rewards collected under the BC objective, then generalized under reinforcement learning.
- *samples_per_update*: It defines the amount of steps on which the BC influence is active during training. It can be used to create hybrid trainings, with first experience and rewards collected under the BC objective, then generalized under reinforcement learning.

GAIL extrinsic reward signal can also be tuned using parameters in the *.yaml* configuration file. Its influence in conjunction with the strength of the extrinsic reward can be tuned, together with the network specification of the GAIL discriminator. The parameters for the *gail* section are:

- *demo_path*: It specifies the path to the *.demo* file where the demonstration is stored.
- *strength*: It is a factor that multiplies the raw reward. It varies from 0.1 to 1 and should be kept smaller in the case of human demonstration to leave space to reinforcement learning improvements and optimization.
- *gamma*: It is the discount factor for future rewards. It indicates how far in the future the Agent should think about possible rewards. It must be strictly smaller than 1, with a default value of 0.99.
- *learning_rate*: It indicates the learning rate for the GAIL discriminator. If the training is unstable, it should be decreased. The default value is $3e-4$ and it ranges from $1e-5$ to $1e-3$
- *network_settings*: This section for GAIL contains the same parameters as that for PPO. The parameters of this section have been set to match those of PPO to allow for performance comparison under the same settings.

3.4 Reward Engineering Challenges

On of the main challenges in reinforcement learning systems is the definition of a suitable reward function that optimizes the learning process of the Agent. Although the design of the rewards seems an easy task since it should reflect the desired behavior, it hides several issues. This process, also referred to as reward engineering, has recently attracted more attention due to the increasing importance of reinforcement learning. The first challenge is the actual transformation of the desired behavior into a reward function. Reinforcement learning rewards can vary in structure and distribution, from being constantly fed to the Agent every step to being event-driven. The same behavior can be coded in different ways, from a single reward for the completion of the task to complex rewards consisting of sums of different conditions experienced by the Agents.

The navigation task that can be translated as *reach the target without collisions with other entities*, for example, can be designed in numerous different ways. An example of an easy implementation and an example of a complex one are given hereinbelow:

- The reward function can be null in the whole environment, with only positive value when the robot reaches the target position. When the robot collides with other objects, reset the episode.
- The reward function adds a value to the reward at each step, which is the sum of several components:
  - a value proportional to the proximity of walls,
  - a value proportional to the progress of the robot from the previous step,
  - a value proportional to the velocity of the robot that is compared with a target velocity,
  - a value proportional to the robot heading error relative to the robot-to-target direction,
  - etc.

More complex reward functions can speed up the training processes, guiding intermediate behavior instead of blind objective seeking. However, this type of function shaping can increase the risk of introducing unintended incentives that can cause the robot to have different behaviors from that designed. The balance between complexity and training speed becomes another challenge in reward function engineering.

The magnitude of the rewards assigned for the various events within the environment gives rise to a further problem. For example, in navigation tasks an agent may receive a reward for having reduced the distance to a goal and a separate reward for having avoided obstacles. If the penalty assigned for the obstacle avoidance is too heavily weighted, the agent may minimize collisions but never attempt to reach the goal. In the results according to Ibrahim et al. [24] it is emphasized how actual intentional reward components can dominate optimization and lead to degeneration of behaviors. Since the agent optimizes what it is rewarded for and not necessarily what is intended, every component of the reward must be checked for unplanned influence.

Another issue concerns the non-stationarity of the environment, even more when a social scene in simulated. As an agent improves, the states it visits shift, along with the feedback from the reward signal. If the reward design does not account for this, then the optimization may converge prematurely or settle into a local optimum. In particular, algorithms such as PPO greatly suffer from the settlement to local optima, with small room for improvement. Once the agent has discovered a behavior that reliably brings moderate reward and avoids major penalties, it may cease exploring and refine this strategy. Avoiding poor exploration and acceptable partial policy is part of a good reward design.

Since the same task can be carried out in different environments with different complexities, it is possible that the reward function needs to evolve with the environment. For example, the navigation of an empty room involves other needs in the robot behavior

than those needed by a room with people or a room with fixed obstacles, etc. Whereas a reward for traveling straight to the target in an empty room may result in optimal training, avoiding possible intermediate walls or objects can fail the same reward function in a room with obstacles. Each configuration needs a properly tailored reward function, without modifying the environment characteristics.

Reward functions can also become overly specialized to the environment in which the Agent operates. Most of the time, it is necessary to train a model in different environments to ensure that the learning process is not biased towards a specific environment but adapted to any setting. The generalization and adaptability of a model adds even more difficulty in connection with the reward engineering problem.

In the context of robot navigation, the above reward engineering challenges result in concrete risks:

- The robot may learn that remaining stationary is safer than reaching the goal if the penalties for collisions are too high.
- Sparse goal-based rewards may cause highly slow learning or lack of progress.
- Auxiliary rewards (e.g., maintaining a heading) may inadvertently dominate the goal reward, causing strategy drift, such as a stationary behavior while oriented towards the target.
- High time penalties to encourage fast completion may overcome collision penalties and cause early collisions.
- Rewards for keeping distance from obstacles may cause dangerous navigation in social environments.
- Wrong reward function designs can create repeated actions, for example exiting a proximity reward zone and entering it in a loop fashion to accumulate rewards.

Therefore, when designing a reward for the robot to navigate into a room with people and walls, rewards for reached targets, penalties for obstacles, progress shaping, and exploration incentives must be carefully balanced. It is often preferable to start with simpler scenarios such as an open space without walls, and gradually increase complexity, monitoring the mean reward, standard deviation, episode length, and actual behaviors.

A good solution to the reward engineering challenges is the use of imitation learning as intrinsic reward structure (as GAIL is used in Ml-Agents package), where the reward function is implicitly defined by expert demonstrations, reducing the need for explicit reward design. This compensates for sparse rewards and complex behaviors, as the agent learns and is rewarded based on expert behavior. Its performance is analyzed in chapter 5, along with that of Behavioral Cloning, confronting the two imitation learning methods with pure reinforcement learning.

After several trials with different reward functions, the one chosen for the tests was an event-based reward system, where the agent receives a positive reward of $+20$ for reaching the target, a negative reward of -10 for colliding with people, a negative reward

of -5 for colliding with obstacles or causing a timeout, and a small negative reward of $-0.02/timeout$ for each time step taken to encourage faster navigation. This reward structure aims to balance the need for efficient navigation with the importance of avoiding collisions. Other structures were tested (shown in the Appendix 7, along with their results), but the reward function described above was found to be the most effective.

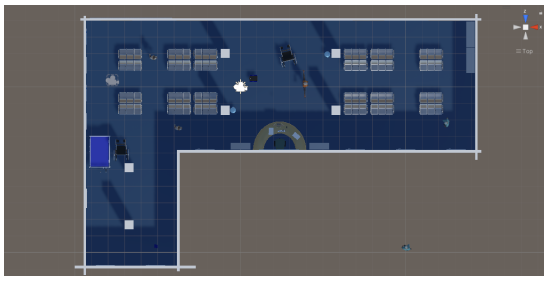
Chapter 4

Environment setup

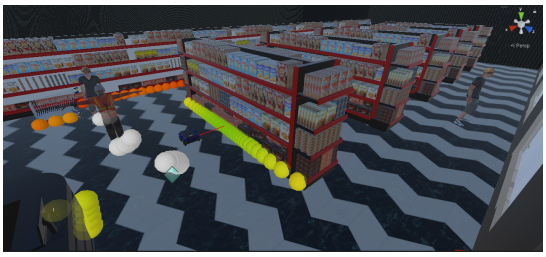
4.1 Unity Game Engine

The Unity Real-Time Development Platform [25] is a cross-platform game engine that enables the creation of 2D, 3D, virtual, and augmented reality environments in real time. Originally developed by Unity Technologies in 2005, it has evolved into one of the most widely adopted real-time 3D (RT3D) engines. Although primarily known as a game engine, Unity has increasingly been used in fields such as research, robotics, simulation, and application development. It features characteristics that make it suitable for intuitive user interaction, including an integrated editor and a component-based architecture, which make the environment highly customizable and flexible.

Unity provides the foundation for the training environment developed in this thesis. Once a project is created, the virtual environment can be customized according to the specific requirements of the training scenario. This customization can be achieved either by adding 3D models designed for the particular context or by using prefabs - reusable virtual components that can be instantiated across multiple scenes. Unity also offers a set of basic 2D and 3D primitive objects (such as cubes, spheres, and planes) that can be directly created within any scene, while more complex assets — including prefabs, textures, and models obtained from external sources — must first be imported into the project directory structure. Within the Assets folder, Unity organizes files into subdirectories according to their function: for example, *Prefabs* stores reusable object templates, *Scripts* contains the C# code controlling object behavior, and *Scenes* hold the different environments that form the project. Once the virtual environment is properly configured, it can be saved as a scene. Each scene represents a self-contained environment within the project and can later be reused, modified, or combined with others, allowing for the creation of multiple training scenarios within the same Unity project.



(a) Example of a hospital environment



(b) Example of a supermarket environment

Figure 4.1: Examples of virtual environments created with Unity

A simple scene has been created for our training environment. The walls in the scene consist of cubes with dimensions that have been properly modified. A tag *Wall* has been added to these objects. The floor is a plane referred to as *Floor* in the hierarchy. Both these tags and names will be important since they are used by the code controlling the Agent. Since the task is complex, multiple scenes have been designed to make the training gradual with increasing difficulty. The first scene has no walls, the second one has one wall in the middle of the room, the third one presented moving animations while the fourth one has the final configuration of the room.

Since the social context requires the presence of people, it can be simulated by inserting animated human models into the virtual environment. Mixamo [26] provides a library of standard humanoid models and animations that can be imported into Unity to simulate people moving within the scene. The animations are managed through the use of an *Animator*, a Unity component that controls the transition between different animation states based on predefined parameters.

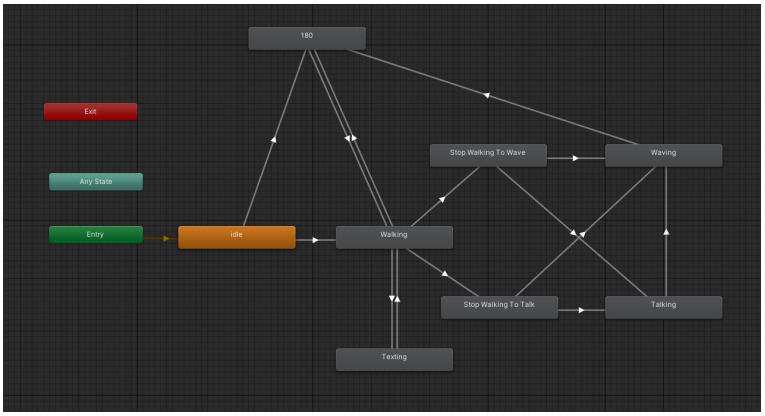


Figure 4.2: Animator configuration

With the Animator configured as shown in 4.2 and adding two scripts (*WallDetector* and *PersonDetector*), the characters in the scene will begin walking, occasionally pausing

to look at their phones before resuming movement. When they encounter a wall, they will turn around, while interactions between two characters are handled probabilistically: upon meeting, they may either stop to converse or exchange a wave before turning around and keep going on their respective paths. All animations representing humans should be tagged as *Person* in the simulation to ensure proper recognition. Once these are added to the scene, the final virtual scene for training is set up.

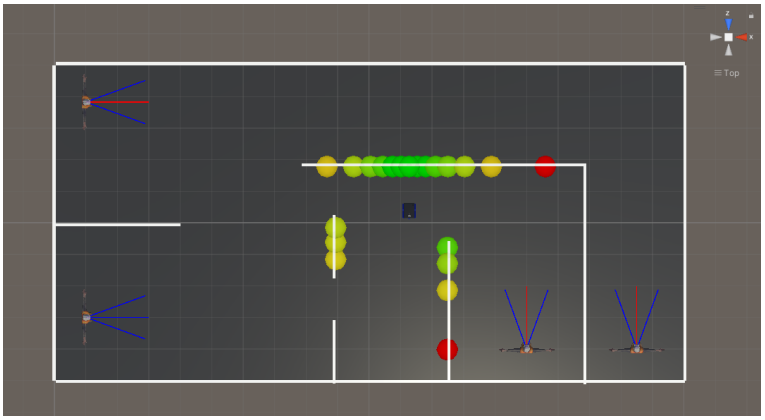


Figure 4.3: Final virtual scene for training

4.2 Unity-Robotics-Hub

In order to import the robot model and enable ROS2 communication within Unity, the *Unity Robotics Hub* [27] package must be installed. This package provides the necessary tools to integrate Unity with robotic frameworks by allowing URDF files to be imported and a communication to be established with a ROS2 system through a TCP/IP connector.

The TCP/IP connector included in the Robotics Hub acts as a communication bridge between the Unity simulation and the ROS2 system. It enables real-time data exchange between Unity and ROS2 nodes through network sockets, allowing Unity to publish sensor data and receive control commands. This communication link is essential for testing robotic algorithms to simulate the robot behavior as accurately as possible.

On the ROS2 side, the *Jazzy* distribution [17] has been used. In order to establish a communication with Unity via TCP/IP, the *ROS-TCP-Endpoint* package [28] must be installed within the ROS2 workspace. The complete tutorial for the installation can be found in the Unity Robotics Hub documentation [27].
Once the package is sourced and built, the endpoint can be launched using:

```
ros2 run ros_tcp_endpoint default_server_endpoint --ros-args -p ROS_IP:=<IP> -p
    ROS_TCP_PORT:=<PORT>
```

This command initializes a communication channel between Unity and ROS2 on the specified IP address and port. If both Unity and ROS2 are running on the same machine,

the IP address can be set to *127.0.0.1*. The port parameter may be omitted or explicitly defined; if unspecified, the default port *10000* is used.

In this project, the simulated robot is the *Jackal* (3.2). By importing a URDF into Unity, a copy of the physical robot can be created. A four-wheeled AGV controller must be added to the Jackal, replacing the controller with the model. The settings for the component are shown in the picture 4.4.

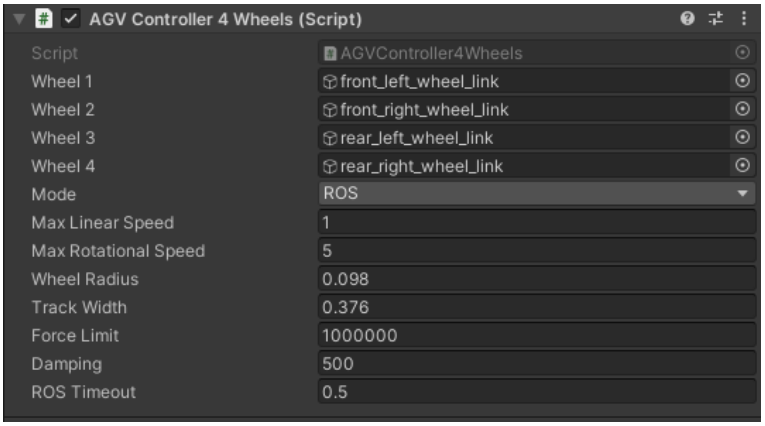


Figure 4.4: Settings for the four-wheeled AGV controller attached to the Jackal robot

Although the URDF model of the Jackal robot already includes a LIDAR system that publishes data through the topic *\scan* and provides a visualization suite displaying the sensor readings as point clouds, an additional custom LIDAR module has been developed for this project. This method allows the user to configure both the number of laser points generated by the sensor and the number of points rendered in the visualization, offering greater flexibility and control over the perception process.

4.3 Agent's implementation

The agent is added to the scene thus creating an empty GameObject that will be referred to as the Agent. Multiple components must be attached thereto:

- *Behavior Parameters*: This component contains all parameters related to the behavior of the agent. In particular, the following items must be specified:
  - *Behavior Name*: This value needs to match the one in the configuration file.
  - *Space Size* and *Stacked Vectors*: These values determine the characteristics of the observation vector.

- *Actions*: This section contains the information on the action branches. In particular, whether the action used are continuous or discrete, how many branches for each type of action, and the values the actions can take.
- *Model*: In the case of inference mode, the model of the trained NN must be inserted into this section. For this purpose, the model needs to be in the project files and can be dragged from the *Assets* window to this section.
- *Behavior Type*: It can be set to *Default*, *Inference Only* or *Heuristic Only*, and determines what type of behavior the Agent carries on.

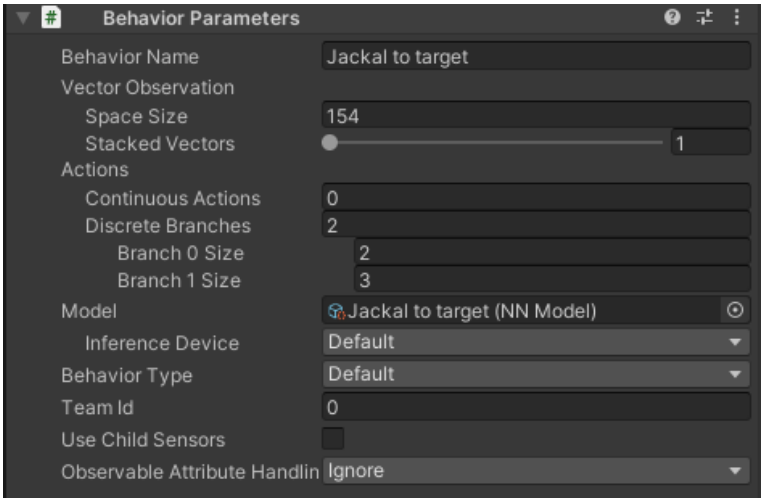


Figure 4.5: Settings for the Behavior Parameters component

- *RobotAgent*: This script is the center of this project and is used to define the Agent. Multiple parameters must be specified in this section to appropriately set the Agent on the environment in which it is found.
  - *Max Step*: It indicates how many steps the Agent can take before the end of the training. If set to 0, it is ignored and the value specified in the configuration file is used.
  - *Root Articulation Body*: In this section the *ArticulationBody* component of the Jackal robot must be specified. The Agent can thereby follow the movements of the robot and perceive any collisions.
  - *Multiplier* and *Angular Multiplier*: These parameters can be modified to change the value of the twist message sent on the ROS topic *cmd_vel*.
  - *Simulation LIDAR Settings* section: In this section, through the use of the various parameters, it is possible to indicate the characteristics of the LIDAR system to be simulated. *Perception Radius* determines the maximum distance objects are detected by the LIDAR system, *Angle Span* indicates the angle in front of the robot where the rays are spread, and *Ray Count* parameter determines how many rays are casted.

– *Camera People Detection Settings* section: Like the one for the LIDAR, this section contains the parameters for the simulated camera on the robot. *Camera Range* indicates the maximum distance people can be detected, *Camera Angle Span* determines the angle in front of the robot that the camera can see, whereas *Max People To Detect* specifies how many people can be recognized by the robot at the same time. If this last parameter is increased, it is necessary to increase the observation vector size as well, since each additional person detected carries a 3 observation vector.

– *Navigation Settings* section: This section contains the parameters that specify how the robot should navigate the environment. *Optimal Wall Distance* parameter allows the user to specify the distance at which the robot must be kept from the walls when it navigates. *Target Reached Distance* indicates the distance at which the robot must be from the target position in order to consider the task completed. *Target Position* indicates the position of the first target to be reached: if left to values [0,0,0], the position will be chosen randomly. It is mainly used for development debug.

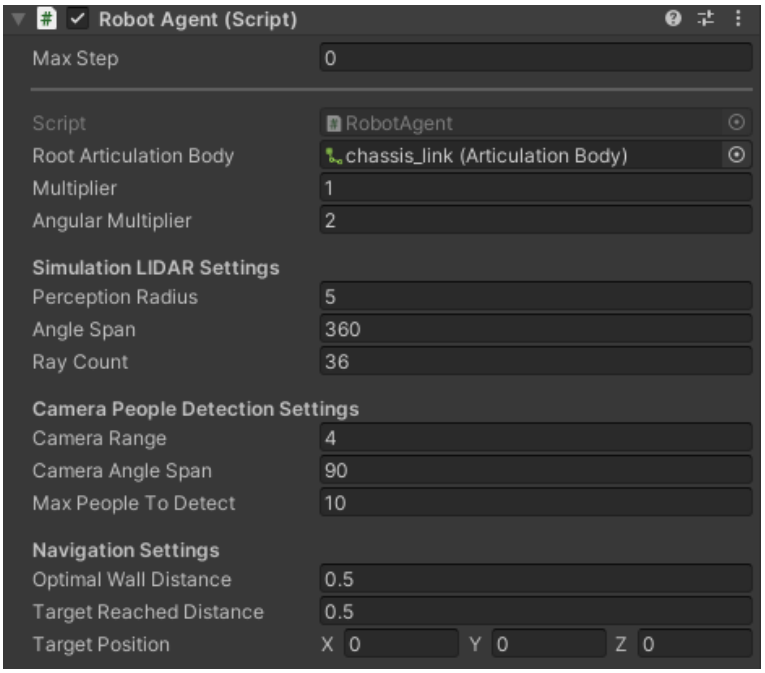


Figure 4.6: Settings for the Robot Agent component

- *Decision Requester*: This component specifies the characteristics of the decisions taken by the model. Here the *Decision Period* can be specified and whether the Agent can take actions between decisions can also be chosen.

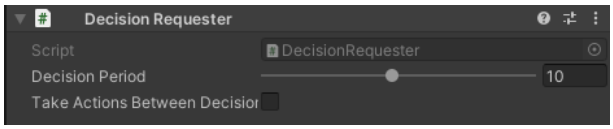


Figure 4.7: Settings for the Decision Requester component

- *Demonstration Recorder*: This component allows recording demonstrations. It has a *Record* button that activates the demonstration recording. Here, the name and the directory for the demonstration can be specified.

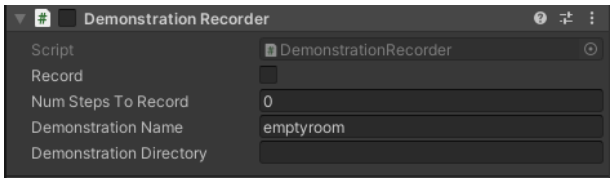


Figure 4.8: Settings for the Demonstration Recorder component

Once these components are attached to the Agent GameObject, the Agent is ready to be used.

To complement this, a new visualization system referred to as the Visualization Suite has been designed. In order to add it to the scene, the scripts of the suite need to be in the Assets directory of the project. In particular, to keep the project organized it is suggested to place them in the Scripts folder. Once the scripts are in the project, a new Tool option will appear in the top bar of the Unity window, referred to as the Visualization Suite. In order to add the visualization tool, simply open this new window and create a new visualization manager.

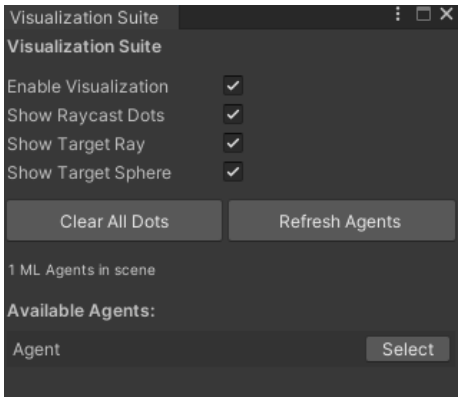


Figure 4.9: Visualization suite tool

Detected points are rendered in the Unity scene with a color coding that reflects the object type and distance: red, yellow, and green indicate walls and obstacles (with red representing the farthest detections and green the closest ones), and white corresponds to detected human tags. The target is visualized in the scene as a white sphere, denoting the point in space that the robot has to reach. Additionally, a dynamic line is drawn between the robot and its target, changing color from red to green when the target becomes directly visible for the robot, allowing the user to easily track the navigation of the robot during simulation. Detected people are surrounded by a visible white cube, as they are perceived by the robot in its observations. All visualization components can be displayed or not by toggling a button in the *Visualization Suite* window.

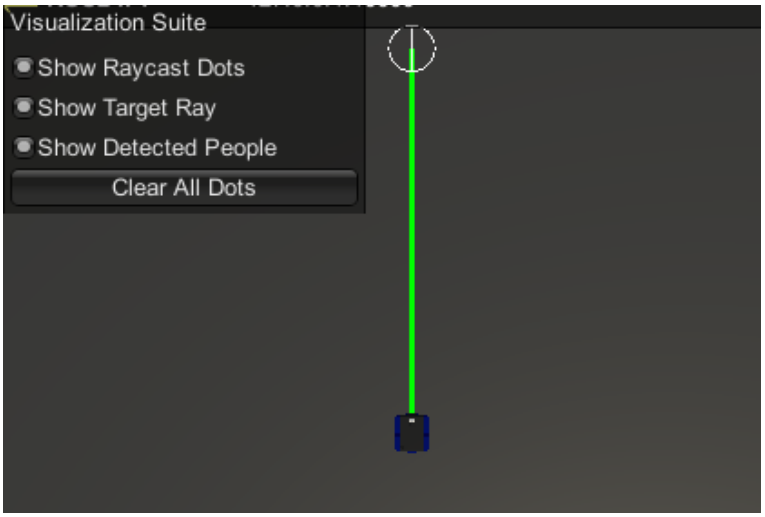


Figure 4.10: Visualization suite window in game mode

Once the environment has been created, the user can record a demonstration, train the Agent, or play a trained model to see its behavior.

4.3.1 Demonstration recording

In order to record a demonstration, the button *Record* in the *Demonstration Recorder* must be active before the play mode is started. The demonstration name and the directory to store it can be specified in the component. For a demo recording, the heuristic behavior must be set in the *Behavior Parameters* section and then, once the scene is started, the user can demonstrate the desired behavior using an input method (e.g., the keyboard or a joystick). The demo will end as soon as the scene is stopped, then it will be stored either in the specified directory or in the directory *Demonstrations* that will be created in the project files.

Longer demos are indicated to show the correct behavior in the maximum number of states possible. The demonstration will map directly each state to an action and can be

used by imitation learning algorithms to make training faster and more performing. All demonstrations recorded for this project have been done using the VR headset (4.4).

4.3.2 Model training

In order to train a new model, the Python package *mlagents* needs to be installed. This package requires Python 3.10.12. To isolate the ambient for the package, a virtual environment has been created. Once inside the environment and python is installed, the package can be installed using the commands

```
cd /<path to ml-agents>
python -m pip install ./ml-agents-envs
python -m pip install ./ml-agents
```

Once this is done, it will be possible to run the *mlagents* commands. In particular, in order to start the training, the command *mlagents-learn* will be required. The next step is the creation of a *.yaml* configuration file containing the parameters indicated in section 3.3.1. The parameters will have to be tuned to mirror the training needs and contain the same behavior name as the Agent present in the Unity scene. The training can then be started using the command

```
mlagents-learn <path to .yaml file> --run-id=<name>
```

exclusively from inside the directory of *ml-agents* package. Additional settings can be added to the training command

- *–force*, used when a training name has already been used, but it is intended to overwrite the model;
- *–initialize-from=<name>*, that allows specifying another model as the starting point of the training;
- *–env=<path to environment>*, to use a specific environment for the training. If this is not specified, the training can still be carried out in the editor by calling the training command in the terminal and then pressing the play button in the editor;
- *–resume*, to resume an interrupted run;

In order to interrupt a training, press *CTRL+C* and wait for the process to terminate.

Once the training is completed, the package will generate a new folder in the *results* directory, named after the run and containing three elements:

- The summaries containing the training metrics that can be used to analyze the training performance and identify the changes to be made in the configuration parameters. This metrics can be displayed using *TensorBoard*.
- The models with extension *.onnx* created at each checkpoint by training. A final model is also created when the training ends or is stopped.
- The timer files that contain aggregated metrics on the training, indicating the time taken for decisions and algorithms updates, for example.

4.3.3 Model running in the environment

Once a model has been trained, it can be run into the environment to verify the behavior that it has learnt. From the *results* folder, the *.onnx* file needs to be copied into the project files. The model needs then to be placed in the *Model* section of the *Behavior Parameters* component. Next, the *Behavior Type* must be set to *Inference Only* to show the actions received only by the model.

By pressing play, the scene will switch to the play mode, and the robot will be under the control of the trained model, moving accordingly with its actions.

4.4 VR headset

The use of a VR headset is indicated for demonstration recording and training or model supervision. The user can enter the scene either to impersonate the robot in order to show the right actions to perform or to modify the scene for inserting randomicity and checking the behavior of the model. The VR headset can be introduced into the environment using the Interaction SDK.

The *Meta Quest 2* is a virtual-reality (VR) headset developed by Meta Platforms Inc. [29]. It is equipped with an integrated Snapdragon XR2 processor, 6 GB of RAM, and two high-resolution LCD displays (1832 × 1920 pixels per eye) running at refresh rates up to 120 Hz. The headset features six degrees of freedom tracking through integrated cameras, which allows users to move within a tracked space without requiring external sensors. It supports *PC VR mode*, where rendering is performed on an external computer and transmitted to the headset via cable or Air Link connection.

The *Interaction SDK* [30] provided by Meta is a high-level library that extends Unity's XR Integration Toolkit. It offers pre-built components for hand-tracking, controller input, grab interactions, and movement management. The Interaction SDK integrates typical VR interactions into modular prefabs such as *HandInteractionController*, *RayInteractor*, and *XR Grab Interactable*. This greatly simplifies the development of interactive scenes and allows developers to manage easily the possible user actions into the scene.

During the development of the training environment, the Meta Quest 2 headset has been integrated to allow the user to enter the scene and record the demonstrations. In order to utilize the Meta Quest 2 headset and its controllers into the Unity environment, the following steps must be carried out:

1. **Install** the *Meta XR SDK* from the Unity Asset Store. Enable *XR Plugin Management* in the Project Settings and select *Oculus* as the provider for both desktop and Android platforms.

2. **Scene configuration.** Add to the scene the building block *Camera Rig*. This object contains the camera and the controllers, managing their position and orientation in the virtual environment. In order to make the user enter the scene it with the point of view of the robot, a new component *Attach Camera Rig* has been created. This script attaches the camera rig to the robot position, updating its position and rotation in each frame.

3. **Record demonstration.** When the headset is connected (via cable or Air Link connection) and the scene is played, the Unity Game View mirrors the user's perspective. The joystick on the right hand is mapped automatically in Unity to send horiziontal and vertical inputs. These are read by the Agent script like keyboard inputs and transformed into ROS commands to the robot. The user can then navigate the environment and record demonstrations to show the robot the desired behavior.

4.5 The code

The most important part for the training of an Agent is the script defining the agent itself. In this section the different functions will be described in terms of structure and purpose. The Agent code, as already said in section 3.3, must define the functions *OnEpisodeBegin()*, *CollectObservations()*, *OnActionReceived()* and *Heuristic()*. Unity functions like *Start()* and *Update()* have been modified to set up the agent. Other functions have been defined to modularize the code and make it clearer. The functions in the code are:

- *ResetRobotPosition()*: This function resets the robot to the position it had at the beginning of the episode, taking its velocity to 0.

- *moveTarget()*: This function calls a new type of class, the *TargetMover*, that moves the target to a new position to be reached by the robot. The user can create a new instance of this class, modifying the rules for the positioning of the target. If no mover is specified, the Agent creates and utilizes a component referred to as the *Normal Target Mover*. With this script, the position of the target is randomly chosen on the object in the scene referred to as the *Floor*. The code also checks that the target is not positioned where other objects are present, avoiding unreachable objectives. The position of the target is also limited to certain areas depending on the point in the training at which the agent is found. The training process has been divided into difficulty steps, each with its own rules for the possible positions of the target:

  - *Very Easy Mode*: The target can be placed at a maximum distance of 4 units, such that there are no objects between it and the robot, and in an angle of 180 degrees in front of it.
  - *Easy Mode*: The target can be placed at a maximum distance of 4 units from the robot.
  - *Medium Mode*: The target can be placed at a maximum distance of 8 units from the robot.
  - *Hard Mode*: The target can be placed anywhere in a valid position.

- *Awake()*: In this function, the ROS connection is initialized. This process is placed in the *Awake()* function instead of the *Start()* function since it is of maximum importance that the ROS connection is immediately established.

- *Start()*: This function initializes the environment for the training. Since the Jackal model consists of multiple components, each with its own collider, a component *ArticulationCollisionForwarder* created for this project is added to each of the child objects of the model. If the position of the target has not been changed in the Unity editor, the *moveTarget()* function is called to place it randomly.
- *Update()*: This function is called in every step of the training and is used to update the scene. First, it keeps the position of the Agent anchored to that of the robot. It then checks if the robot fell or flipped to reset its position in such cases. The timeout for the episode is checked, thus ending the episode if the robot took too much time to reach the target. Since the target is not a physical object, the task is considered completed when the robot reaches a certain position close enough to the target position. This function also carries out this check, ending the episode if the task has been successfully completed.
- *SetTargetPositionFromCurrentPosition()*: This function lies behind the component section *Target Position*. This function allows the user to visualize the position of the target in the edit mode as well.
- *OnEpisodeBegin()*: This function is called each time an episode starts. The number of episodes is increased and the episode timer reset. If the target is reached, it is moved to another position. The current position becomes the start and reset position for the robot, until a new target is reached. If the episode ends for other causes, the robot position is reset to try again to carry out the task. If animations are present in the scene, their position is reset every 10 episodes to make the start of each episode different in configuration.
- *CollectObservatons(VectorSensor sensor)*: This function collects the observations to define the state of the Agent. The information added using *sensor.AddObservation()* must describe the environment so that the agent can learn how to behave depending on the state in which it is found. The observations added are:
  - The linear and angular velocity of the robot, overall $3 + 3 = 6$ observations.
  - The direction to the target, normalized, adding 3 observations.
  - The direction in which the robot is oriented, 3 more observations.
  - The distance to the target, one additional numerical observation.
  - The observations of the simulated LIDAR, using the function *addSimulationLidarObservations(sensor)*. The LIDAR will add *raycount*$*$3 observations. This value can be changed in the *Ray Count* parameter of the Agent component.
  - The observations of the simulated camera, using the function *addPeopleCameraObservations(sensor)*. The camera will add *maxPeopleToDetect*$*$3 observations.This value can be changed in the *Max People To Detect* parameter of the Agent component.

  The final number of observations is calculated as

$$3 + 3 + 3 + 3 + 1 + raycount * 3 + maxPeopleToDetect * 3$$

Using the default values $rayCount = 36$ and $maxPeopleToDetect = 10$, the number of observations is in total 154. This value need to be put in the *behavior Parameters* component, in the section *Space Size* of the *Vector Observations*.

- *addSimulationLidarObservations(VectorSensor sensor)*: This function simulates a LIDAR sensor. It casts *rayCount* rays at an angle *angleSpan* in front of the robot to detect any other object inside the perception range, with maximum distance *perceptionRadius*. All parameters can be indicated in the Agent component in the *Simulation LIDAR Settings*. Since the robot model consists of many GameObjects, the perception is designed to ignore the objects that are children of the robot and perceive only those that are external. For each ray, one vector with three coordinates indicating the relative position of the detected object from the robot's point of view is passed as an observation.

- *addPeopleCameraObservations(VectorSensor sensor)*: This function simulates the work of a camera in people recognition. In the simulation, it is still based on the casting of rays and detection of people around the robot. The animations and the user with the VR set in the scene need to have assigned the tag *Person* in order to make them recognizable for the simulated camera. People can be recognized once they enter the camera range with maximum distanc*cameraRange* and angle span *cameraAngleSpan*. Both parameters can be modified in the Agent component in the *Camera People Detection Settings*. Once a person is detected, a vector with three coordinates indicating the relative position of the person is passed to the Agent as an observation. The camera is designed to perceive a maximum of *maxPeopleToDetect* to make the number of observation not variable. If fewer people are detected, the remaining observation will be passed as zero vectors.

- *GetDetectedPeopleInView()*: This function is necessary to pass the people GameObjects that are perceived by the robot to the *Visualization Suite*. A white box surrounding the person in the camera view will thus appear in the scene when a person is detected.

- *CalculateReward(bool TargetCollision, bool ObstacleCollision, bool PersonCollision, bool Timeout)*: This function computes the reward to be sent to the Agent. When testing the performance of the Agent, the reward function takes different forms:

  - **Event-based reward:** The function is called only when an event occurs, such as collision, reached targets, or timeouts. A reward value is assigned for each event, taking its importance into account. In general terms, the highest penalty is given to a collision with a person, while timeout and wall collision can have smaller penalties. If the target is reached, a substantial reward is assigned.
  - **Event-based reward with periodic rewards:** By maintaining the rewards assigned to the events, some periodic rewards have been added to stimulate and provide guidance to the robot for faster task performance. Periodic rewards are given to the robot every *rewardTimeout* and can be rewards/penalties for

the distance travelled towards or away from the target, or penalties on the time elapsed without reaching the goal to speed up the task.

- **Event-based reward with on action rewards:** Rewards can also be assigned to each action taken by the Agent, in order to give an immediate feedback of the consequences of that action. This method can be too heavy with respect to the computational power and the hardest to tune, since the number of actions during each episode is very high.

The possible rewards that have been tested are described in section 3.3. The reward structure chosen for carrying on test is instead described in chapter 3.4.

- *OnActionReceived(ActionBuffers actionBuffers)*: This function maps the actions sent by the Agent with the actual commands sent to the ROS system. Discrete actions have been chosen over continuous actions to make the training easier and allow the robot to make a limited range of movements. During testing, two different configurations have been analyzed:
  - **One Discrete Branch:** Only one branch per action has been defined. The Action can take values from 0 to 3, each corresponding to move forward, turn left, or turn right. The robot is limited to take these actions one at the time. This limits the possibilities of the robot and can help with action rewards. The command is sent to the ROS system using the *SendTwist(forward,turn)* function.
  - **Two Discrete Branches:** Two branches are defined, one for the forward command and one for the turning one. To avoid backwards movement, the forward branch has been limited to only two values, 1 and 0, indicating going forward and staying still. For the turning branch, three values are possible, 0 for not turning, 1 for turning left, and 2 for turning right. The command is sent to the ROS system using the *SendTwist(forward,turn)* function.
- *SendTwist(float forward, float turn)*: This function receives the forward and twisting intended movements and encodes them in a *TwistMsg*, suitable for ROS messages. This command is then sent on the *\cmd_vel* topic.
- *CheckAndResetIfTipped()*: This function uses the rotation of the robot body to check if the robot has flipped upside down.
- *OnRobotCollision(GameObject other)*: This function has been defined to replace the *OnCollisionEnter(Collision collision)* since the Jackal model consists of multiple objects. In the initialization, the component *ArticulationCollisionForwarder* has been attached to each child object of the robot. This components carries a script that calls the Agent's function *OnRobotCollision(GameObject other)* whenever the *OnCollisionEnter(Collision collision)* function is called on any of these objects. Therefore, when any part of the robot collides with another external object, the Agent will be updated. Collisions are seen as final events, so this function calls the *CalculateReward(bool TargetCollision, bool ObstacleCollision, bool PersonCollision, bool Timeout)* method and ends the current episode.

- *Heuristic(in Actionuffers actionsOut)*: This function maps the users inputs to the Agent actions. As for the *OnActionReceived(ActionBuffers actionBuffers)* method, two configurations have been tested:
  - **One Discrete Branch:** Both vertical and horizontal inputs from the user are mapped on the same branch. In particular, using the keyboard, the upwards arrow corresponds to value 1, the left arrow to value 2, the right arrow to value 3, and no key pressed correspond to value 0.
  - **Two Discrete Branches:** When two branches are defined, the vertical commands are given to the first branch while the horizontal ones go to the second one. The upwards arrow corresponds to value 1 in the first branch, while the downwards arrow can indicate value 2, depending if backwards movement is allowed. No vertical key pressed indicates no linear movement. The left arrow corresponds to value 1 on the second branch, while the right arrow to command 2. No horizontal key pressed corresponds to no turning movement.

Chapter 5

Results

In order to visualize and confront the results obtained from the different training methods described in chapter 3, this chapter presents the training performance and test outcomes for each method. The focus is on the agent's ability to navigate to targets placed at various distances and orientations, while avoiding obstacles and people animations in the environment.

The configuration for the models has been kept constant to ensure the same conditions for all methods. The default hyperparameters provided by Unity ML-agents for the PPO, BC and GAIL algorithms were used, changing some parameters to improve training performance:

- *time_horizon: 128*: This parameter was increased from the default 64 to allow the agent to collect more experience before each policy update, which can help in environments with delayed rewards. Since the rewards are event-based and sparse, a longer time horizon allows the agent to better associate actions with their outcomes.

- *batch_size: 64* and *buffer_size: 1024*: These values were kept small since discrete actions are used, allowing for more frequent updates to the policy.

- *hidden_units: 256*: The size of the neural network's hidden layers was increased from the default 128 to provide the model with more capacity to learn complex patterns in the environment. The high number of observations justifies a larger network.

- *beta: 1.0e-2*: This value was increased from the default $5.0e-3$ to encourage more exploration during training, which is beneficial in sparse reward settings. In tests with smaller values, the agent tended to converge to suboptimal behaviors due to insufficient exploration.

- *beta_schedule: constant*: The exploration parameter was kept constant to maintain a high level of exploration throughout the training process.

- *epsilon_schedule: constant*: The clipping parameter for PPO was kept constant instead of making it decrease linearly over time. This helps to maintain stable

updates to the policy, avoiding making the policy converge too quickly to suboptimal solutions.

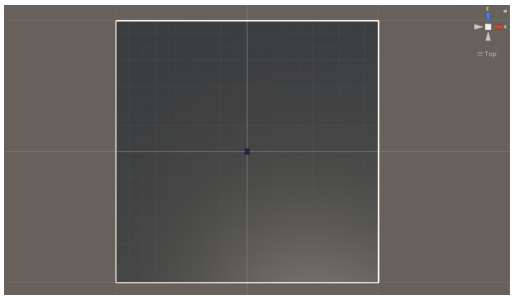
- *gamma: 0.995*: The discount factor for future rewards was set slightly higher than the default 0.99 to focus on long-term rewards, which is important in navigation tasks where the goal is reached after a sequence of actions.

For the configuration of imitation learning methods, both Behavioral Cloning (BC) and Generative Adversarial Imitation Learning (GAIL) were trained using four dataset of 300 demonstrations collected by manually controlling the robot in the environment using the VR headset. The demonstrations were recorded in the same four environments used for training, to provide the models with experience in all scenarios they would encounter. Both BC and GAIL trainings used the same configuration of PPO, with strenght of the imitation learning reward signal set to 1.0. The reward structure chosen for carrying on the trainings and the tests is described in chapter 3.4.

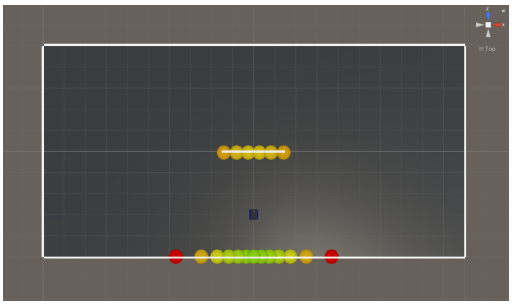
5.1 Trainings

Each model has been trained for 500000 steps in four different environments, to ensure the same amount of training time for all methods. Each training session took approximately 4 hours for environment. The first environment (Figure 5.1a) is an empty room with only the perimeter walls, to train pure navigation skills. The second environment (Figure 5.1b) included a static obstacle between the area where the agent starts navigating and the target location, to encourage obstacle avoidance behavior. The third environment (Figure 5.1c) added moving people animations to simulate a crowded environment, while the fourth environment (Figure 5.1d) combined both static obstacles and moving people. The results presented in this chapter show the performance of the models in various test scenarios after training in these environments.

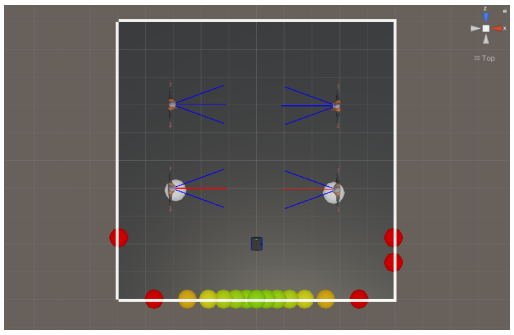
In order to evaluate the effect of reinforcement learning applied on a imitation learning model, tests were also conducted on models trained firstly with BC or GAIL, followed by further training with PPO (again of additional 500000 steps). The results are compared with models trained solely with PPO, BC, and GAIL.
After each training session, its performance was evaluated observing the reward curve and the episode length graph obtained in the TensorBoard logs.



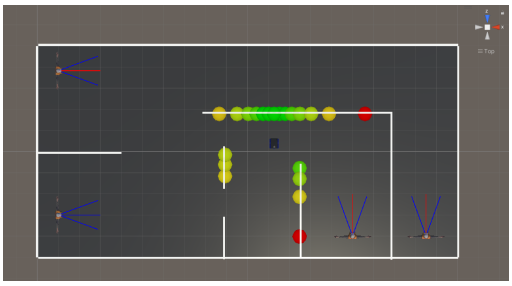
(a) Empty room environment



(b) Obstacle avoidance environment



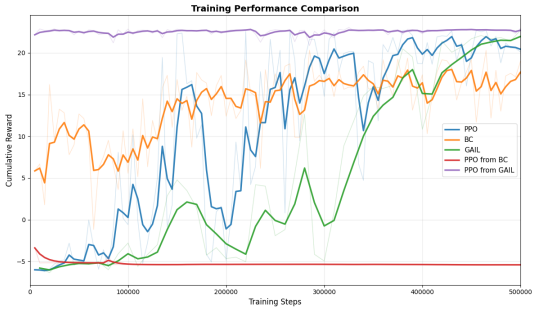
(c) People avoidance environment



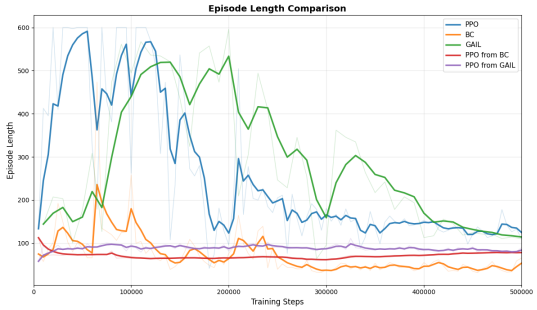
(d) Full environment

Figure 5.1: Training environments used for the training sessions

5.1.1 Empty room training



(a) Cumulative reward curve for training in empty room environment



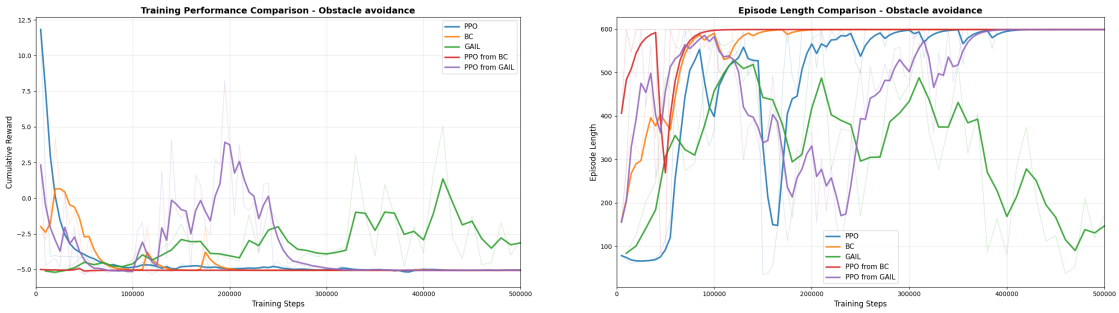
(b) Episode length curve for training in empty room environment

Figure 5.2: Training metrics for empty room environment

Observing the cumulative reward and confronting with the episode length it is possible to derive what kind of behavior the robot had during the training. In the empty room environment (Figure 5.2) the agent learned to navigate to the target with almost all methods. PPO shows a good exploration with initial longer episodes, that get shorter as the agent learns to reach the target faster. GAIL also shows a good performance, with a steady

increase in cumulative reward and decreasing episode lengths. BC shows a better start, with a quick rise in cumulative reward due to the imitation of expert demonstrations, but then plateaus, indicating limited further learning. The combined methods (BC+PPO and GAIL+PPO) show mixed results, with GAIL+PPO improving greately over GAIL alone, while BC+PPO shows a drastic drop in performance. Observing the behavior learned by the model, the BC+PPO made no additional exploration and converged to a suboptimal behavior, bumping into walls as soon as possible to avoid additional time penalties.

5.1.2 Obstacle avoidance training



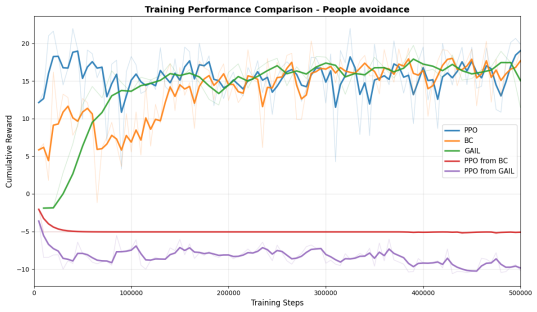
(a) Cumulative reward curve for training in the obstacle avoidance environment

(b) Episode length curve for training in the obstacle avoidance environment

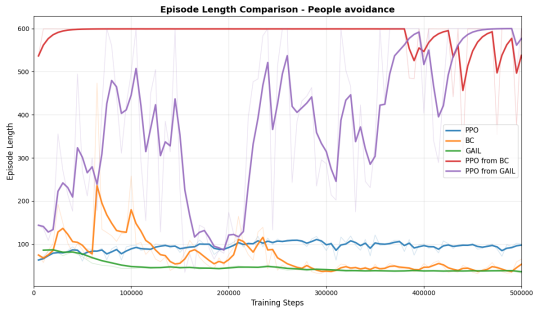
Figure 5.3: Training metrics for the obstacle avoidance environment

The addition of obstacles in the robot's path (Figure 5.3) made the training more challenging. An obvious drop in performance was made by all methods, with sporadic successful navigation from GAIL methods. The general behavior learned by all other methods is a standing one, to avoid penalties for bad exploration and obstacle collisions. This is verified by the steep increase of the episode length and constant negative cumulative reward. The only method that keeps positive exploration is GAIL method, moved by the intrinsic rewards to act similarly to the expert actions. Generally, all methods encounter great difficulties moving around big size objects, not having any reward system to guide their obstacle avoidance learning.

5.1.3 People avoidance training



(a) Cumulative reward curve for training in the people avoidance environment

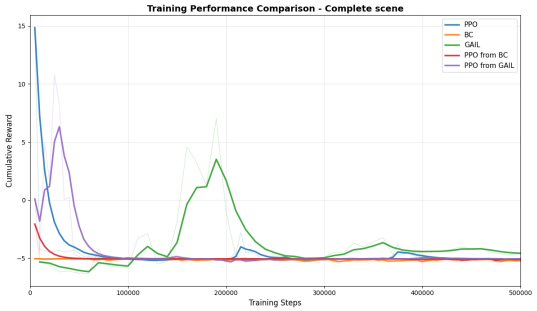


(b) Episode length curve for training in the people avoidance environment

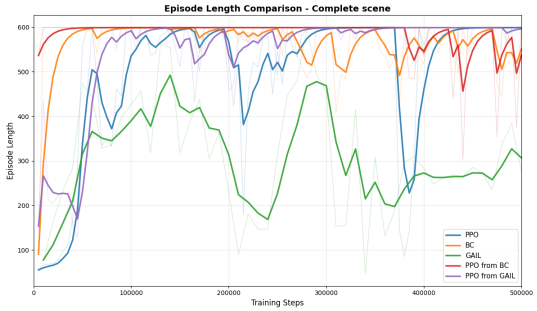
Figure 5.4: Training metrics for the people avoidance environment

People avoidance training (Figure 5.4) showed better results than obstacle avoidance, with PPO, BC and GAIL methods learning to navigate around moving people. PPO+BC method kept long episodes and constant rewards, converging again to a standing behavior. Since the person animation can be seen as a narrow moving obstacle, the possibility to avoid such obstacle with less actions with respect to an extended wall explains the better performance. Observing the model behavior with GAIL, the robot is able to wait for the obstacle to go out of its path and then reach its goal, as it was shown in the demonstrations. The other two successful methods tend to avoid people animations by moving through them.

5.1.4 Full environment training



(a) Cumulative reward curve for training in the complete environment



(b) Episode length curve for training in the complete environment

Figure 5.5: Training metrics for the complete environment

As expected from the obstacle avoidance training, the complete environment (Figure 5.5) presented the greatest challenge for all methods. The agent struggled to learn effective navigation strategies, with all methods showing long episode lengths and negative cumulative rewards. GAIL method showed some positive exploration, but still failed to consistently reach the target. The other methods converged to standing behaviors, avoiding movement to minimize penalties. This indicates that the combination of static obstacles and moving people created a complex environment that was difficult for the agent to navigate without more sophisticated learning strategies or reward structures.

5.2 Tests

Several tests have been conducted to evaluate the performance of the different training methods described in Chapter 3.1. The tests focus on the agent's ability to reach targets placed at various distances and orientations, behind obstacles or in the presence of people animations. Each test consists of 10 trials per configuration, and success is defined with the agent reaching the target before timeout. All tests were conducted in a controlled environment, with the agent starting from a fixed position and orientation.

5.2.1 Orientations tests

For the first batch of tests, the target was placed at various distances (5, 10, and 15 units) and orientations (0°, 45°, 90°, 135°, 180°, 225°, 270°, and 315°) relative to the agent's starting position. Each configuration was tested 10 times to account for variability in performance. The agent's success rate was recorded for each method and configuration.

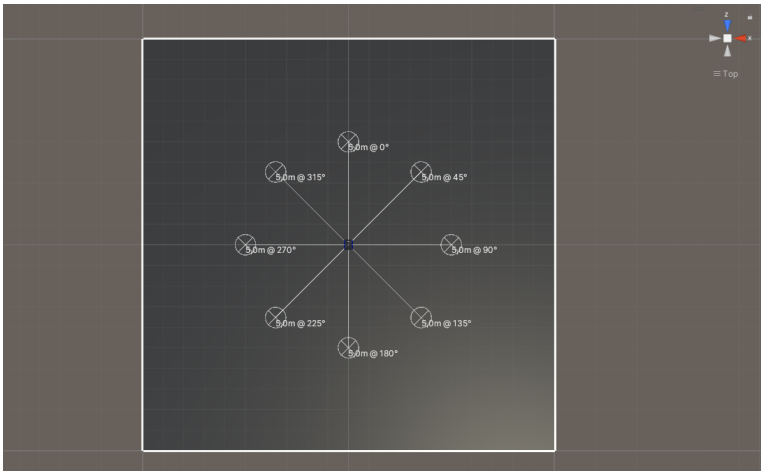


Figure 5.6: Target placements used in the orientation tests

Table 5.1: Performance Across Angular Target Orientations, 5 units distance

Method	0°	45°	90°	135°	180°	225°	270°	315°
PPO	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
GAIL	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
GAIL+PPO	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10
BC	5/10	3/10	0/10	0/10	0/10	0/10	0/10	2/10
BC+PPO	1/10	0/10	0/10	0/10	0/10	0/10	0/10	0/10

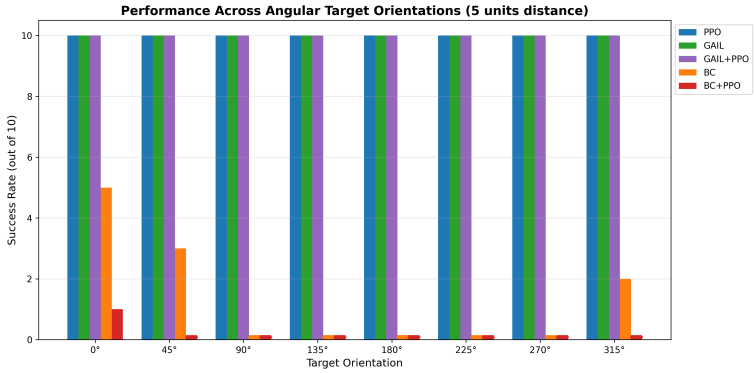


Figure 5.7: Performance Across Angular Target Orientations, 5 units distance

While PPO, GAIL, and GAIL+PPO methods demonstrated perfect performance across all target orientations at a distance of 5 units, BC and BC+PPO methods struggled significantly. BC managed to reach the target only when it was directly in front of the agent (0°) or slightly moved, while BC+PPO showed almost no success. This poor navigation ability of BC-based methods can be attributed to the limits of BC training, which relies heavily on the quality and diversity of the demonstration data. Since the problem analyzed is very complex, the demonstrations didn't cover all possible scenarios. As a result, the BC-trained models failed to identify a good policy for a consistent positive navigation, converging to suboptimal behaviors when the target was not directly in front of them. The addition of PPO training after BC did not improve performance, likely because the initial policy learned from BC was too weak to benefit from further reinforcement learning.

5.2.2 Distance tests

The second batch of tests focused on the ability of the agent to reach targets placed at varying distances (5, 6, 7, 8, 9, and 10 units) from its starting position. The targets were placed in the four cardinal directions to evaluate performance across different orientations. Each configuration was tested 10 times, with best performance expected when the target is in front of the agent and worst performance when the target is behind the agent.

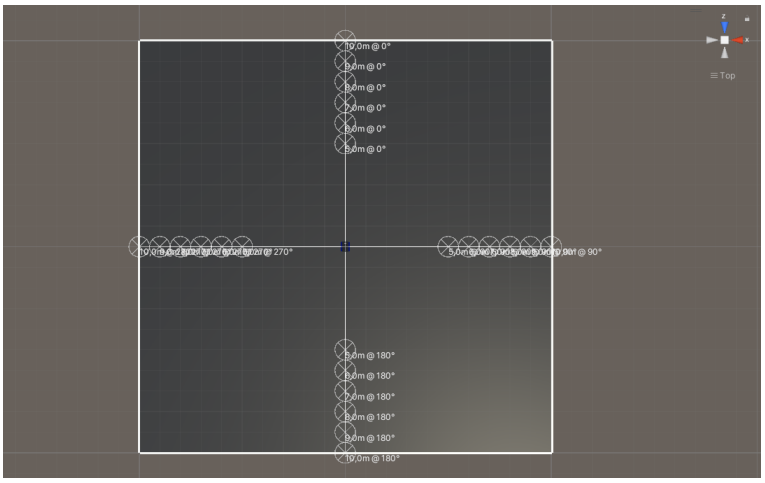


Figure 5.8: Target placements used in the distance tests

Table 5.2: Performance Across Distance variations, 0° target orientation

Method	**5 units**	**6 units**	**7 units**	**8 units**	**9 units**	**10 units**
PPO	10/10	10/10	10/10	10/10	10/10	10/10
GAIL	10/10	10/10	10/10	10/10	10/10	10/10
GAIL+PPO	10/10	10/10	10/10	10/10	10/10	10/10
BC	6/10	3/10	7/10	10/10	2/10	10/10
BC+PPO	0/10	0/10	0/10	0/10	0/10	0/10

Table 5.3: Performance Across Distance variations, 90° target orientation

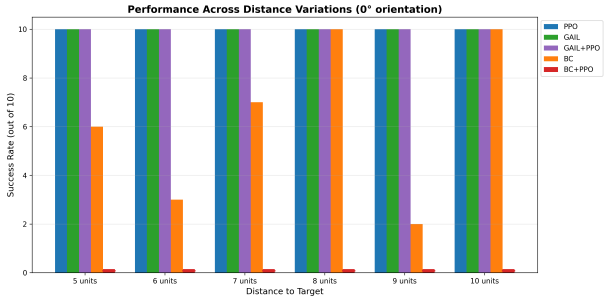
Method	**5 units**	**6 units**	**7 units**	**8 units**	**9 units**	**10 units**
PPO	10/10	10/10	10/10	10/10	10/10	10/10
GAIL	10/10	10/10	10/10	10/10	10/10	10/10
GAIL+PPO	10/10	10/10	10/10	10/10	10/10	10/10
BC	0/10	0/10	0/10	0/10	0/10	0/10
BC+PPO	0/10	0/10	0/10	0/10	0/10	0/10

Table 5.4: Performance Across Distance variations, 180° target orientation

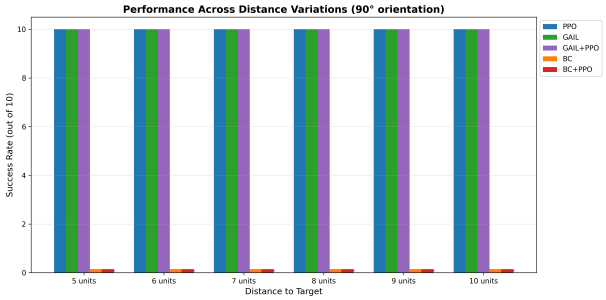
Method	**5 units**	**6 units**	**7 units**	**8 units**	**9 units**	**10 units**
PPO	10/10	10/10	10/10	10/10	10/10	10/10
GAIL	10/10	10/10	10/10	10/10	10/10	10/10
GAIL+PPO	10/10	10/10	10/10	10/10	10/10	10/10
BC	0/10	0/10	0/10	0/10	0/10	0/10
BC+PPO	0/10	0/10	0/10	0/10	0/10	0/10

Table 5.5: Performance Across Distance variations, 270° target orientation

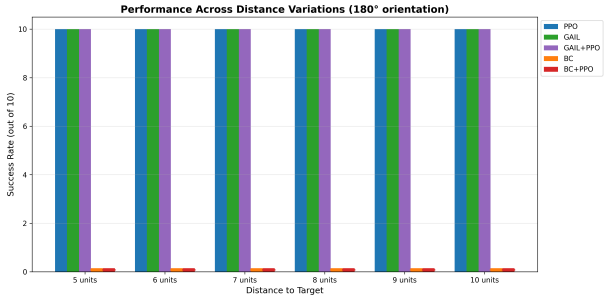
Method	5 units	6 units	7 units	8 units	9 units	10 units
PPO	10/10	10/10	10/10	10/10	10/10	10/10
GAIL	10/10	10/10	10/10	10/10	10/10	10/10
GAIL+PPO	10/10	10/10	10/10	10/10	10/10	10/10
BC	0/10	0/10	0/10	0/10	0/10	0/10
BC+PPO	0/10	0/10	0/10	0/10	0/10	0/10



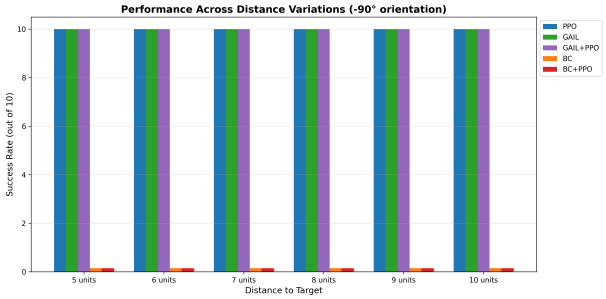
(a) Performance Across Distance Variations, 0° target orientation



(b) Performance Across Distance Variations, 90° target orientation



(c) Performance Across Distance Variations, 180° target orientation



(d) Performance Across Distance Variations, 270° target orientation

Figure 5.9: Tests results across distance variations at different target orientations

Similar to the orientation tests, PPO, GAIL, and GAIL+PPO methods demonstrated optimal performance across all distances and orientations. BC method showed inconsistent performance, managing to reach the target only at certain distances when it was directly in front of the agent (0° orientation). The performance dropped to zero for BC at other orientations, indicating poor exploration and navigation capabilities. BC+PPO method remained consistent on the standing behavior, failing to reach the target in any configuration. These results further highlight the limitations of BC-based methods in complex tasks, where it is difficult to provide a set of demonstrations covering all possible scenarios the agent might encounter.

5.2.3 Obstacle avoidance tests

Next, the obstacle avoidance scenario has been analyzed. The robot needed to reach a target distant 6 units at 0° having an obstacle on its way. The success rate with various obstacle dimensions has been collected.

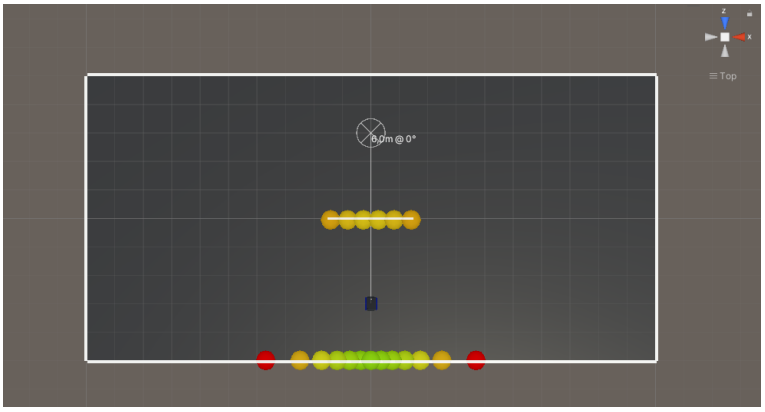


Figure 5.10: Target placements used in the obstacle avoidance tests

Table 5.6: Performance across obstacle avoidance scenarios, 0° target orientation, different obstacle dimensions

Method	**1 unit**	**2 units**	**3 units**	**4 or more units**
PPO	8/10	8/10	3/10	0/10
GAIL	7/10	3/10	0/10	0/10
GAIL+PPO	3/10	3/10	0/10	0/10
BC	4/10	3/10	1/10	0/10
BC+PPO	1/10	0/10	0/10	0/10

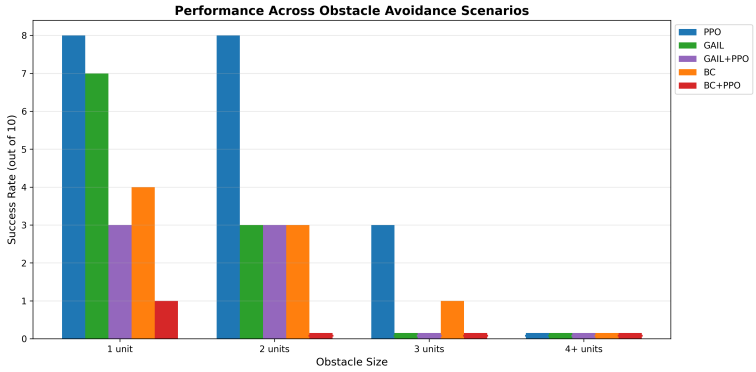
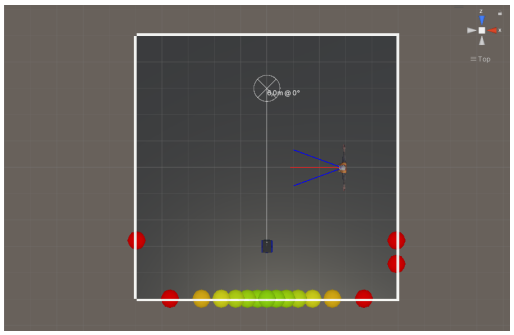


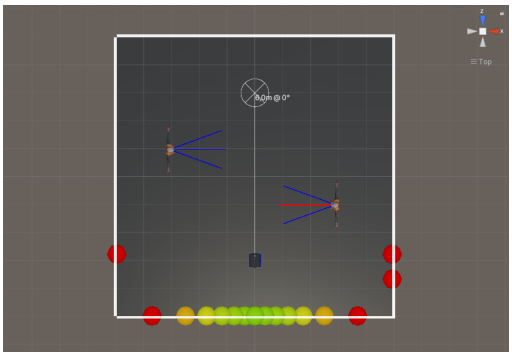
Figure 5.11: Performance across obstacle avoidance scenarios, 0° target orientation, different obstacle dimensions

The results indicate that as the obstacle size increases, the success rates for all methods drops. PPO maintains a relatively higher performance compared to other methods, while BC and BC+PPO struggle significantly with larger obstacles. GAIL shows moderate performance but also quickly declines with increasing obstacle size. Observing the quality of the performance, BC methods learned a standing behavior in front of the obstacle, while PPO and GAIL were able to navigate around, ending the episode closer to the target with a collision or a success. These results on are probably due to the structure of the reward function, that reinforces only behaviors that decrease the distance with the target, without rewards or incentives for obstacle avoidance. GAIL+PPO also opted for a more conservative behavior, navigating with success the area getting closer to the obstacle and the target, but then converging a standing behavior with only slight orientation changes. PPO reached the best performance in the obstacle avoidance tests due to the configuration of the training. Since the exploration parameter *beta* was increased and set to constant, with an higher exploration PPO was able to explore more and find paths around the obstacles.

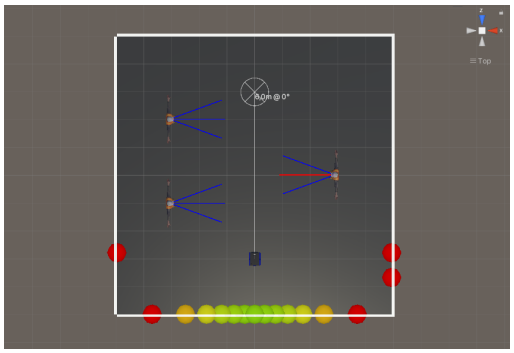
5.2.4 People avoidance tests



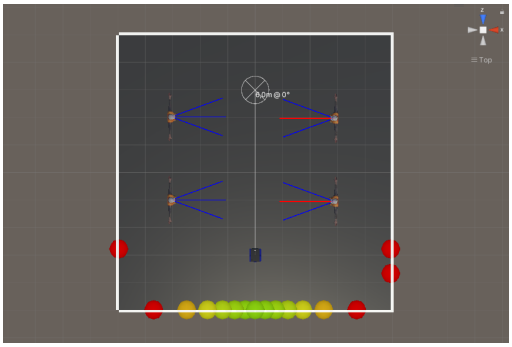
(a) Animation configuration with 1 person



(b) Animation configuration with 2 people



(c) Animation configuration with 3 people



(d) Animation configuration with 4 people

Figure 5.12: Training environments used for the people avoidance tests

Table 5.7: Performance across person avoidance scenarios, 0° target orientation, different number of people

Method	**1 person**	**2 people**	**3 people**	**4 people**
PPO	6/10	5/10	5/10	5/10
GAIL	10/10	9/10	6/10	7/10
GAIL+PPO	1/10	1/10	0/10	1/10
BC	8/10	7/10	6/10	5/10
BC+PPO	0/10	0/10	0/10	0/10

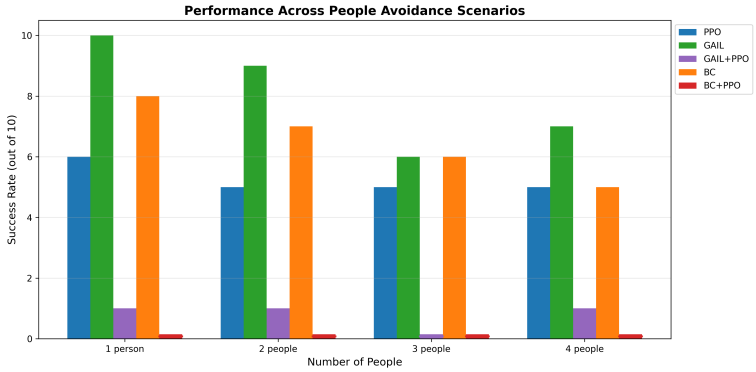


Figure 5.13: Performance across person avoidance scenarios, 0° target orientation, different number of people

Since the animations move randomly during the tests, to get the statistics for the people avoidance several batches of 10 episodes have been run, then the average number of successes rounded by defect has been taken (shown in table 5.7 and figure 5.13). In this scenario, GAIL outperforms other methods, maintaining high success rates even as the number of people increases. PPO shows a moderate decrease in performance with more people, while BC performs relatively well compared to its results in other scenarios. The combined methods (BC+PPO and GAIL+PPO) show poor performance, mostly because this models learned a standing behavior to avoid large penalties. With four people present, since the animations are programmed to stop and run a talk animation when they meet, the agent found it slightlyeasier to navigate with all algorithms, since the people were less likely to cross its path. The high performance of GAIL can be explained by observing the effective behavior learned by the model. The demonstration of the expert contained avoidance behavior alongside waiting behavior, keeping distance from the animation and moving only when the path became free. GAIL model learned this behavior, that was applied during several of the trials in the people avoidance environment. The improved performance of the BC method can be explained by the quality of the demonstrations. Since the animations move randomly around the robot, each trajectory covers many possible states, increasing the visited space. The BC method is then able to learn successfully, having more probability to encounter demonstration states.

Chapter 6

Future works and conclusions

This project has analyzed the possibilities to use demonstrations for training robots, in particular to carry out navigation tasks in social environments. This idea has been developed in a simulated environment in Unity to allow repeated testing in a safe and confined environment, highly customizable and easy to use. Unity allows for the presence of the user in the scene by virtue of the use of a VR headset that has allowed a more socially focused testing. The results obtained have shown that reinforcement learning algorithms and imitation learning algorithms suffer from convergence problems during training, leading to suboptimal policies. While BC has shown critical flaws in terms of successful navigation, PPO and GAIL have shown better results, even if not completely satisfactory. With sparse rewards and complex environments, the learning process carried out by this algorithms has been successful only in clear environments, while the presence of large obstacles like walls has led to poor performances. The presence of people in the environment was less decisive in affecting the performance of the agents, since the models learned to avoid small obstacles. However, collision with people should be complitely avoided to ensure safe navigation in social environments, so the model cannot be considered reliable. The tests demonstrated that the combination of imitation learning and reinforcement learning, in particular PPO, enhanced the issues related to this method, leading to convergence to standing policies and suboptimal behaviors. Considering all tests, the most reliable algorithm for training the robot has been GAIL, which has shown the best performance in terms of navigation success and people avoidance. The use of demonstrations as auxiliary reward function has allowed the agent to have feedback on its actions even in the presence of sparse rewards, leading to better exploration and more stable learning. However, also this algorithm has shown limitations in terms of navigation success in complex scenes, leading to the conclusion that further improvements are needed to achieve reliable navigation in social environments.

In order to extend the analysis conducted in this project, the next steps can be the implementation of continuous actions instead of discrete ones, and the analysis of performance on longer curriculum trainings. Due to the high variability in the design of reward functions, other distributions of rewards can be analyzed to find better performing solutions. This work has considered the use of a single environment for each training, since the connection with the ROS2 system did not support multiple robots or multiple

connections. The development of a way to train multiple robots with the same agent can be an important step to speed up the training process and improve the learning performance. To improve the quality of the trainings with sparse rewards, the implementation of curiosity-driven exploration methods can be considered, to allow the agent to explore the environment more efficiently and discover new strategies for navigation.

Since reinforcement learning and imitation learning navigation are not completely reliable for navigating, future progress in this work could be the integration of navigation frameworks. The idea underlying the use of agents can be changed and they can be implemented in other fashions, such as using them to learn and foresee people's movements, creating new information to improve the path planning and social behavior. The implementation of agents would be centered on learning specific social behaviors, such controlling velocity control or adjusting the trajectory around people, while the navigation is carried out by traditional model-based path planning algorithms.

Chapter 7

Appendix

The code developed to create the environment and carry out the training sessions is available at the following GitHub repository: https://github.com/PIC4SeRThesis/AsiaFerri.
In this chapter are reported additional configurations that have been tested and discarded during the development of the environment. More tests than the ones listed were carried out, but these are the most significant ones that show the evolution of the project and the performance with different settings.

- **Complex reward function with PPO:** A more complex reward function has been tested during the development of the environment, applying all rewards described in section 3.3. This configuration included rewards for maintaining a forward velocity, heading rewards, and other factors. However, this approach led to slower learning and less stable policies, as the agent struggled to reach consistently the target. For this training the timer was set at 3000 seconds, and the event based rewards were kept the same as in the final version of the environment. The training was carried out in the complete environment, with obstacles and people presence.

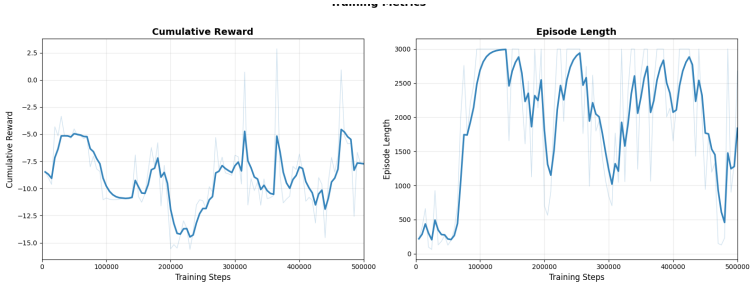


Figure 7.1: Training metrics with complex reward function

The figure 7.1 shows the training metrics obtained with this configuration, where it is possible to see that the average reward is lower and less stable compared to

the simpler reward function used in the final version of the environment. The agent struggled to consistently reach the target, leading to a lower overall performance.

- **Complex reward function with PPO using one discrete branch for Actions:** In order to try making the learning process more stable using the complex reward function, a configuration with one discrete action branch has been tested. The action space included 4 discrete actions: move forward, turn left, turn right and stand still. However, this approach still led to suboptimal performance, as the agent struggled to learn smooth navigation behaviors. As in the previous configuration, the timer was set at 3000 seconds, and the event based rewards were kept the same as in the final version of the environment. The training was carried out in the complete environment, with obstacles and people presence.

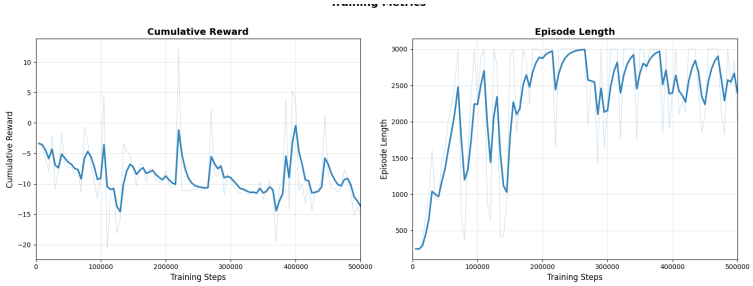


Figure 7.2: Training metrics with complex reward function and one discrete action branch

The behavior learned by the agent with this configuration converged to a standing one, showed by high episode length and low rewards (figure 7.2). The agent preferred to avoid movement to minimize penalties, resulting in poor navigation performance.

- **Medium complexity function with PPO:** A medium complexity reward function has also been tested, including rewards for distance to target and heading, but excluding forward velocity rewards, proximity rewards and time penalties. This configuration aimed to balance simplicity and informativeness in the reward signal. For this configuration the timer was set to 1500 seconds, and the event-based rewards were enhanced to provide more feedback on reaching the target and avoiding collisions. The training was carried out in the complete environment, with obstacles and people presence.

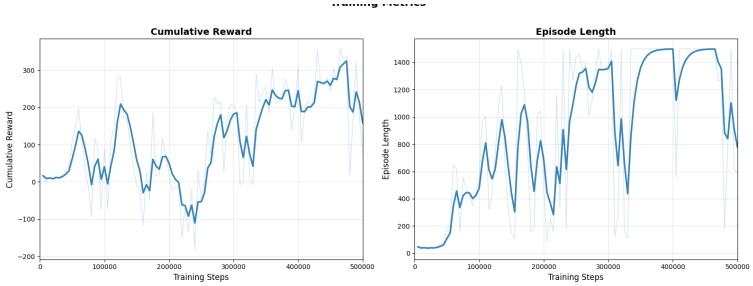


Figure 7.3: Training metrics with medium complexity reward function

While the performance improved compared to the complex reward function, the agent still exhibited instability in learning, as shown in figure 7.3. The PPO algorithm struggled to consistently reach the optimal solution, finally converging to a standing suboptimal policy.

- **Event-based reward function with default configuration:** The final reward function used also to retrieve the results shown in chapter 5 has been tested with the default configuration of the Unity ML-agents framework, to test its performance without hyperparameter tuning. This configuration included a timer set to 1500 seconds, and the training was carried out in the empty room environment.

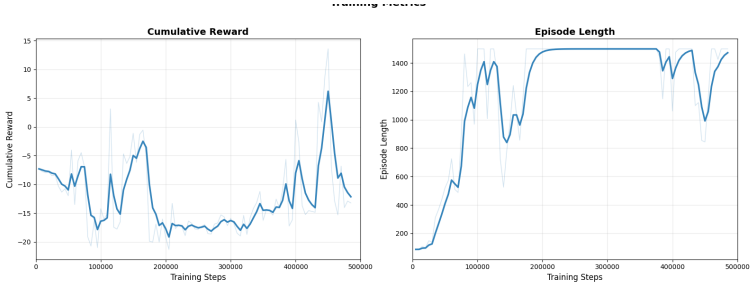


Figure 7.4: Training metrics with default configuration

The exploration process was drastically less efficient with default configuration than with the tuned hyperparameters, as shown in figure 7.4. The agent struggled to learn effective navigation behaviors, resulting in lower rewards and longer episode lengths. This configuration highlighted the importance of hyperparameter tuning in achieving optimal performance with reinforcement learning algorithms.

- **Behavioural cloning implementation with strenght parameter set to** 0.1**:** Since BC methods showed poor performance in the tests described in chapter 5, additional configurations have been tested changing the strenght of the influence of the BC method. One such configuration involved setting the strenght parameter of the BC implementation to 0.1 instead of the full strenght. This parameter controls

the weight given to the imitation loss during training, with lower values placing more emphasis on matching the expert demonstrations. The timer was set to 600 seconds, and the event-based rewards were kept the same as in the final version of the environment. The training was carried out in the empty room environment.

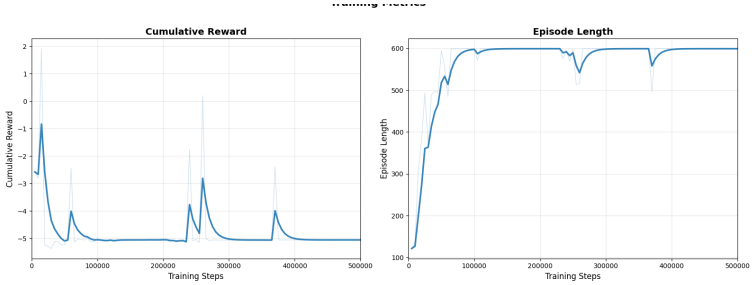


Figure 7.5: Training metrics with BC strenght parameter set to 0.1

With this configuration, the agent showed a slow exploration, converging to a standing behavior as shown in figure 7.5. The reduced emphasis on imitation led to less effective learning, as the agent struggled to generalize from the expert demonstrations. Full strenght BC provided better results in terms of navigation performance compared to this configuration, showing that the agent benefited from a stronger focus on imitation learning in the empty room scenario.

Bibliography

[1] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents, 2020.

[2] Andre Cleaver, Darren Tang, Victoria Chen, and Jivko Sinapov. Haven: A unity-based virtual robot environment to showcase hri-based augmented reality, 2020.

[3] Hasan Seyyedhasani, Daniel Udekwe, and Muhammad Ali Qadri. Comparative evaluation of vr-enabled robots and human operators for targeted disease management in vineyards, 2025.

[4] Sumey El-Muftu and Berke Gur. A robot simulation environment for virtual reality enhanced underwater manipulation and seabed intervention tasks, 2025.

[5] Giacomo Franchini, Brenno Tuberga, and Marcello Chiaberge. Advancing lunar exploration through virtual reality simulations: a framework for future human missions, 2024.

[6] Linqi Ye, Rankun Li, Xiaowen Hu, Jiayi Li, Boyang Xing, Yan Peng, and Bin Liang. Unity rl playground: A versatile reinforcement learning framework for mobile robots, 2025.

[7] Shokhikha Amalana Murdivien and Jumyung Um. Boxstacker: Deep reinforcement learning for 3d bin packing problem in virtual environment of logistics systems. *Sensors*, 23(15), 2023.

[8] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

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

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

[11] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[12] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011.

[13] Dylan J. Foster, Adam Block, and Dipendra Misra. Is behavior cloning all you need? understanding horizon in imitation learning, 2024.

[14] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning, 2016.

[15] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress, 2020.

[16] Clearpath Robotics. *Jackal Description Package — URDF and Meshes for Clearpath Jackal Robot.* ROS Index. https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/.

[17] Open Robotics. *ROS 2 Documentation: Jazzy Jalisco.* ROS.org. https://docs.ros.org/en/jazzy/.

[18] M. Everett, S. Chen, and J. How. "mit develops autonomous 'socially aware' robot using jackal ugv". *Clearpath Robotics – Case-Study.* https://clearpathrobotics.com/customers-robot-case-stories/mit-develops-socially-aware-jackal/.

[19] Unity Technologies. *Unity ML-Agents Toolkit Documentation (Version 4.0)*, 2024. https://docs.unity3d.com/Packages/com.unity.ml-agents@4.0/manual/index.html.

[20] Unity Technologies. Unity ml-agents toolkit. https://github.com/Unity-Technologies/ml-agents.

[21] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

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

[23] Stephane Ross and Drew Bagnell. Efficient reductions for imitation learning. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 661–668, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[24] Sinan Ibrahim, Mostafa Mostafa, Ali Jnadi, Hadi Salloum, and Pavel Osinenko. Comprehensive overview of reward engineering and shaping in advancing reinforcement learning applications, 2024.

[25] Unity Technologies. Unity – real-time development platform. https://unity.com/.

[26] Adobe Systems Incorporated. *Mixamo: 3D Character Animation Platform.* Adobe Inc. https://www.mixamo.com/.

[27] Unity Technologies. *Unity Robotics Hub.* Unity Technologies. https://github.com/Unity-Technologies/Unity-Robotics-Hub.

[28] Unity Technologies. *ROS–Unity TCP Endpoint (ROS 2 Integration Plugin).* Unity Technologies. https://github.com/Unity-Technologies/ROS-TCP-Endpoint.

[29] Meta Platforms, Inc. Meta – social technology company. https://www.meta.com/.

[30] Meta Platforms, Inc. Unity interaction sdk overview — meta horizon. https://developers.meta.com/horizon/documentation/unity/unity-isdk-interaction-sdk-overview/.

[31] Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning, 2018.

[32] Zizhao Wang, Xuesu Xiao, Garrett Warnell, and Peter Stone. Apple: Adaptive planner parameter learning from evaluative feedback. *IEEE Robotics and Automation Letters*, 6(4):7744–7749, October 2021.

[33] Jiequan Cui, Beier Zhu, Qingshan Xu, Zhuotao Tian, Xiaojuan Qi, Bei Yu, Hanwang Zhang, and Richang Hong. Generalized kullback-leibler divergence loss, 2025.

[34] Mauro Martini, Noé Pérez-Higueras, Andrea Ostuni, Marcello Chiaberge, Fernando Caballero, and Luis Merino. Adaptive social force window planner with reinforcement learning, 2024.

[35] Andrew Patterson, Victor Liao, and Martha White. Robust losses for learning value functions, 2023.

[36] Russ Tedrake. *Underactuated Robotics*. https://underactuated.csail.mit.edu.

[37] OpenAI. Proximal policy optimization (ppo) — spinning up in deep rl. https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[38] OpenAI. Proximal policy optimization (ppo) — openai baselines. https://openai.com/index/openai-baselines-ppo/.

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