DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

University of Padua

Department of Information Engineering

Master Thesis in Control Systems Engineering

Evaluating Deep Reinforcement Learning Algorithms for Autonomous Navigation on Edge Devices

Supervisor:

Prof. Gian Antonio Susto

Co-advisor:

Niccolò Turcato

Alberto Sinigaglia

Candidate:

Matteo Dal Nevo

ID: 2087919

Academic Year 2024/2025

"Life is like riding a bicycle.

To keep your balance, you must keep moving."

— Einstein

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Gian Antonio Susto, for inspiring me through his course and for providing invaluable guidance throughout this journey. I also sincerely thank my co-supervisors, Niccolò and Alberto, for their invaluable advice and support throughout this research.
In particular, Niccolò's hands-on guidance during every phase, especially in the laboratory, was instrumental in refining our experimental protocols, troubleshooting hardware issues, and interpreting results, ensuring the success of this work.
Alberto's expertise was equally vital in the initial stages; his assistance in setting up the simulation environment, developing the computational models, and validating early results laid the groundwork for all subsequent experiments.

My deepest gratitude goes out to all the people who have been by my side during these years. My family, Danilo and Claudia, and my "second family" Laura, Sabrina, and her grandparents for their unwavering love and encouragement.
I am also grateful to all the friends and colleagues I have met throughout this Master's journey, especially the group of classmates, whose support and presence made this experience truly memorable.

Finally, I am thankful to all those who, even from a distance, have always been there for me in times of need.

Padova, July 2025 *Matteo Dal Nevo*

Abstract

This thesis investigates the application of Deep Reinforcement Learning (Deep-RL) algorithms for autonomous navigation tasks in resource-constrained environments. Specifically, we focus on three state-of-the-art continuous control algorithms: Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic Policy Gradient (TD3), and Soft Actor-Critic (SAC), assessing their performance, computational requirements, and sim-to-real transferability. The experimental framework progresses from simulation to real-world deployment on a custom TurtleBot3 platform, addressing the challenges of deploying Deep-RL solutions on edge computing devices. Results demonstrate that simulation pre-training followed by real-world fine-tuning provides significant advantages in learning efficiency compared to training from scratch, and that algorithms with higher control frequencies (DDPG and TD3) can outperform slower ones (SAC) in resource-constrained settings. This is in contrast with the actual setup/choice commonly seen in the field, as SAC is the most considered algorithm due to its high performance in non-computationally constrained settings. This work involves more than 120 hours of real-world experiments and shows evidence of a possible gap between on-paper performance and real-world performance of Deep-RL algorithms due to their different computational requirements and assumptions.

Contents

Chapter 1

Introduction

Reinforcement Learning (RL) has emerged as a transformative paradigm for robotic control, fundamentally altering how autonomous systems acquire and refine their behaviors through interaction with complex environments, as illustrated in Fig. 1.1. Unlike supervised learning approaches that depend on pre-labeled datasets or traditional control methods that require explicit mathematical models, RL enables agents to autonomously learn optimal behaviors by engaging with their environment through systematic trial and error. This learning paradigm has proven particularly valuable for dynamic and uncertain domains such as robotics, where environmental conditions are often unpredictable and system dynamics may be too complex to model analytically.

The evolution from classical reinforcement learning to deep reinforcement learning (Deep-RL) represents a significant advancement in both capability and applicability. By integrating sophisticated deep neural networks to approximate complex policies and value functions, Deep-RL has transcended the limitations of traditional tabular and linear function approximation methods. This advancement enables robust performance in high-dimensional state spaces and continuous action domains that were previously intractable, opening new possibilities for autonomous robotic systems operating in real-world environments (Li, 2023).

1.1 Challenges in Real-World Deep-RL Deployment

Despite its significant successes in simulated domains—such as playing video games (Mnih et al., 2015), robotic manipulation (Levine et al., 2016), and autonomous navigation, the real-world deployment of Deep Reinforcement Learning (Deep-RL) on physical robots remains a non-trivial challenge (Wiebe, Turcato, Dalla Libera, et al., 2024; Wiebe, Turcato, Libera, et al., 2025). The transition from simulation success to real-world deployment reveals three primary categories of challenges that fundamentally impact the viability and performance of Deep-RL approaches in practical robotic applications:

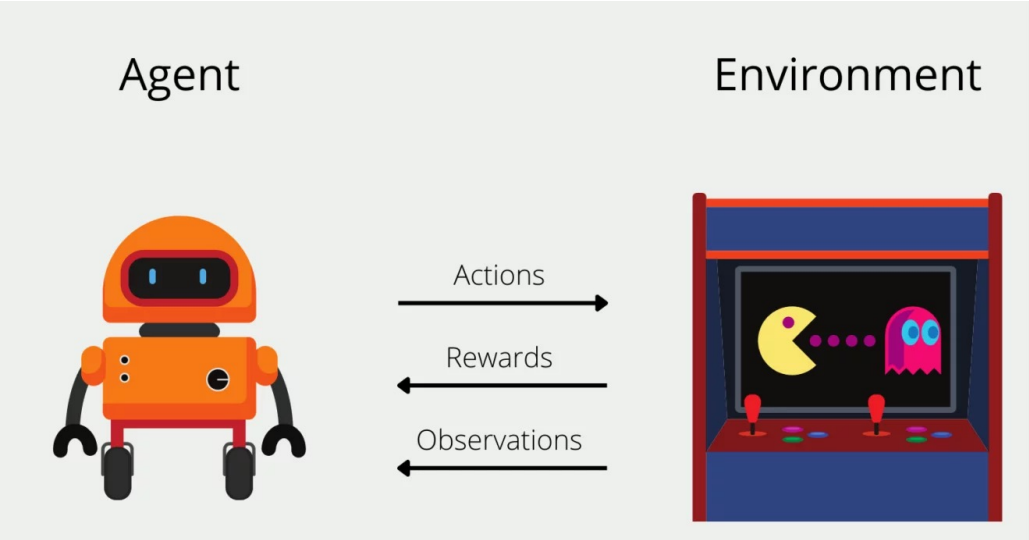

Figure 1.1: Basic interaction loop in Reinforcement Learning.

1.1.1 High Data Collection Costs and Physical Risk

Real-world training necessitates extensive interaction with the physical environment, creating substantial practical barriers that do not exist in simulation. The exploratory nature of RL algorithms require agents to attempt numerous actions, many of which will initially be suboptimal or potentially dangerous. In simulation, failed attempts carry no consequences. Indeed, a virtual robot can collide with obstacles, fall from heights, or execute erratic maneuvers without any lasting impact. However, when these same learning processes occur on physical hardware, each failed attempt carries real costs and risks.

The financial implications are substantial. Physical robots represent significant capital investments, and the trial and error nature of RL can result in component damage, wear, and the need for frequent maintenance or replacement. During the early stages of training, robots tend to explore their environment more broadly. If not properly controlled, this exploration can lead to situations where actuators are overstressed, collisions occur that may damage sensors or structural components, or the robot falls, potentially causing serious damage to the system.

Beyond direct hardware costs, the time requirements for real-world data collection are often prohibitive. Training sessions that require minutes in simulation may extend to hours or days on real hardware, factoring in the time needed for manual resets, battery changes, equipment maintenance, and safety monitoring. This temporal overhead dramatically increases the overall cost of experimentation and limits the practical feasibility of extensive hyperparameter exploration or algorithm comparison studies.

Safety considerations add another layer of complexity, particularly when robots operate in environments where humans are present or where collisions could cause damage to surrounding infrastructure. The implementation of safety mechanisms (e.g. emergency stops,

collision detection, or workspace boundaries) may interfere with the natural exploration process that RL algorithms depend upon for effective learning, potentially compromising the quality of the learned policies.

1.1.2 Computational Limitations of Edge Devices

Modern Deep-RL algorithms were primarily developed and optimized for high-performance computing environments, typically featuring powerful GPUs, multi-core processors, and abundant memory resources. These algorithms often demand high computational throughput to maintain the update frequencies and inference rates necessary for responsive performance in dynamic environments. However, practical robotic deployment typically occurs on embedded platforms such as Raspberry Pi, Jetson Nano, or custom embedded controllers that possess orders of magnitude less computational capacity than the development environments used for algorithm design.

This computational disparity creates fundamental bottlenecks that affect multiple aspects of system performance. Control loop frequency represents one of the most critical limitations. High-performance robotic control typically requires update rates ranging from 50 Hz to 1000 Hz or higher to maintain stability and responsiveness, particularly for dynamic tasks involving fast movements or precise positioning. However, the forward pass through deep neural networks used in Deep-RL policies can be computationally expensive, potentially limiting achievable control frequencies to much lower rates that may compromise system stability and performance.

The learning process itself presents additional computational challenges. Online learning requires frequent network updates, backpropagation computations, and gradient calculations that can be prohibitively expensive on resource-constrained hardware. This limitation not only affects the speed of learning but can also impact the quality of learned policies, as irregular or infrequent updates may lead to unstable training dynamics or convergence to suboptimal solutions.

Memory constraints further compound these issues. Deep neural networks, particularly those with multiple hidden layers and large numbers of parameters, require substantial memory for both model storage and intermediate computations during forward and backward passes. Embedded systems often have limited RAM and storage capacity, potentially necessitating model compression techniques or architectural modifications that may impact performance.

Power consumption represents an additional constraint that is often overlooked in laboratory settings. Many embedded platforms operate under strict power budgets, particularly for battery-powered mobile robots. The computational intensity of Deep-RL algorithms can lead to rapid battery depletion, limiting operational time and requiring careful power

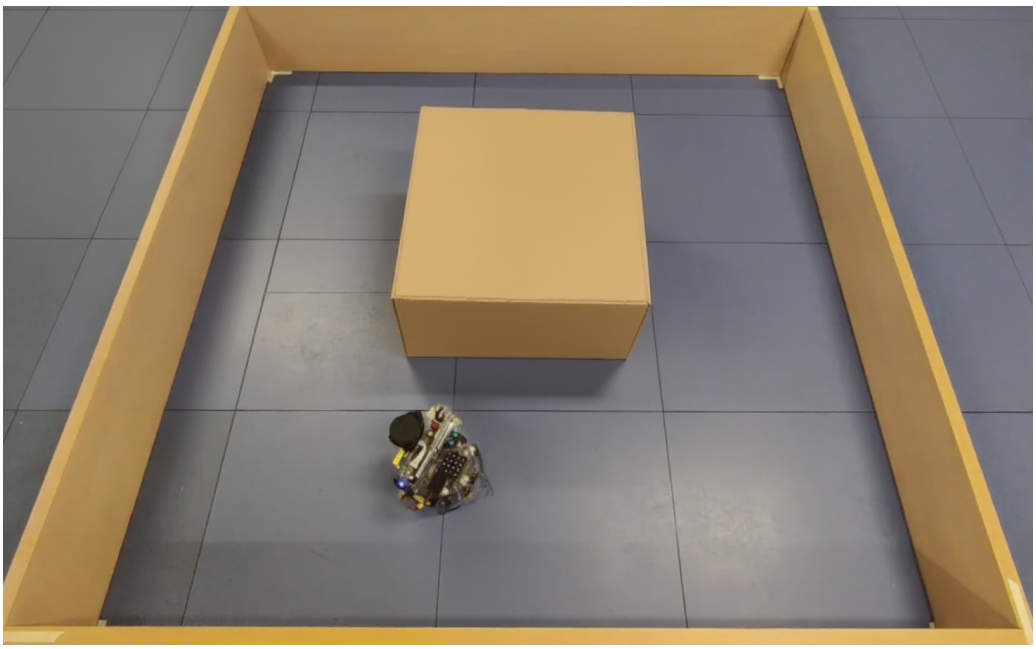

Figure 1.2: DEI TurtleBot3 in Task 3 environment.

management strategies.

1.1.3 The Sim-to-Real Transfer Gap

Perhaps the most studied and challenging aspect of real-world Deep-RL deployment is the sim-to-real transfer gap (Salvato, Fenu, and Medvet, 2021). This phenomenon occurs when policies that demonstrate excellent performance in simulation fail catastrophically when deployed on physical hardware. The gap arises from numerous sources of discrepancy between simulated and real environments, creating a complex web of modeling errors and unmodeled effects that can undermine policy performance.

Physical simulators, despite increasing sophistication, necessarily make simplifying assumptions and approximations about the complex physics of real-world systems. Fundamental physical phenomena such as friction, backlash, elasticity, air resistance, and electromagnetic effects are either approximated through simplified models or entirely omitted from most simulation environments. These simplifications can lead to significant discrepancies in system dynamics, particularly for tasks requiring precise control or exploitation of physical properties.

Sensor modeling represents another major source of sim-to-real discrepancy. Real sensors introduce noise, drift, delays, and various artifacts that are difficult to model accurately. Camera sensors may exhibit lens distortion, varying lighting sensitivity, or temporal noise characteristics that are not captured in simplified simulation models. Inertial measurement units (IMUs) and encoders may have bias, drift, or nonlinear response characteristics that significantly impact state estimation accuracy.

Actuator limitations present additional challenges that are often inadequately modeled in simulation. Real motors and actuators exhibit delays, backlash, torque limitations, and nonlinear response characteristics that can significantly affect control performance.
Simulation models typically assume ideal actuators with instantaneous response and perfect torque tracking, leading to policies that may rely on actuator capabilities that do not exist in reality.
Environmental factors represent yet another source of discrepancy. Real-world environments contain complexities such as varying lighting conditions, temperature effects, air currents, and surface irregularities that are challenging to model comprehensively. Policies that learn to exploit consistent environmental conditions in simulation may fail when faced with the variability and uncertainty of real-world environments.

1.2 Research Objectives

To address these fundamental challenges in real-world Deep-RL deployment, this thesis investigates the performance characteristics of continuous control Deep-RL algorithms under the computational constraints and practical limitations typical of embedded robotic systems (Edge Devices). This research focuses specifically on autonomous navigation for ground robots, a domain that encompasses many of the critical challenges while providing a concrete and measurable application context.
The investigation utilizes a custom-built TurtleBot3 platform equipped with realistic sensing and control interfaces, representing a typical differential drive mobile robot commonly used in research and educational settings, as illustrated in Fig. 1.2. Navigation tasks are initially developed and tested using the Gazebo simulator (Koenig and Howard, 2004), Fig. 1.3, providing a controlled environment for initial algorithm development and comparison while maintaining sufficient fidelity to support meaningful transfer to real hardware.
A central focus of this research is the systematic comparison between two distinct training paradigms that represent different approaches to the sim-to-real transfer problem. The first paradigm involves pre-training policies in simulation followed by fine tuning on real hardware, leveraging the efficiency and safety of simulation for initial learning while adapting to real-world conditions through targeted real-world experience. The second paradigm involves training policies entirely from scratch on physical hardware, potentially avoiding sim-to-real transfer issues at the cost of increased data collection requirements and safety risks.
The evaluation encompasses three state-of-the-art continuous control Deep-RL algorithms: Soft Actor-Critic, Twin Delayed Deep Deterministic Policy Gradient, and Deep Deterministic Policy Gradient. All are tested across multiple task configurations and resource constraints. These algorithms represent different approaches to the continuous control

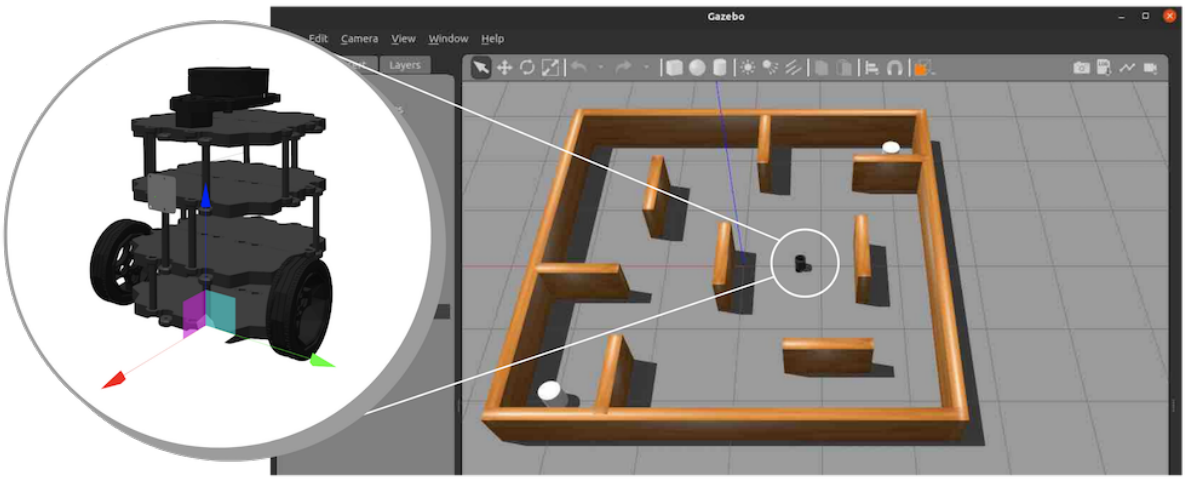

Figure 1.3: TurtleBot3 model in Gazebo - a maze simulation environment.

problem and exhibit different computational characteristics, making them ideal candidates for studying the impact of resource limitations on performance.

1.3 Literature Review

Deep-RL has significantly advanced the capabilities of autonomous agents. In robotics, it has been employed across a range of domains including manipulation, navigation, and aerial control, as illustrated in Fig. 1.4. Nevertheless, a persistent challenge remains: the reliable transfer of policies developed in simulation to real-world robotic systems.

Applications in Robotics. In the domain of robotic manipulation, Deep-RL has enabled agents to learn complex grasping and object interaction behaviors (Kober, Bagnell, and Peters, 2013). Deep-RL has enabled agile control of underactuated systems, such as the double pendulum, demonstrating robust performance in complex dynamics (Wiebe, Turcato, Libera, et al., 2025). In autonomous driving, end-to-end reinforcement learning pipelines have been proposed for lane following and high-level decision making (Kiran et al., 2022). These contributions are extensively reviewed in recent surveys and benchmarking studies (Zhang, C. Chen, Zhou, et al., 2021), which highlight both the advancements achieved and the limitations that persist.

Sim-to-Real Transfer. One of the foremost challenges in Deep-RL for robotics is the sim-to-real gap. Policies trained exclusively in simulation frequently underperform when deployed on physical systems due to discrepancies in dynamics, sensor noise, delays, and unmodeled phenomena. Several methodologies have been proposed to address this gap, including domain randomization (Tobin et al., 2017), adversarial robustness training (Pinto et al., 2017), and dynamics randomization (Peng et al., 2018). Empirical studies, such as

those by Sadeghi and Levine, 2016 and Miki et al., 2022, demonstrate that reliable sim-to-real transfer is feasible when agents are trained under sufficiently diverse and randomized conditions.

Robotic Navigation. In mobile robotics, early work demonstrated collision avoidance using Deep-RL agents trained in 2D simulated environments (Tai, Paolo, and Liu, 2017). Subsequent research extended these efforts to camera-based navigation in unstructured, real-world terrain (Kahn et al., 2018). Further progress has focused on generalization and scalability. For instance, Francis et al., 2020 proposed compact policy architectures that enabled long-range navigation in complex indoor environments, demonstrating the potential for real-world deployment.

Aerial Robotics. Within aerial robotics, Deep-RL has been used to train controllers capable of executing aggressive and agile flight maneuvers. Hwangbo et al., 2017 developed recovery strategies for quadrotors, enhancing robustness after disturbances. More recently, Kaufmann et al., 2023 reported human-level performance in drone racing tasks, underscoring the efficacy of Deep-RL in controlling highly dynamic and underactuated systems.

Computational Constraints. Despite these advancements, the high computational demands of many Deep-RL algorithms limit their practicality for real-time robotic applications. SAC (Haarnoja et al., 2018) and TD3 (Fujimoto, Van Hoof, and Meger, 2018) offer robust and stable learning but often require powerful GPUs and fast CPU cycles for efficient training and inference. Prior studies have shown that reduced update rates or communication delays can significantly degrade policy performance (Kober, Bagnell, and Peters, 2013). In response, research efforts have explored model compression, lightweight actor-critic architectures, and neural network quantization to improve computational efficiency (Kumar et al., 2021; G. Chen et al., 2020).

Algorithm Comparisons. While SAC is widely adopted for its entropy-regularized exploration strategy, resource-constrained environments may favor simpler alternatives such as TD3 or Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015).

Identified Gap and Contributions. Although considerable research has explored Deep-RL performance in simulation, relatively few studies have systematically evaluated how embedded hardware limitations influence algorithmic performance in real-world robotic deployments. This work addresses that gap by assessing Deep-RL policies across varying control and learning frequencies on a physical robot, providing practical insights and deployment guidelines for resource-constrained environments.

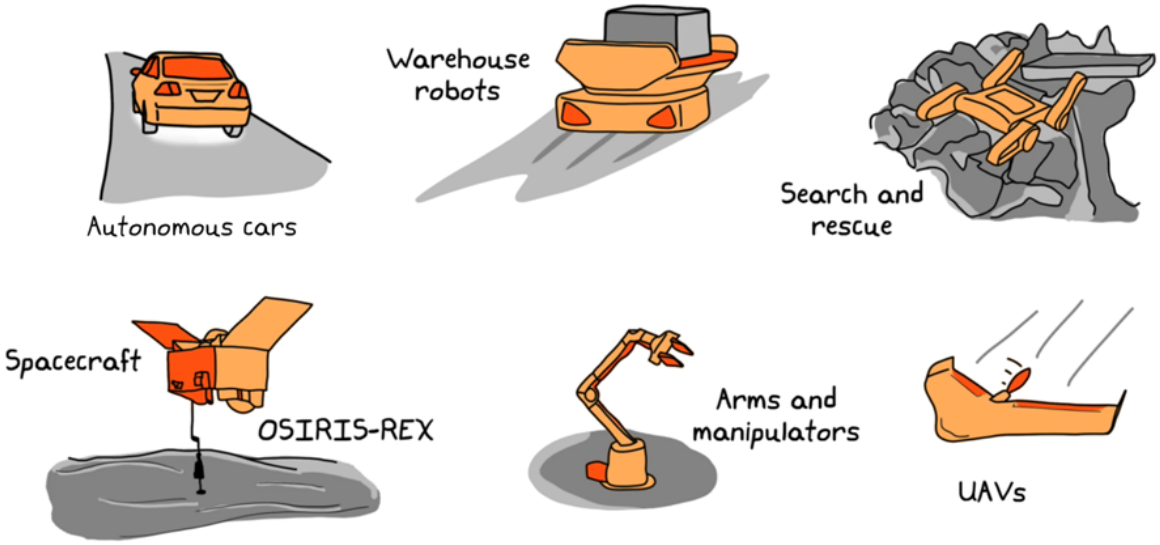

Figure 1.4: Autonomous Navigation Devices

1.4 Structure of the Thesis

The remaining chapters of this thesis are organized to provide a comprehensive progression from theoretical foundations through experimental methodology to practical results and conclusions:

- **Chapter 2** establishes the theoretical foundations essential for understanding the research presented in this thesis. This chapter introduces the mathematical framework of reinforcement learning, provides detailed descriptions of the Deep-RL algorithms under investigation, and discusses the specific characteristics of continuous control problems. Additionally, it covers the fundamentals of unmanned ground vehicles (UGVs), including their kinematic modeling based on the unicycle robot assumption.

- **Chapter 3** formulates the autonomous navigation problem as a reinforcement learning task. It presents the robot's kinematic model, defines the control interface, and describes three navigation tasks—point-to-point, path following, and corridor navigation—each with tailored state representations and reward functions to evaluate Deep-RL performance.

- **Chapter 4** outlines the experimental methodology, detailing simulation and real-world environments, hardware and ROS-based communication setup, neural network architectures optimized for embedded deployment, and the standardized training and evaluation protocols used to benchmark reinforcement learning algorithms across tasks.

- **Chapter 5** presents and analyzes the quantitative and qualitative results from evaluating the performance of Deep-RL algorithms across three navigation tasks. The chapter compares training strategies (simulation-only, real-world from scratch, and fine-tuning), discusses real-world transferability under computational constraints, and highlights trajectory behaviors that reflect each algorithm's strengths and limitations.
- **Chapter 6** concludes with final observations and directions for future work.

Chapter 2

Theoretical Background

In this chapter, a comprehensive overview of the theoretical foundations of reinforcement learning is provided, with a focus on the mathematical principles underlying deep reinforcement learning algorithms. The chapter is organized into several sections, each addressing key concepts and methodologies relevant to the field. Moreover, the chapter includes a discussion of the specific unmanned ground vehicle (UGV) model used in the experiments.

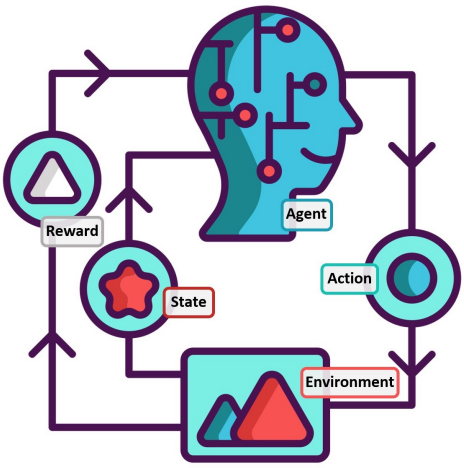

Figure 2.1: Reinforcement learning interaction.

2.1 Reinforcement Learning

Reinforcement learning is a subfield of machine learning that focuses on training agents to make sequential decisions in an environment. It is based on the idea of learning through interaction, where an agent learns to take actions in an environment to maximize a cumulative reward signal. The agent receives feedback from the environment in the form of rewards or penalties, which guide its learning process. This approach takes inspiration

from behavioral psychology, where agents learn from their experiences and adapt their behavior over time.
The RL framework is typically formalized using Markov Decision Processes (MDPs), which provide a mathematical representation of the environment and the agent's interactions with it. An MDP consists of a set of states, actions, transition probabilities, and rewards, allowing the agent to learn an optimal policy.
The key components of RL include:

- **Agent:** The entity that interacts with the environment and learns to make decisions.
- **Environment:** The external system with which the agent interacts, providing states and rewards.
- **State:** A representation of the current situation of the environment.
- **Action:** A decision made by the agent that affects the environment.
- **Reward:** A scalar feedback signal received by the agent based on action / state.
- **Policy:** A mapping from states to actions, defining the agent's behavior.
- **Value Function:** A function that estimates the expected cumulative reward from a given state or state–action pair.

The RL process involves the agent observing the current state of the environment, selecting an action based on its policy, receiving a reward and a new state from the environment, and updating its policy based on this experience. The goal is to learn a policy that maximizes the expected cumulative reward over time.

2.1.1 Markov Decision Processes

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment.
These interact continually: the agent selects actions and the environment responds to these actions, presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

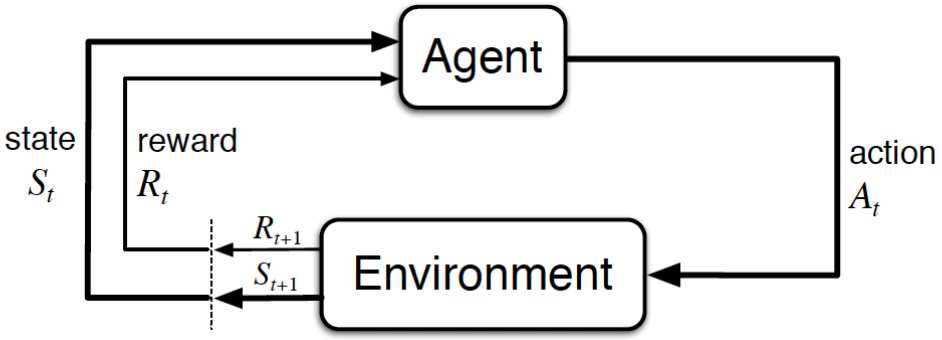

Figure 2.2: The agent–environment interaction in a Markov decision process.

Formally, a Markov Decision Process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ is the set of states. At each timestep t, the agent observes a state $S_t \in \mathcal{S}$.
- $\mathcal{A}$ is the set of actions. When in state S_t, the agent selects an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t) \subseteq \mathcal{A}$ denotes the actions available in S_t.
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function. For any s, a, s',

$$P(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a). \tag{2.1}$$

- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function. Taking action $A_t = a$ in state $S_t = s$ and transitioning to s' yields a scalar reward

$$R_{t+1} = R(s, a, s'). \tag{2.2}$$

- $\gamma \in [0, 1]$ is the discount factor, which trades off immediate and future rewards.

The agent's goal is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expected discounted return

$$G_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1}. \tag{2.3}$$

The value function $V^\pi(s)$ and action-value function $Q^\pi(s, a)$ under policy π are:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s], \tag{2.4}$$

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.5}$$

Optimality is defined by:

$$V^*(s) = \max_\pi V^\pi(s), \tag{2.6}$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a). \tag{2.7}$$

The Bellman optimality equations provide recursive definitions:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s') \right], \tag{2.8}$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]. \tag{2.9}$$

An optimal policy π^* can be obtained by:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a). \tag{2.10}$$

2.1.2 Value-based Methods

Almost all reinforcement learning algorithms involve estimating value functions (i.e. functions of states or state–action pairs that quantify how beneficial it is for an agent to be in a particular state or to take a particular action from that state). These estimations are based on the expected cumulative future rewards, defined formally as the *expected return*. Since future rewards depend on the actions the agent will take, value functions are always defined with respect to a policy π.
The *state-value function* $V^\pi(s)$ gives the expected return when starting from state s and following policy π thereafter:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s \right]. \tag{2.11}$$

Similarly, the *action-value function* $Q^\pi(s,a)$ gives the expected return when starting from state s, taking action a, and thereafter following policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s, A_t = a \right]. \tag{2.12}$$

Value-based methods aim to estimate these functions as accurately as possible and use them to derive improved policies. The agent typically follows an ϵ-greedy policy derived from the action-value function, where the best-known action is chosen most of the time, but random actions are occasionally selected to encourage exploration.

Learning Value Functions There are several methods to estimate value functions:

- **Monte Carlo (MC) methods:** Estimate value functions by averaging actual returns observed in complete episodes. These are simple and unbiased but require episodic tasks and complete trajectories.

- **Temporal-Difference (TD) learning:** Combines ideas from Monte Carlo and dynamic programming. TD methods (e.g., TD(0), SARSA) update estimates based on bootstrapping, using existing estimates to refine value predictions before episodes complete.

Among value-based methods, **Q-learning** is one of the most prominent. It is an off-policy algorithm that learns the optimal action-value function $Q^*(s, a)$ using the Bellman optimality equation. The Q-values are iteratively updated as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{2.13}$$

Limitations Despite their strengths, value-based methods face several challenges:

- They can struggle with continuous or very large action spaces, where action-value enumeration is impractical.
- Value approximation can be unstable, especially when combined with function approximators like neural networks.
- Exploration can be inefficient in sparse-reward environments.

These challenges motivate the development of alternative or hybrid approaches, such as policy-based and actor-critic methods, which are more suitable for certain classes of problems.

2.1.3 Policy-based Methods

Policy-based methods are a class of reinforcement learning algorithms that directly parameterize the policy $\pi_\theta(a|s)$ and optimize its parameters θ through gradient ascent to maximize the expected return. Unlike value-based approaches, which estimate the value of actions or state-action pairs to derive a policy, policy-based methods focus on learning the policy itself, making them especially well-suited for environments with high-dimensional or continuous action spaces.
The objective function to be maximized is typically defined as the expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t], \tag{2.14}$$

where G_t is the cumulative reward starting at time step t. The policy gradient theorem provides a way to compute the gradient of this objective with respect to the policy parameters:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s, a) \right]. \tag{2.15}$$

This gradient can be used to perform the parameter update:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \tag{2.16}$$

where α is the learning rate. A common choice for the objective function J is the state-value function $V^\pi(s)$, as maximizing it leads to improved expected returns from a given state.
One challenge with policy gradient methods is the high variance of the gradient estimates, which can hinder learning stability and slow convergence. To address this, techniques such as baseline subtraction (e.g., using the value function as a baseline), entropy regularization, and trust region constraints have been developed.
Despite this limitation, policy-based methods offer several advantages:

- They naturally support stochastic policies, allowing effective exploration.
- They are applicable to both discrete and continuous action spaces.
- They can model multi-modal or complex policy distributions more flexibly than value-based approaches.

These characteristics make policy gradient methods particularly effective in domains like robotics, autonomous systems, and complex decision-making environments. Notable algorithms include *REINFORCE*, a Monte Carlo policy gradient algorithm, and more advanced variants that incorporate variance reduction and value function estimation, such as *Actor-Critic* methods.
It is important to note that, although the policy gradient theorem provides a theoretical foundation, practical application often requires approximation techniques, as it involves expectations over trajectories and the stationary state distribution $\mu(s)$, which is generally unknown. Theoretical formulations thus serve as a basis from which practical algorithms are derived, as discussed in subsequent sections.

2.1.4 Actor-Critic Methods

Actor-critic methods are a class of reinforcement learning algorithms that unify the strengths of value-based and policy-based approaches. These methods consist of two components: the *actor*, which is responsible for selecting actions based on a parameterized policy π_θ, and the *critic*, which evaluates the current policy by estimating value functions.
The actor interacts directly with the environment and aims to learn an optimal policy that maximizes expected cumulative rewards. This policy is typically represented by a neural network and is updated through gradient ascent. The critic, on the other hand, provides

feedback by estimating either the state-value function $V_\phi(S)$ or the action-value function, guiding the actor toward more effective behavior. The critic's estimate is updated using temporal-difference (TD) learning, which relies on the TD error:

$$\delta_t = R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t). \tag{2.17}$$

The updates for the critic and actor parameters are given by:

$$\phi \leftarrow \phi + \alpha_c \delta_t \nabla_\phi V_\phi(S_t), \tag{2.18}$$

$$\theta \leftarrow \theta + \alpha_a \delta_t \nabla_\theta \log \pi_\theta(A_t|S_t). \tag{2.19}$$

This coordinated learning allows the actor to improve the policy using feedback from the critic, while the critic's estimates become more accurate as the actor explores the environment. Despite their effectiveness, actor-critic methods face several practical challenges:

- **High Variance in Policy Updates:** Since the actor updates its policy based on the critic's value estimates, noise or inaccuracies in these estimates can introduce significant variance and lead to instability.
- **Bias in Value Function Approximation:** The critic's function approximator (e.g., a neural network) can introduce bias, especially if the architecture or training procedure is suboptimal.
- **Exploration vs. Exploitation:** Balancing the need to explore unseen actions and states while exploiting known rewarding behavior remains a complex aspect of actor-critic strategies.
- **Hyperparameter Sensitivity:** The performance of actor-critic algorithms can depend heavily on choices like learning rates and discount factors, often requiring extensive tuning.
- **Non-Stationarity:** As both the actor and critic are updated simultaneously, the underlying data distribution changes continuously, which can slow convergence.
- **Sample Inefficiency:** These methods may require substantial amounts of interaction data, making them computationally expensive in complex environments.

To improve stability and reduce variance in updates, actor-critic algorithms are often extended. Notable variants include Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C), and Proximal Policy Optimization (PPO). These improvements typically involve techniques such as using advantage functions, parallel environments, or clipped objective functions to stabilize learning and enhance performance.

2.2 Deep Reinforcement Learning

Deep reinforcement learning combines the representational power of deep learning with the decision-making framework of reinforcement learning, enabling agents to learn intelligent behaviors in complex and high-dimensional environments. This integration has led to groundbreaking achievements in areas such as game playing, robotics, and natural language processing, where Deep RL agents have, in some cases, surpassed human-level performance. Deep learning, a subfield of machine learning, employs deep neural networks to learn complex patterns and features from high-dimensional data. By leveraging deep learning techniques, Deep RL is able to overcome the limitations of traditional RL methods, particularly in environments with unstructured or raw input spaces such as images or sensor data.

It achieves this by using neural networks as function approximators for key RL components, such as value functions, policy functions, or action-value (Q) functions. This eliminates the need for manual feature engineering, allowing agents to learn directly from raw inputs and adapt to highly complex scenarios.

One of the most significant milestones in Deep RL is the development of Deep Q-Networks (DQN) Mnih et al., 2015, which introduced the use of convolutional neural networks to approximate the action-value function. DQN demonstrated impressive results on the Atari 2600 suite of games, where a single agent learned to play multiple games directly from pixel input and reward signals. Since then, many extensions and improvements have been proposed, including double DQN, dueling networks, and distributional RL.

These capabilities are essential for developing autonomous systems that can interact with the real world in a robust and adaptive manner.

Despite its success, Deep RL also presents several challenges:

- **Exploration in High Dimensions:** In environments with high-dimensional or continuous action spaces, effective exploration becomes difficult, often requiring sophisticated strategies.
- **Sample Inefficiency:** Training Deep RL agents can require vast amounts of interaction data, making learning costly and time-consuming.
- **Instability and Hyperparameter Sensitivity:** Deep neural networks are sensitive to training dynamics and require careful tuning of learning rates, architectures, and other hyperparameters.
- **Computational Demands:** The training process is typically resource-intensive, often necessitating specialized hardware (e.g., GPUs or TPUs).

To address these challenges, various enhancements to Deep Reinforcement Learning have been proposed. In particular, actor-critic algorithms have gained prominence for their

ability to handle continuous action spaces and sample inefficiency.
The following sections focus on three representative actor-critic algorithms that have been widely adopted in robotic control.

2.2.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient is an off-policy, actor-critic algorithm designed for environments with continuous action spaces. This model-free method builds upon the deterministic policy gradient (DPG) algorithm and leverages deep neural networks to learn policies in high-dimensional, continuous domains.
DDPG combines the actor-critic framework with key ideas from the Deep Q-Network (DQN), particularly the use of experience replay and target networks. These elements contribute to a more stable and robust training process. Off-policy learning allows the agent to reuse past experiences stored in a replay buffer, while target networks provide consistent targets for the temporal difference updates, reducing the risk of divergence during learning.
In summary, DDPG is characterized by three essential components. Its relatively straightforward architecture makes it easy to implement:

- An actor network $\mu(s|\theta^\mu)$ that maps states to deterministic actions.
- A critic network $Q(s, a|\theta^Q)$ that estimates the value of state-action pairs.
- Target networks $\theta^{\mu'}$ and $\theta^{Q'}$ that are slowly updated versions of the actor and critic networks, used for stable target computation.

The critic is trained by minimizing:

$$L(\theta^Q) = \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma Q(s', \mu(s'|\theta^{\mu'})|\theta^{Q'}) - Q(s, a|\theta^Q) \right)^2 \right]. \tag{2.20}$$

The actor is updated via the deterministic policy gradient:

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_s \left[\nabla_a Q(s, a|\theta^Q)\big|_{a=\mu(s)} \nabla_{\theta^\mu} \mu(s|\theta^\mu) \right]. \tag{2.21}$$

To encourage exploration with a deterministic policy, DDPG adds temporally correlated noise to the actor's output, sample from a noise process $\mathcal{N}_t$ that can be chosen to suit the environment:

$$a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t \tag{2.22}$$

Transitions (s_t, a_t, r_t, s_{t+1}) are stored in a replay buffer $\mathcal{B}$; at each update step we sample a mini-batch of size N uniformly from $\mathcal{D}$ to compute both the critic loss and the policy gradient, thereby breaking correlations between consecutive samples.

For stability, the critic and actor targets use slowly updated "soft" target networks:

$$\theta^{Q'} \leftarrow \tau\,\theta^{Q} + (1-\tau)\,\theta^{Q'}, \quad \theta^{\mu'} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\,\theta^{\mu'}, \tag{2.23}$$

with a small mixing factor $\tau \ll 1$ (e.g. 10^{-3}).

Algorithm 1 DDPG algorithm

Initialize: critic $Q(s, a \mid \theta^Q)$ and actor $\mu(s \mid \theta^\mu)$ with weights θ^Q, θ^μ
Initialize target networks Q' and μ' with $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $\mathcal{B}$
for episode $= 1 \rightarrow M$ **do**
    Initialize exploration noise process $\mathcal{N}$
    Receive initial state s_1
    for $t = 1 \rightarrow T$ **do**
        Select action $a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t$ according to the current policy and noise
        Execute action a_t, observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$
        Sample mini-batch of N transitions $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{N}$ from $\mathcal{B}$
        Compute targets $y_i = r_i + \gamma\, Q'\big(s_{i+1},\, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'}\big)$
        Update critic by minimizing $L = \frac{1}{N}\sum_i \big(y_i - Q(s_i, a_i \mid \theta^Q)\big)^2$
        Update actor with sampled policy gradient

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a \mid \theta^Q)\big|_{s_i, a=\mu(s_i)}\, \nabla_{\theta^\mu}\mu(s_i \mid \theta^\mu)$$

        Soft-update targets:

$$\theta^{Q'} \leftarrow \tau\,\theta^{Q} + (1-\tau)\,\theta^{Q'}, \quad \theta^{\mu'} \leftarrow \tau\,\theta^{\mu} + (1-\tau)\,\theta^{\mu'}$$

    end for
end for

2.2.2 Twin Delayed DDPG (TD3)

In value-based reinforcement learning methods such as Deep Q-Learning, the use of function approximation can lead to *overestimation bias*, where the Q-values systematically overestimate the true expected returns. This phenomenon also affects actor-critic methods like DDPG, where the critic estimates action values and guides the policy improvement. Due to the coupling between the actor and the critic, overestimated Q-values can mislead the policy to favor suboptimal actions, leading to degraded performance or instability in training.

Overestimation Bias in Actor-Critic Methods

The overestimation problem arises because the max operator in the target value calculation tends to select overestimated values when the critic is imperfect. Specifically, when computing the target in TD learning, such as:

$$y = r + \gamma Q(s', \mu(s')),$$

the policy μ may select an action that appears optimal under the critic's current estimate, but due to approximation noise or error, this action's Q-value may be inaccurately high. Over multiple updates, this can accumulate and bias the critic toward overestimated returns, destabilizing both the value function and the policy.

TD3 is a variant of DDPG specifically designed to address this overestimation issue and improve stability in continuous control tasks. It introduces three key improvements:

- **Clipped Double Q-Learning:** TD3 maintains two independent critic networks, Q_{θ_1} and Q_{θ_2}. When computing the target value, it takes the minimum of the two estimates:

  $$y = r + \gamma \min(Q_{\theta_1}(s', \tilde{a}), Q_{\theta_2}(s', \tilde{a})),$$

  which acts as a form of conservative estimation to reduce the overestimation bias.

- **Delayed Policy Updates:** The actor and target networks are updated less frequently than the critics (e.g., once every d steps). This prevents the actor from exploiting inaccuracies in the critic too early, which improves learning stability.

- **Target Policy Smoothing:** To make the critic less sensitive to slight errors in action selection, noise is added to the target action during critic updates:

  $$\tilde{a} = \mu_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c).$$

  This technique encourages the critic to learn a value function that is robust to small variations in the action, similar to regularization.

Together, these modifications help TD3 mitigate overestimation and improve sample efficiency and policy stability, particularly in environments with high-dimensional continuous action spaces.

Algorithm 2 TD3

Initialize: critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network π_ϕ with random params θ_1, θ_2, ϕ
Initialize target networks $\theta'_1 \leftarrow \theta_1,\ \theta'_2 \leftarrow \theta_2,\ \phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1 \rightarrow T$ **do**
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma)$
    Observe reward r and new state s'
    Store transition tuple (s, a, r, s') in $\mathcal{B}$
    Sample mini-batch of N transitions $\{(s_i, a_i, r_i, s'_i)\}$ from $\mathcal{B}$
    Compute perturbed target action $\tilde{a}_i = \pi_{\phi'}(s'_i) + \tilde{\epsilon}, \quad \tilde{\epsilon} \sim \text{clip}\big(\mathcal{N}(0, \tilde{\sigma}), -c, c\big)$
    Compute targets $y = r + \gamma\ \min_{i=1,2} Q_{\theta'_i}\big(s', \tilde{a}\big)$
    Update each critic by

$$\theta_i \ \leftarrow\ \arg\min_{\theta_i}\ \frac{1}{N} \sum \big(y - Q_{\theta_i}(s, a)\big)^2 \quad (i = 1, 2)$$

    if t mod d **then**
        Update ϕ by the deterministic policy gradient:

$$\nabla_\phi J(\phi) = \frac{1}{N} \sum \nabla_a Q_{\theta_1}(s, a)\big|_{a=\pi_\phi(s)}\ \nabla_\phi\, \pi_\phi(s)$$

        Soft-update target networks:

$$\theta'_i \ \leftarrow\ \tau\, \theta_i + (1 - \tau)\, \theta'_i, \quad \phi' \ \leftarrow\ \tau\, \phi + (1 - \tau)\, \phi' \quad (i = 1, 2)$$

    end if
end for

2.2.3 Soft Actor-Critic (SAC)

Soft Actor-Critic is an off-policy actor-critic algorithm that optimizes a stochastic policy in continuous action spaces, based on the *maximum entropy reinforcement learning* framework. Unlike standard RL approaches that maximize expected return, SAC encourages exploration by maximizing a trade-off between return and policy entropy.
This leads to the following objective:

$$J(\pi) = \sum_t \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)) \right], \tag{2.24}$$

where $\mathcal{H}(\pi(\cdot|s_t)) = -\mathbb{E}_{a_t \sim \pi}[\log \pi(a_t|s_t)]$ denotes the entropy of the policy, and α is the temperature parameter that balances exploration (entropy) and exploitation (reward max-

imization).
This entropy-augmented objective encourages the policy to remain stochastic during training, leading to more robust and exploratory behavior.

Network Architecture and Objectives

SAC simultaneously learns three function approximators:

- A **soft Q-function** $Q(s, a|\theta^Q)$ to estimate expected return.
- A **value function** $V(s|\theta^V)$ to estimate the state value under the current policy.
- A **stochastic policy** network $\pi(a|s|\theta^\pi)$ to sample actions from a learned distribution (typically Gaussian).

The soft Q-function is trained to minimize the Bellman residual:

$$J_Q(\theta^Q) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(Q(s,a) - (r + \gamma V(s'))\right)^2\right]. \tag{2.25}$$

The value network is updated by minimizing:

$$J_V(\theta^V) = \mathbb{E}_{s\sim\mathcal{D}}\left[\left(V(s) - \mathbb{E}_{a\sim\pi}\left[Q(s,a) - \alpha \log \pi(a|s)\right]\right)^2\right]. \tag{2.26}$$

The policy is trained to maximize the expected Q-value while also maximizing entropy:

$$J_\pi(\theta^\pi) = \mathbb{E}_{s\sim\mathcal{D}, a\sim\pi}\left[\alpha \log \pi(a|s) - Q(s,a)\right]. \tag{2.27}$$

Automatic Entropy Adjustment

One of SAC's key innovations is the **automatic tuning of the entropy coefficient** α, allowing the agent to balance exploration and exploitation adaptively. The objective for α is to match a target entropy $\mathcal{H}_{\text{target}}$:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi}\left[-\alpha\left(\log \pi(a_t|s_t) + \mathcal{H}_{\text{target}}\right)\right]. \tag{2.28}$$

This adjustment stabilizes learning and removes the need for manually setting α.
In summary, SAC combines the strengths of off-policy learning, stochastic policies, and entropy regularization into a highly effective and robust deep reinforcement learning algorithm for continuous control.

Algorithm 3 Soft Actor-Critic

Initialize parameter vectors ψ, $\bar{\psi}$, θ, ϕ
for each iteration **do**
  for each environment step **do**
    $a_t \sim \pi_\phi(a_t|s_t)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$
    $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i)$ ▷ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J_\pi(\phi)$
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
  end for
end for

2.3 Unmanned Ground Vehicle

This section introduces the Unmanned Ground Vehicle model used in this work. The platform is a differential-drive mobile robot, a common design in robotics research due to its simplicity, ease of control, and suitability for planar motion tasks. Differential-drive robots serve as a canonical testbed for reinforcement learning-based navigation and control algorithms.

2.3.1 Differential Drive Robots

A differential-drive robot consists of two wheels mounted on the same axis, each driven independently. Steering is achieved by varying the relative speeds of the wheels. The kinematic model of such a system is well-approximated by the *unicycle model*, Fig. 2.3, which captures the essential motion characteristics under ideal (non-slipping) conditions. The continuous-time kinematic equations are given by:

$$\dot{x} = v\cos(\theta), \tag{2.29}$$

$$\dot{y} = v\sin(\theta), \tag{2.30}$$

$$\dot{\theta} = \omega, \tag{2.31}$$

where (x, y) denotes the robot's position in the plane, θ its orientation, v the linear velocity, and ω the angular velocity. For a differential-drive system, these velocities are

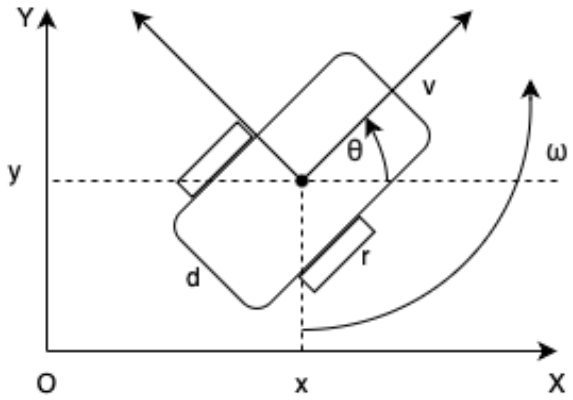

Figure 2.3: Differential Drive Robot scheme.

computed from the wheel velocities as:

$$v = \frac{v_r + v_l}{2}, \tag{2.32}$$

$$\omega = \frac{v_r - v_l}{b}, \tag{2.33}$$

where v_r and v_l are the right and left wheel velocities respectively, and b is the wheel-base—the distance between the two wheels.

Modeling and Simulation

In practical applications and simulations, additional non-idealities such as wheel slip, motor saturation, actuator delays, and friction are considered. These dynamic effects are incorporated to better replicate the real-world behavior of the UGV and to provide more informative state transitions and rewards during reinforcement learning training.
The differential-drive UGV model serves as the environment for training Deep-RL algorithms. The agent receives observations such as position, orientation, and velocity, and outputs control actions that are mapped to wheel speeds. This model supports the study of goal-directed navigation, obstacle avoidance, and path-following tasks under partial observability and noisy dynamics.

Chapter 3

Problem Formulation

In this chapter, the autonomous navigation problem is formally cast as a reinforcement learning task. The objective is to train a robot to navigate through diverse and potentially cluttered environments by learning optimal behaviors through trial and error. Specifically, the robot must avoid obstacles and reach specific targets or follow designated paths using only sensory observations and learned control policies.
To this end, we define the mathematical modeling of the robotic platform, the formulation of the RL problem, and the design of the training environments and reward functions. These components collectively guide the learning process and serve as a testbed for evaluating different Deep-RL algorithms.

3.1 Robot Model

As described in Chapter 2, the robot is modeled using a unicycle kinematic model, which captures the essential motion characteristics of differential-drive robots:

$$\dot{\mathbf{x}}_t = f(\mathbf{x}_t, \mathbf{u}_t) = \begin{bmatrix} \dot{x}_t \\ \dot{y}_t \\ \dot{\theta}_t \end{bmatrix} = \begin{bmatrix} v_t \cos(\theta_t) \\ v_t \sin(\theta_t) \\ \omega_t \end{bmatrix}, \tag{3.1}$$

where $\mathbf{x}_t = [x_t, y_t, \theta_t]^T$ denotes the robot's 2D pose, and $\mathbf{u}_t = [v_t, \omega_t]^T$ represents the linear and angular velocity control inputs.
To map the policy output to physical commands, an affine transformation is applied to the action vector $\mathbf{a}_t \in [-1, 1]^2$, resulting in:

$$\mathbf{u}_t = \mathbf{B}(\mathbf{M}\,\mathbf{a}_t + \mathbf{d}) = \begin{bmatrix} v_M & 0 \\ 0 & \omega_M \end{bmatrix} \Big(\begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{a}_t + \begin{bmatrix} 1/2 \\ 0 \end{bmatrix} \Big), \tag{3.2}$$

where $\mathbf{B}$ scales the velocities to the robot's maximum linear v_M and angular ω_M speed

limits. $\mathbf{M}$ and $\mathbf{d}$ define a linear-affine mapping from normalized action space to control space.
This mapping ensures that the robot is constrained to forward motion, aligning with real-world robotic constraints and improving safety in navigation.

3.2 Tasks Description

To evaluate the performance and generalization of Deep-RL agents, three distinct navigation tasks are designed. Each task targets a specific skill, uses a tailored reward function, and defines custom state/action representations.

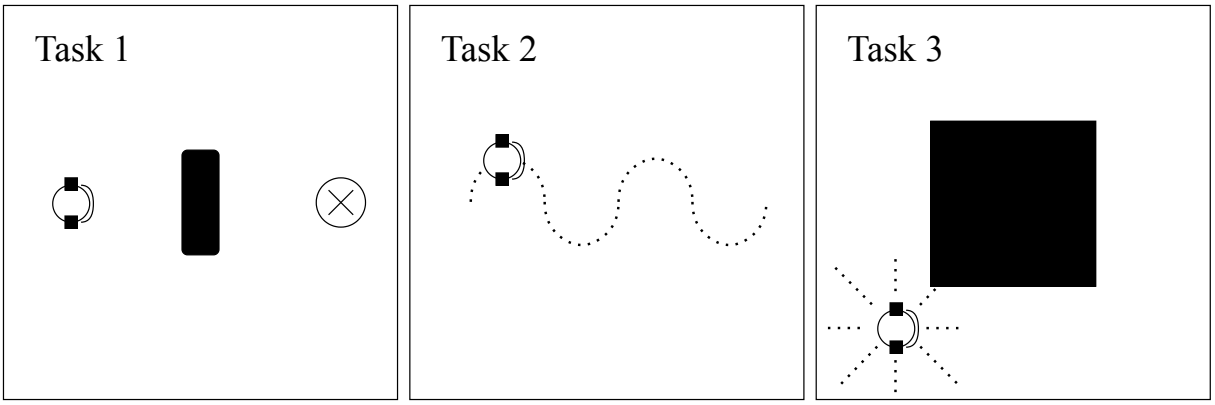

Figure 3.1: Schematics overview of the three tasks.

3.2.1 Task 1: Point-to-Point Navigation

In this task, the robot must navigate from a random initial pose to a fixed goal location while avoiding a central obstacle. The initial pose is sampled from a circular region centered at $[-1, 0]^T$, and is formally defined as:

$$\mathbf{x}_0 \in \left\{ \begin{bmatrix} -1 + \rho\cos(\beta) \\ \rho\sin(\beta) \\ \xi \end{bmatrix} \middle| \rho \in [0, 0.1],\ \beta \in [0, 2\pi],\ \xi \in [-0.01, 0.01] \right\} \tag{3.3}$$

The fixed goal location is a circular area centered at $[x_g, y_g]^T = [1, 0]^T$ with a radius of acceptance of 0.15 meters. A rectangular obstacle of dimensions 0.2×0.5 meters is placed at the origin, as illustrated in Fig. 3.1 (left).
The state space at time t is:

$$\mathbf{s}_t = \begin{bmatrix} d_{\text{goal}} & e_\theta & v_\parallel & v_\perp & \omega_t & d_{\text{obs}} \end{bmatrix}^T, \tag{3.4}$$

where d_{goal} is the robot's distance to the goal, $e_\theta = \theta_t - atan2(y_g - y_t, x_g - x_t)$ is the heading error, $v_\parallel = v_t \cos(e_\theta)$ is the velocity parallel to the goal direction, $v_\perp = v_t \sin(e_\theta)$ is the velocity perpendicular to the goal direction, ω_t is the angular velocity, and d_{obs} is the distance to the obstacle.
The reward function $r(s_t, a_t, s_{t+1})$ is designed to encourage efficient navigation to the target while avoiding the obstacle:

$$r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) = \begin{cases} +100, & \text{if target reached} \\ -25, & \text{if collision occurs} \\ -50, & \text{if } (x_t, y_t) \notin [-1.2, 1.2]^2 \\ +2, & \text{if } \Delta_d > 0.01 \text{ m} \\ -1, & \text{otherwise} \end{cases}, \tag{3.5}$$

where Δ_d is the difference between the current and previous distances to the target. The intermediate reward encourages the robot to move toward the target. The episodes end when: (i) the robot enters the obstacle area, (ii) it goes outside the position limits $[-1.2, 1.2]^2$, (iii) the target is reached, that is, $d_{\text{goal}} < 0.15$. For this task, the tracked performance metric is the success rate, measured as the target reaching rate, over 5 episodes.

3.2.2 Task 2: Path Following

At the beginning of the task, the agent is placed near the point $[-1, 0]^T$, with a small variation in its orientation. Specifically, its initial pose is randomly sampled from the set:

$$\mathbf{x}_0 \in \left\{ \begin{bmatrix} -1 \\ 0 \\ \arctan(\pi) + \xi \end{bmatrix} \middle| \, \xi \in [-0.1, 0.1] \right\},$$

where ξ introduces a slight angular perturbation to simulate uncertainty in the heading direction.
From this starting point, the agent is tasked with following a smooth, predefined path that guides it from the left to the right side of the environment. As illustrated in Fig. 3.1 (center), the reference trajectory traces a sine wave, encouraging the agent to perform continuous and coordinated motion. The path is mathematically described by:

$$y_{\text{ref}} = 0.5 \sin(2\pi x_t), \quad \text{with } x \in [-1, 1], \tag{3.6}$$

where x_t denotes the agent's position along the horizontal axis at time t. This trajec-

tory provides a dynamic challenge that requires the agent to adapt its heading while maintaining smooth lateral movements.

At each time step t, the robot's state is represented by the vector:

$$\mathbf{s}_t = \begin{bmatrix} x_t & e_y & e_\theta & \omega_t \end{bmatrix}^T, \tag{3.7}$$

where x_t denotes the robot's horizontal position along the x-axis, $e_y = y_{\text{ref}} - y_t$ represents the lateral tracking error with respect to the reference trajectory, and $e_\theta = \pi \cos(2\pi x_t) - \theta_t$ captures the heading error. The term ω_t denotes the robot's angular velocity at time t.

In this setup, the robot moves with a constant linear velocity, and its control is limited to angular adjustments through the action space $a_t \in [-1, 1]$. As a result, the control input is defined as:

$$\mathbf{u}_t = \begin{bmatrix} v_M \\ \omega_M \cdot a_t \end{bmatrix},$$

where v_M is the fixed linear speed and ω_M is a scaling factor for the angular velocity.

The reward function is designed to encourage forward progression while penalizing deviations from the reference path and excessive rotational motion. It is given by:

$$r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) = \begin{cases} x_t + 1 - |e_y| - 2\,|\omega_t|, & \text{if } d_{\min} \leq 0.2 \\ -2, & \text{otherwise} \end{cases} \tag{3.8}$$

Here, $d_{\min}$ represents the minimum Euclidean distance between the robot's current position and the reference path. When this distance exceeds 0.2 meters, a penalty is applied to discourage the robot from straying too far.

Each episode terminates either when the robot deviates significantly from the path ($d_{\min} > 0.2$), or when it successfully completes the trajectory—defined as reaching a position where $x_t > 0.95$ and $|y_t| < 0.1$.

The performance in this task is evaluated based on the average distance traveled along the x-axis across five episodes, reflecting how effectively the robot follows the desired trajectory.

3.2.3 Task 3: Corridor Navigation

In this task, the robot operates within a confined, rectangular ring-shaped environment, as illustrated in Fig. 3.1 (right). The agent's initial pose is randomly selected within the corridor space, oriented parallel to the nearby closed wall. The objective is to continuously move forward without collisions, requiring the agent to learn a policy that maximizes the total distance traveled while maintaining safe navigation and obstacle avoidance.

The robot's state at time t is composed of a downsampled vector of laser scan measure-

ments combined with its current linear and angular velocities:

$$\mathbf{s}_t = \begin{bmatrix} d_1 & d_2 & \dots & d_n & v_t & \omega_t \end{bmatrix}^T, \tag{3.9}$$

where the d_i are laser range readings evenly sampled from the sensor, with $n = 16$. The minimum laser measurement,

$$d_{\min} = \min\{d_1, \dots, d_n\},$$

is employed both for collision detection and for shaping the reward function.
To encourage fast and smooth motion while ensuring safety, the reward function is defined as:

$$r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) = \begin{cases} 3v_t - \left|\frac{\omega_t}{2}\right| - \frac{1}{2}(1 - d_{\min}), & \text{if } d_{\min} \geq 0.2 \\ -5, & \text{otherwise} \end{cases} \tag{3.10}$$

This formulation rewards higher forward velocity, penalizes excessive angular velocity, and discourages proximity to obstacles by incorporating the minimum laser distance.
Episodes terminate either when the robot gets too close to an obstacle ($d_{\min} < 0.2$) or when a maximum time limit of 17 seconds is reached. The primary performance metric tracked in this task is the total distance traveled, calculated by integrating the robot's linear velocity over 5 episodes.

Chapter 4

Experimental Setup and Methodology

4.1 Simulation Environments

This section presents the simulation environments used for training and evaluating the proposed reinforcement learning algorithms. A two-stage approach was adopted, beginning with a simplified 2D environment for initial development and validation, followed by a more realistic 3D simulation for comprehensive testing and performance evaluation. In addition, a detailed description of the hardware setup and communication protocols is provided, along with the architecture employed for both pre-training and real-world training. Finally, the training and evaluation protocols are outlined to ensure reproducibility and clarity in the methodology.

4.1.1 2D Gym Environment

In the first stage of development, a simplified 2D simulation environment was implemented using the OpenAI Gym framework, which provides a standardized interface for reinforcement learning tasks. This environment represents the robot as a point mass following the differential drive kinematics model described in the previous chapter.
The primary purpose of this phase was to utilize a simplified environment to concentrate on several key objectives during the initial development stage. The approach was designed to enable rapid prototyping and early-stage algorithm development, providing a foundation for iterative refinement of core algorithms without the complexity of full-scale simulation. Additionally, this phase focused on designing and iteratively refining the reward function using reward shaping techniques, allowing for systematic exploration of different reward formulations and their impact on learning performance. The establishment of baseline performance metrics for navigation tasks was another crucial objective, providing quantitative benchmarks against which future developments could be measured. Finally, this simplified environment served to validate the core approach prior to tran-

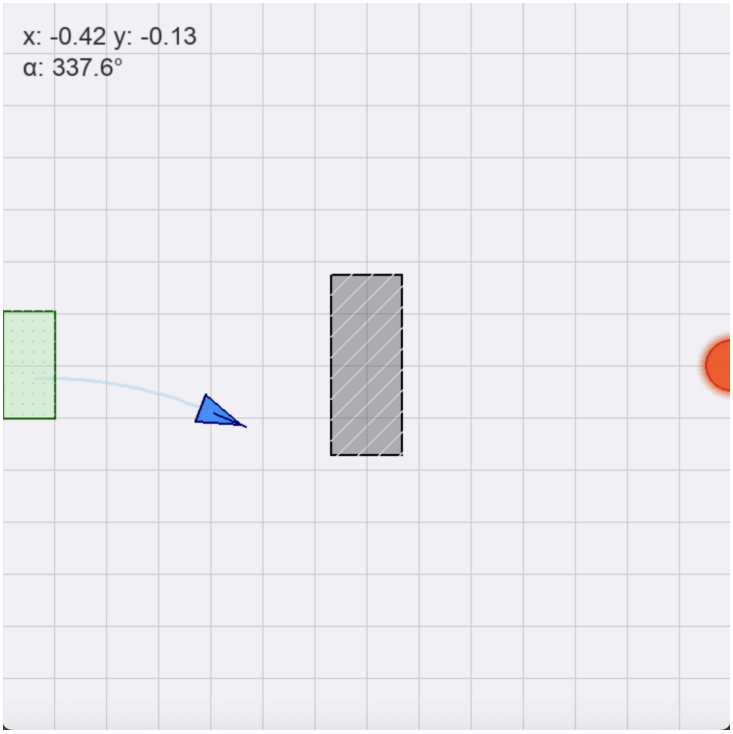

Figure 4.1: 2D OpenAI Gym environmnet for Task 1.

sitioning to more complex simulation environments, ensuring that fundamental design decisions were sound before introducing additional complexity.

The 2D environment provides a simplified abstraction of the real-world dynamics while maintaining the essential characteristics of the navigation problem. This setup enabled efficient iteration and testing of various reward function formulations without the computational burden of a full physics-based simulation. It allowed us to focus on optimizing the reward design itself, ensuring it was not influenced by system dynamics or controller behavior.

The 2D environment is shown in Fig. 4.1. The robot is represented by a blue triangular shape. The green box denotes the initial safety area where the robot can spawn at the beginning of each episode. The red semicircle marks the goal area that the robot must reach to complete the task, while the gray box in the center represents an obstacle that must be avoided. A more detailed explanation of the rationale and implementation details is provided in Appendix A.1.

4.1.2 3D Gazebo Environment

Following the successful development and testing in the 2D environment, all experiments were migrated to a more realistic 3D simulation using the Gazebo platform, as shown in Fig. 4.2. Gazebo offers high-fidelity physics simulation and 3D visualization, enabling accurate modeling of robot dynamics and environmental interactions.

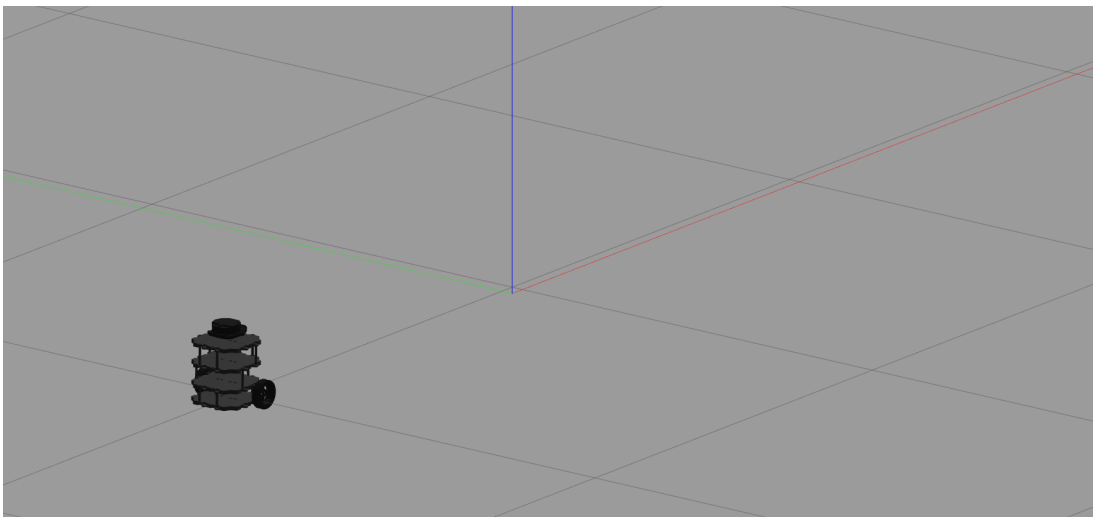

Figure 4.2: TurtleBot3 in Gazebo "Empty World" Environment for Task 1 and 2.

In this environment, the robot is fully modeled as a differential-drive system with realistic:

- Physical dimensions and inertial properties
- Wheel-ground interactions including slip and friction
- Sensor limitations and noise characteristics
- Motor dynamics and actuation constraints
- Environmental elements such as obstacles and terrain

This transition from 2D to 3D simulation represents a critical step in the validation pipeline, allowing assessment of algorithm generalization from abstract to realistic conditions. The reward functions and policies developed in 2D were adapted and fine-tuned for this environment to ensure robust performance under increased physical complexity.
The Gazebo simulation thus acts as an intermediate stage between pure simulation and hardware deployment, offering a controlled but realistic setting to identify failure modes, test latency tolerance, and verify control stability. This two-phase approach improves the efficiency of algorithm development while increasing the likelihood of successful transfer to physical robots.
In the Gazebo “Empty World” setup (Fig. 4.2), which served as the base for Task 1 and Task 2, everything is managed via software: (Task 1) obstacle collisions and map handling (including the safety zone) are managed purely in software; (Task 2) trajectory following and the safety-zone-based episode termination are likewise implemented in software.
The realistic TurtleBot3 model and all supporting software components developed here are later reused in the laboratory.

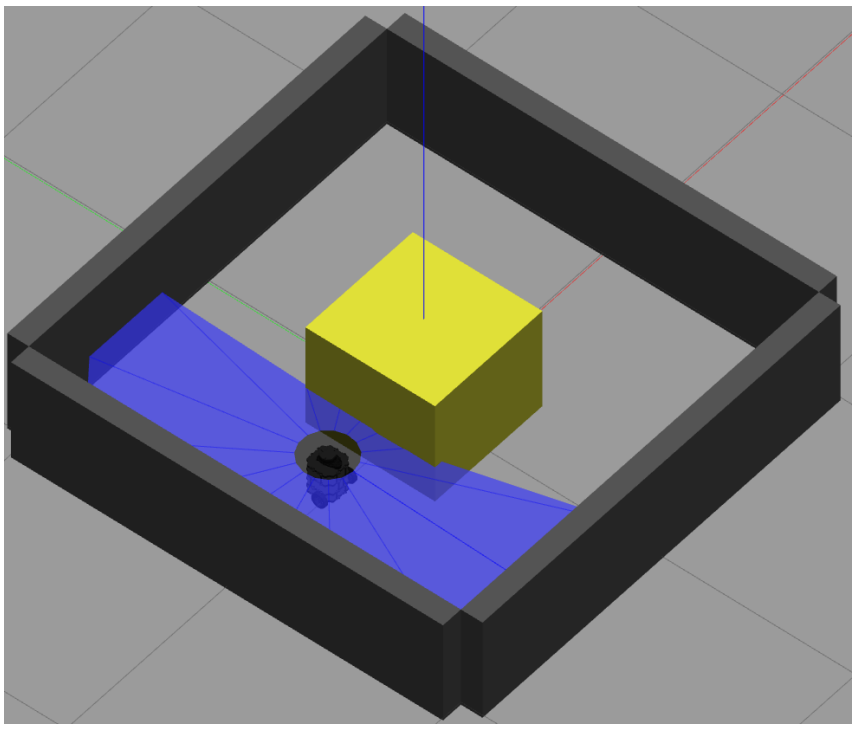

Figure 4.3: TurtleBot3 in Gazebo Custom World for Task 3.

For Task 3, a custom environment mirroring the laboratory equipment was built. Four walls (0.5 m high, 2.0 m long) were arranged around a central box (0.6 m $\times$ 0.6 m) to match the real-world setup. As illustrated in Fig. 4.3, this structure enables a one-to-one correspondence between simulation and hardware tests.

4.2 Hardware and Communication Protocol

4.2.1 DEI TurtleBot

The robot employed for real-world testing is a custom TurtleBot3, equipped with a Raspberry Pi 3B as its core processing unit, a Nucleo board for motor control, and a LiDAR sensor for obstacle detection, as shown in Fig. 4.4, along with other peripherals that are not used in this setup. A motion capture system (VICON) provides accurate pose estimation, while speed measurements are available from the wheel encoders. Real-world robot operations were carried out using ROS1, aligning with the simulation protocols established in Gazebo (ROBOTIS, 2020; Koenig and Howard, 2004).

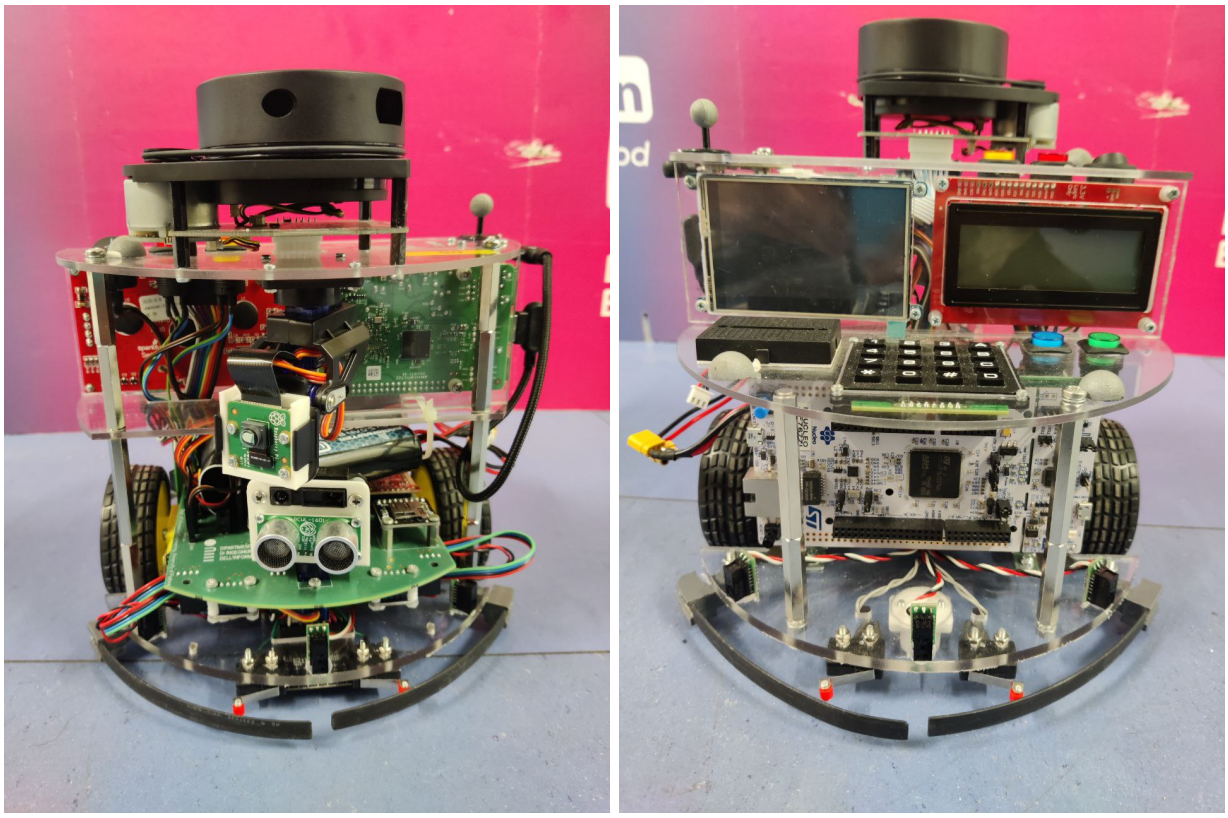

Figure 4.4: Front and Back of the DEI TurtleBot.

4.2.2 ROS 1 Network

The ROS 1 framework serves as the communication backbone for the distributed robotic system. The main PC of the Laboratory, where the experiments were conducted, acts as the master node by hosting the ROS Master, which coordinates all other nodes across the network.

The ROS Master provides naming and registration services for the rest of the system. It maintains a registry of publishers and subscribers to topics, as well as available services. Its primary function is to enable ROS nodes to discover one another; once this discovery phase is complete, nodes communicate with each other directly in a peer-to-peer fashion.

A Raspberry Pi 3B mounted on the custom TurtleBot operates as a client device. It runs low-level ROS nodes responsible for acquiring sensor data and handling motor control via communication with the Nucleo board. Sensor data from the LiDAR and wheel encoders is published over ROS topics, while velocity commands are received from higher-level nodes. Additionally, RL node runs directly on the Raspberry Pi 3B, meaning that all learning and decision-making computations are executed locally on the TurtleBot. The Raspberry Pi computes the desired motion based on the RL policy and sends high-level commands (linear and angular velocities). Based on these, a low-level PID controller

regulates the current supplied to the DC motors.
These commands are then translated into low-level control signals used to drive the motor gear, enabling the physical actuation of the robot's wheels.
An additional PC is dedicated to managing the processing of data from the VICON motion capture system. This machine interfaces with the VICON cameras for image acquisition and optional visual processing. It is also connected to the ROS network, allowing it to publish pose data.
All devices operate within the same local network, with environment variables (`ROS_MASTER_URI` and `ROS_IP`) configured to ensure proper inter-node communication. Time synchronization is maintained using NTP to support accurate timestamping, which is essential for sensor fusion and logging. This networked ROS 1 setup mirrors the architecture used in simulation, ensuring consistency between virtual and physical deployments. A simplified representation can be seen in Fig. 4.5.

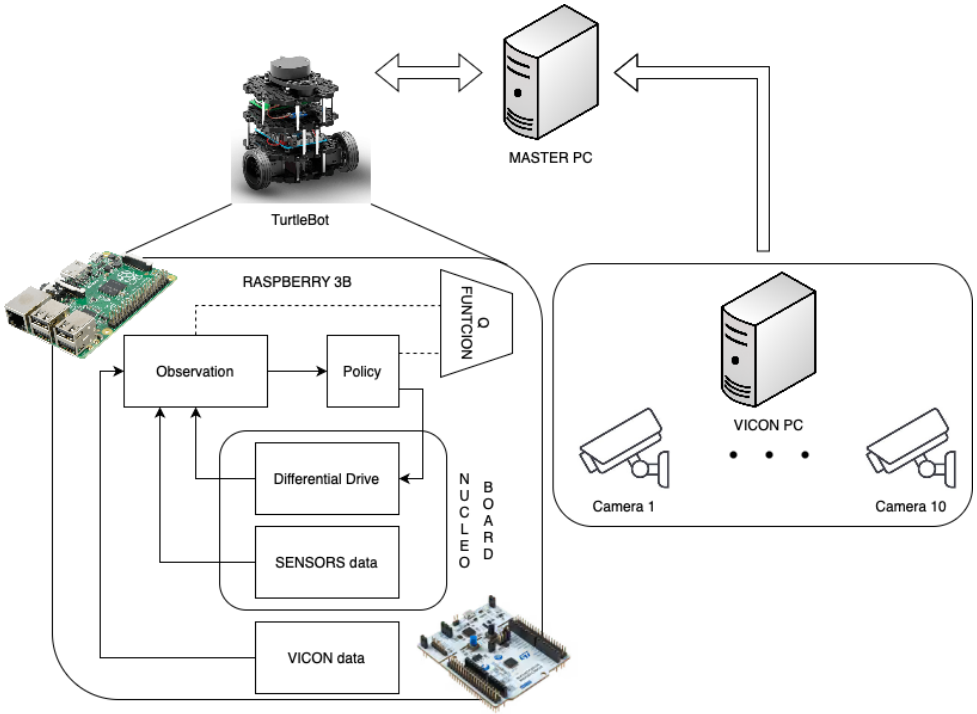

Figure 4.5: Simplified ROS network architecture.

4.3 Neural Network Architectures

Designing neural networks for embedded-hardware reinforcement learning requires balancing representational power against real-time execution constraints. On a Raspberry Pi 3B, each forward and backward pass must complete within a few hundred milliseconds to support control loops running at several hertz. The core components of a feedforward

neural network are first reviewed, followed by an explanation of how the final architectures were selected based on empirical timing tests and stability evaluations.

4.3.1 Feedforward Networks

A feedforward (multilayer perceptron) network consists of an input layer, one or more hidden layers, and an output layer, as shown in Fig. 4.6. Each neuron computes a weighted sum of its inputs followed by a nonlinear activation. The key design choices impact both learning performance and computational load:

- **Depth (number of layers):** More layers allow modelling of complex relationships, but each additional layer increases inference latency.
- **Width (neurons per layer):** Wider layers can capture richer features, yet they also demand proportionally more multiply–accumulate operations.
- **Activation functions:** Common choices like ReLU or tanh balance nonlinearity against numerical stability and execution cost.
- **Batch size:** Larger batches produce smoother gradient estimates but require more memory and lengthen each training step.
- **Initialization:** Methods such as Xavier uniform help maintain healthy gradient scales across layers.

In this context, the network depth was fixed to two hidden layers, and various widths and batch sizes were systematically profiled to identify an optimal configuration.

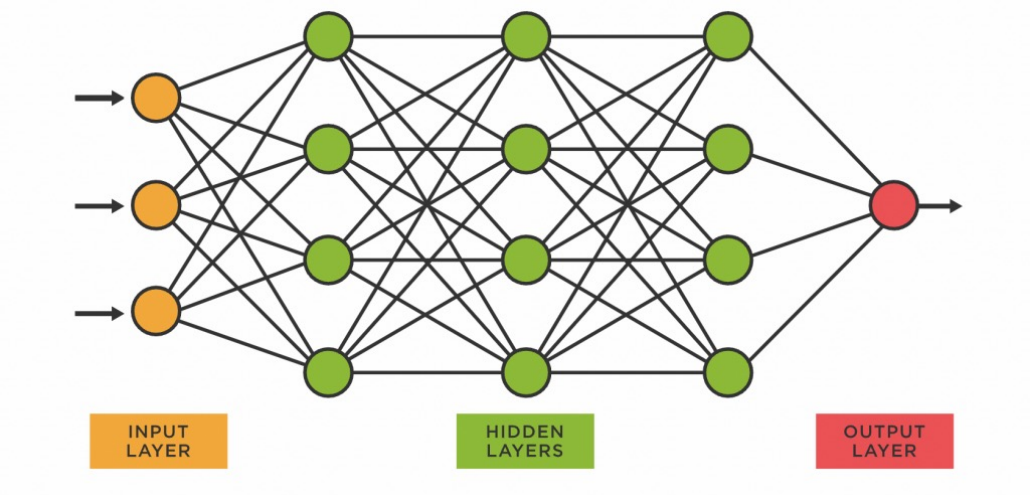

Figure 4.6: Neural network architecture used for policy learning.

4.3.2 Common Architecture & Hyperparameters

After extensive preliminary experiments, we settled on the following configuration for all three algorithms (DDPG, TD3, and SAC). This choice was driven by the need to maintain control and update rates above 3 Hz under full training load:

- **Network architecture:** Both actor and critic networks consist of two hidden layers with 64 neurons each, using ReLU activation functions.
- **Weight initialization:** PyTorch default initialization is used (Kaiming uniform).
- **Replay buffer:** A capacity of 1×10^6 transitions is used.
- **Batch size:** Set to 128 to balance stable gradient estimation with computational efficiency and latency considerations.
- **Learning rates:** Both actor and critic optimizers use a fixed learning rate of 3×10^{-4}.
- **Discount factor and soft update rate:** A discount factor $\gamma = 0.99$ promotes long-term rewards, while a soft target update rate of $\tau = 0.005$ ensures gradual tracking of target networks.

By using identical network shapes and optimizer settings, the effects of algorithmic differences were isolated from those of network architecture.

4.3.3 Actor and Critic Structures

Actor networks translate sensor observations into control commands. Each actor receives the state vector as input, processes it through two 64-neuron ReLU layers, and produces:

- For DDPG/TD3: a deterministic action vector (linear and angular velocity) via a tanh output, scaled to the robot's commanded range.
- For SAC: two parallel outputs (mean and log-standard deviation) defining a Gaussian policy distribution.

Critic networks estimate the expected return of state–action pairs. In DDPG, a single critic processes the state through one ReLU layer, concatenates the action, and passes the result through a second ReLU layer to yield a scalar Q-value. TD3 and SAC employ two independent critics of identical structure to mitigate overestimation bias.

4.3.4 Profiling and Final Selection

To ensure real-world deployability, it was essential to identify neural network configurations that are computationally feasible on the onboard hardware (i.e. the Raspberry Pi 3B). A systematic profiling procedure was carried out to measure both the control loop frequency (actor inference rate) and the training update rate (gradient step frequency) across a grid of network configurations, defined by combinations of hidden-layer widths and batch sizes.
Figure 4.7 summarizes the results of this profiling. The selected configuration, comprising two hidden layers of 64 neurons each and a batch size of 128, strikes a practical balance between learning capacity and execution speed. Under this setup, the average control/update rates observed on the Raspberry Pi 3B were:

- **DDPG:** 5.8 Hz
- **TD3:** 5.9 Hz
- **SAC:** 3.3 Hz

These measured frequencies reveal a notable gap between actor–critic methods with deterministic policies (DDPG and TD3) and the stochastic policy used in SAC. The increased computational load in SAC arises from both its entropy-regularized objective and the use of separate networks for value estimation and policy sampling.
This profiling informed the final algorithm selection and also influenced the design of the simulation experiments. Specifically, the same control/update frequencies were enforced in simulation to ensure consistency with real-world timing constraints, enabling fair and meaningful sim-to-real comparisons.
A complete description of the profiling methodology is provided in Appendix A.1.

4.4 Training and Evaluation Protocol

A consistent training and evaluation pipeline was designed to assess the performance and transferability of the selected algorithms across simulation and real-world environments. The training phase was first conducted in simulation using the Gazebo framework, with different durations chosen based on task complexity: 200 episodes for Task 1, 400 episodes for Task 2, and 15 000 time-steps for Task 3.
Every 20 episodes (or every 5 000 time-steps in Task 3), deterministic evaluation rollouts, without exploration noise, were performed to monitor learning progress and select the best-performing checkpoints for deployment.
The exploration strategy was tailored for each algorithm. DDPG and TD3 followed a linear decay of Gaussian noise from $\sigma = 0.3$ to $\sigma = 0.1$ over 300 episodes, both in

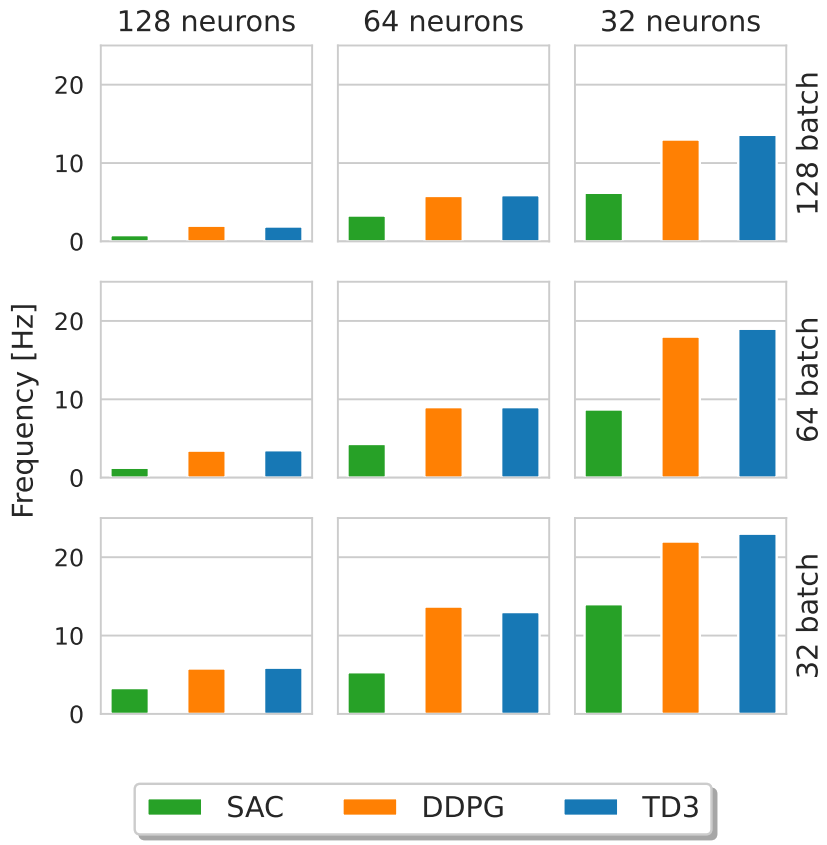

Figure 4.7: Control and training update frequencies on the Raspberry Pi 3B for various network widths and batch sizes.

simulation and real-world training from scratch. SAC, in contrast, relied solely on its entropy-driven stochastic policy and did not incorporate external noise scheduling.

For fine-tuning, a milder decay from $\sigma = 0.1$ to $\sigma = 0.05$ over 50 real-world episodes was used, facilitating smoother adaptation to sensor noise and physical inconsistencies.

Two real-world deployment strategies were explored. The first involved simulation pre-training followed by real-world fine-tuning. Here, both actor and critic networks were initialized from the best simulation checkpoint and trained further on the physical robot. The second strategy trained policies from scratch directly on the real platform using the same noise schedule as in simulation. Tasks 1 and 2 employed 100 real-world episodes for each strategy, with evaluation rollouts every 20 episodes. For Task 3, due to varying episode lengths and power constraints, a fixed budget of 5 000 time-steps was allocated, with performance evaluations conducted only at the beginning and end of training.

To ensure a fair and realistic sim-to-real comparison, the control and training update frequencies observed on the Raspberry Pi 3B (as profiled in the previous section) were mirrored in simulation. This alignment ensured that all policies were subject to the same computational constraints as those encountered in real-world execution, thereby preserving the fidelity of performance comparisons.

Chapter 5

Results and Discussion

This chapter presents the results obtained throughout the project, including pre-training in the Gazebo environment, and real-world deployment in the SPARCS Laboratory (DEI). For each task, performance and reward statistics are aggregated across four independent random seeds. The figures, presented in Fig. 5.1, 5.5, 5.9, show three distinct phases of the training process:

- **Pre-training (Simulation-only)**: Negative x-axis values correspond to training in simplified simulation environments.
- **Fine-tuning (Sim-to-Real Adaptation)**: Initial positive x-axis values represent policy fine-tuning on the real robot after pre-training.
- **Real-World Training (from Scratch)**: Later positive x-axis values show learning curves for policies trained exclusively with real-world interactions.

5.1 Task 1: Point-to-Point Navigation

The agent was trained in the Gazebo simulation environment, following the methodology detailed in Chapter 4. The results obtained from this training phase clearly illustrate the impact of Gazebo's limited update frequency on the performance of different reinforcement learning algorithms. As shown in Figure 5.1, the performance varies significantly across algorithms, which suggests that the simulation constraints play a non-trivial role in influencing learning outcomes.
Notably, DDPG algorithm, typically considered less powerful compared to more recent algorithms such as SAC or TD3, unexpectedly outperforms its counterparts under these conditions. This result highlights how algorithmic simplicity, combined with the specific timing constraints of the simulator, may offer an advantage in certain robotic learning tasks. On the other hand, SAC, which is often praised for its stability and robustness,

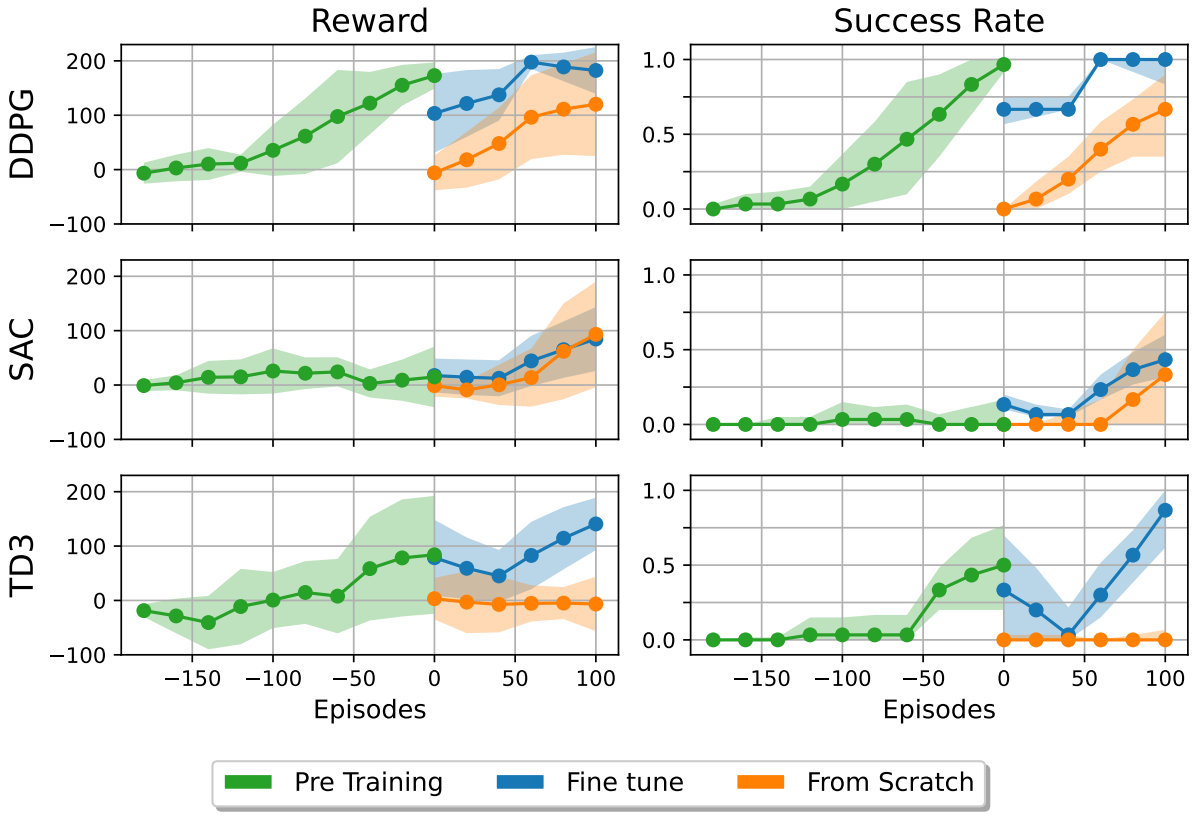

Figure 5.1: Comparison of learning and performance curves for Task 1.

suffers from substantial computational overhead. This inefficiency translates into slower learning and reduced overall performance in terms of both average reward and task success rate. TD3 also exhibits limited performance, likely due to its sensitivity to hyperparameters and increased algorithmic complexity.

An especially intriguing aspect of the study is the transition from simulation to real-world deployment. All algorithms demonstrate a significant performance drop when initially applied to the physical system. This gap underscores the challenges associated with sim-to-real transfer, such as discrepancies in dynamics modeling, sensor noise, and actuation delays. However, with continued training in the real environment, the agents gradually recover performance levels similar to those attained in simulation. This adaptation process suggests that while initial policy generalization may be limited, reinforcement learning agents retain the capacity to relearn and adapt through real-world experience.

Another notable result is observed in the behavior of DDPG trained from scratch in the real-world environment. Surprisingly, the from-scratch policy achieves performance levels nearly on par with those obtained through fine-tuning, which speaks to the generalization capacity of the neural network and the suitability of DDPG for certain low-frequency, continuous control tasks. In contrast, TD3 and SAC fail to match this level of real-world adaptability, reinforcing the idea that algorithmic complexity does not always translate to

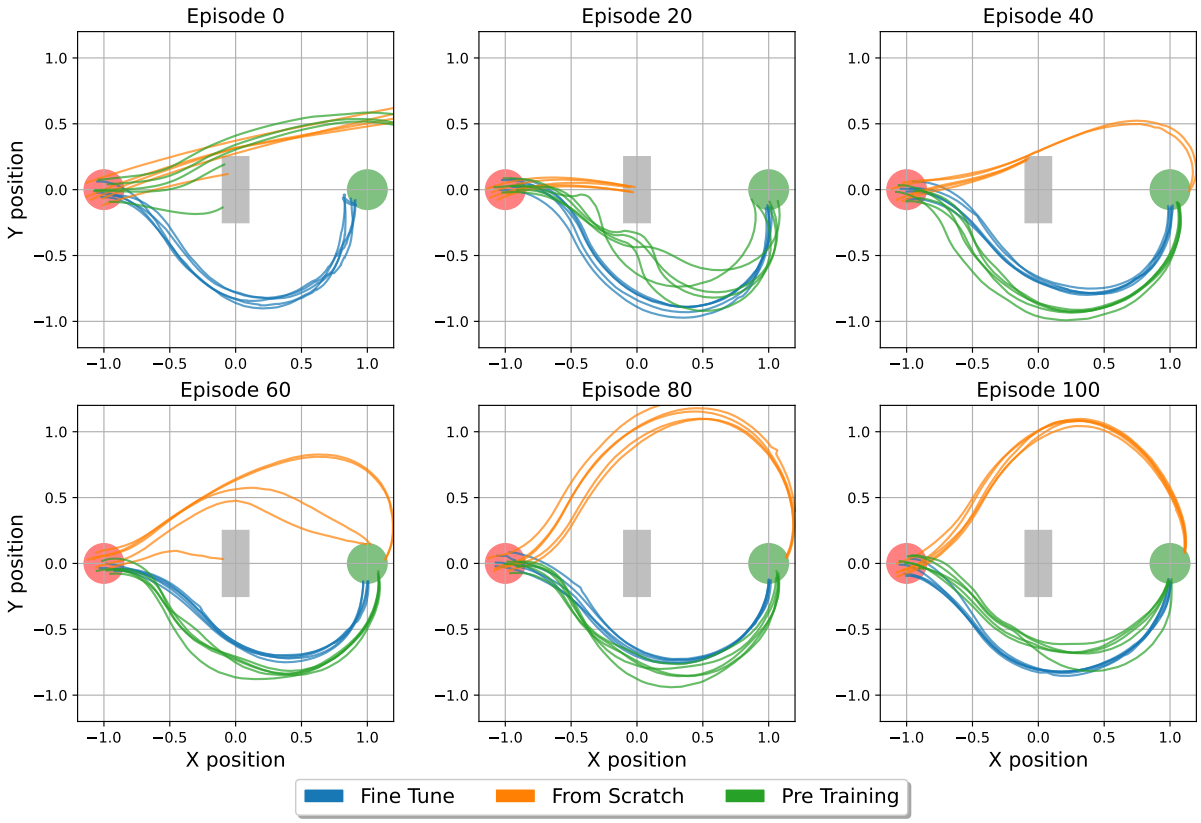

Figure 5.2: Trajectories example for DDPG seed 0 for Task 1.

superior real-world performance, especially in environments constrained by computational or physical limitations.

Since we are dealing with autonomous navigation, it is essential not only to evaluate the performance in terms of reward or success rate, but also to analyze how the robot moves and which trajectories it follows to reach the target. This behavior is entirely determined by the design of the reward function. Therefore, it is not simply a matter of optimality as might be initially assumed. In fact, the perceived optimality of the trajectory depends strictly on how the reward function is formulated. It is possible that the resulting path is suboptimal compared to what was expected or in comparison with trajectories generated by optimal or model predictive control algorithms.

The Fig. 5.2, 5.3, and 5.4 illustrate the trajectories executed by each algorithm during a training cycle, with trajectory snapshots taken every 20 episodes to validate the evolving performance of the learned policy. In each figure, the general layout includes a gray box indicating the static obstacles, a green circle denoting the agent's starting position, and a red circle representing the goal area. For clarity and simplicity, only a subset of training seeds has been reported. Maintaining consistency with the color coding used in the previous performance plots, each category displays five different trajectories executed by the robot.

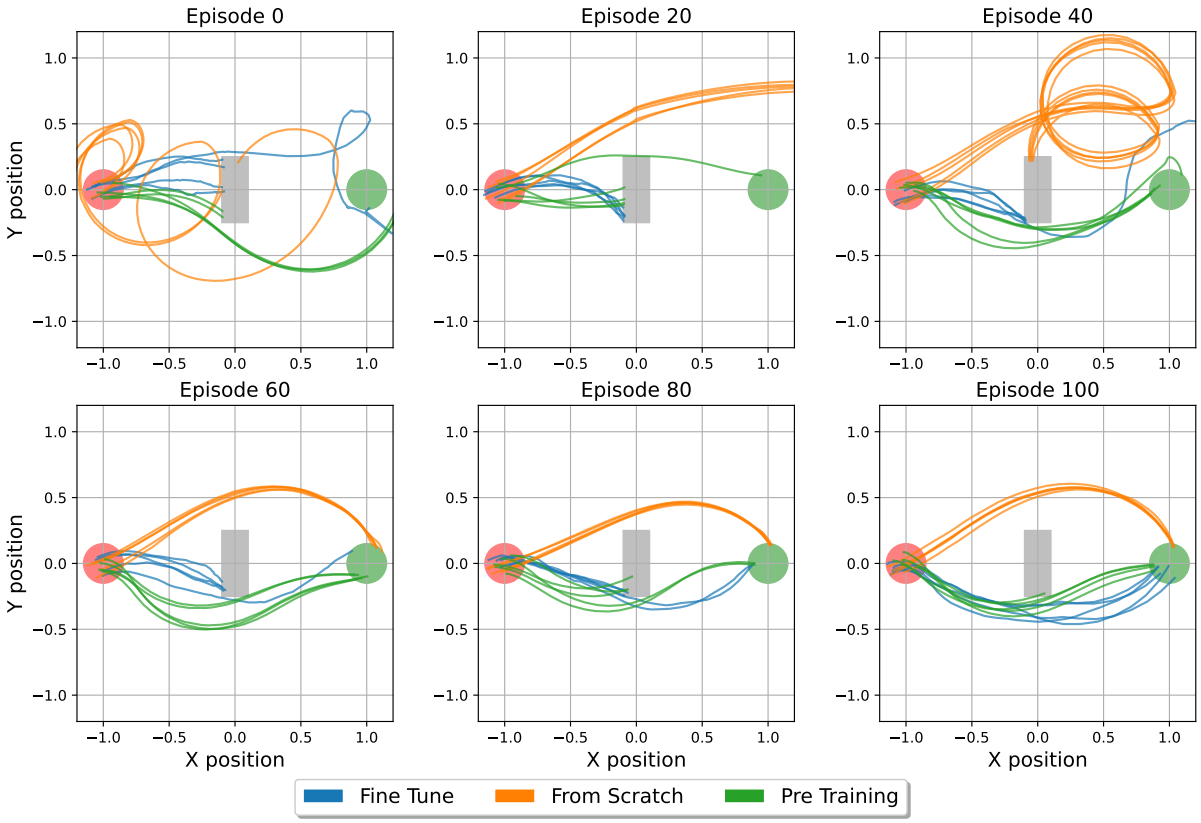

Figure 5.3: Trajectories example of SAC seed 2 for Task 1.

The green trajectories, representing the pre-trained model, clearly demonstrate the DDPG algorithm's capability to improve over time. Initially, the agent learns to avoid obstacles without yet reaching the target. As training progresses, however, the trajectories become increasingly refined, with the robot ultimately navigating towards the goal in a smooth and stable manner, avoiding unnecessary oscillations or erratic movements.
The orange trajectories, corresponding to the policy trained from scratch, exhibit a similar initial behavior: the agent manages to avoid the obstacle but fails to complete the task. Interestingly, at a later stage (e.g. Fig. 5.2 episode 20), the agent appears to "forget" previously learned strategies and repeatedly crashes into the obstacle. This phenomenon arises during the early training stages, when the robot is still exploring the state space and has not yet developed reliable navigation strategies. Such behavior is evident throughout the initial episodes, for example, the erratic trajectories observed at episode 40 in Fig. 5.3 and between episodes 20–40 in Fig. 5.4.

Nonetheless, as training continues, the agent gradually improves and reaches the final target more frequently. It is worth noting, however, that the resulting trajectories differ from those of the fine-tuned model. While the fine-tuned policy tends to produce more refined and efficient paths, closely resembling those from pre-training, the from-scratch model often converges to alternative but still viable solutions.

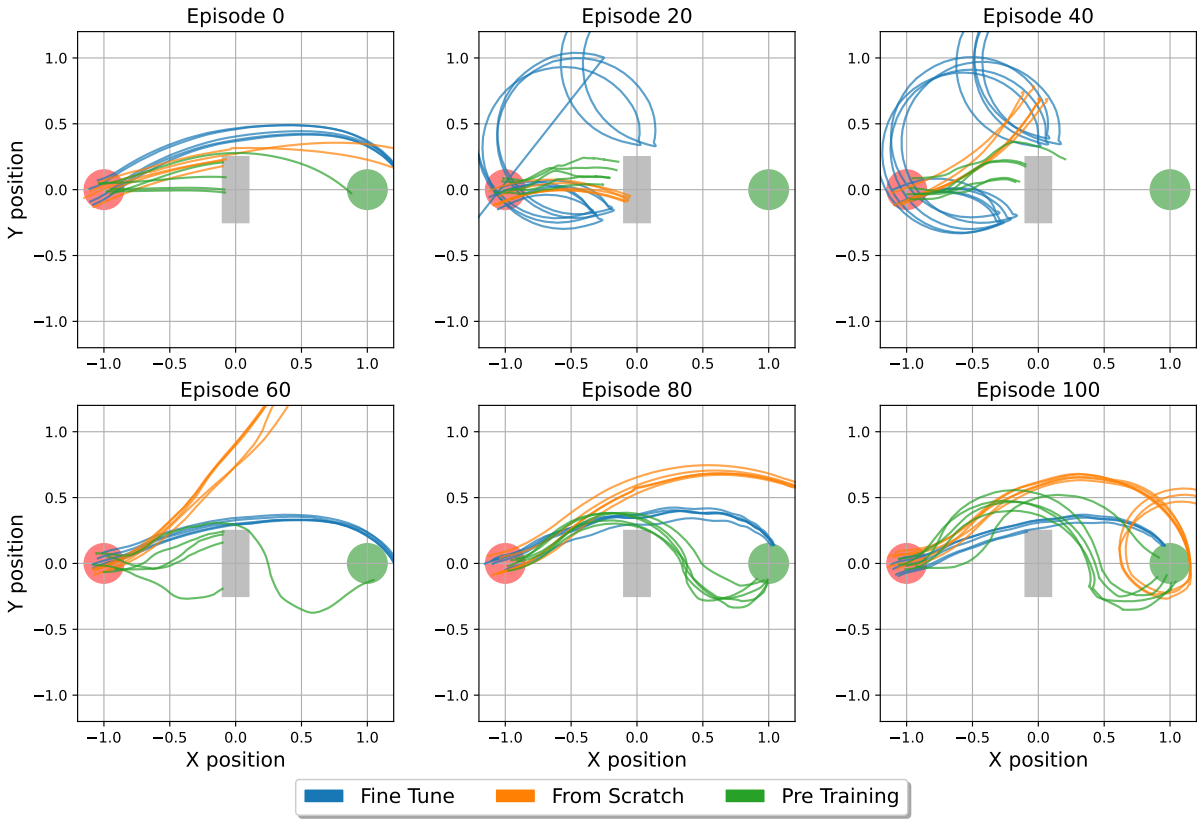

Figure 5.4: Trajectories example of TD3 seed 0 for Task 1.

These qualitative results reinforce the importance of careful reward design and trajectory analysis in evaluating navigation policies. A purely quantitative evaluation might overlook critical aspects of how the robot behaves in physical space, particularly in applications where smoothness, safety, and interpretability of the trajectory are crucial.

5.2 Task 2: Path Following

The second task proved to be the most challenging in terms of reward-function design and consequently exhibited the lowest overall performance among all tasks. As illustrated in Figure 5.5, achieving an acceptable level of performance for even one of the algorithms required roughly twice the number of simulation episodes compared to Task 1. During the pre-training phase in Gazebo, both DDPG and TD3 display nearly identical learning curves, as expected given their shared update frequency and comparable architectural complexity.

Upon transitioning to the laboratory fine-tuning phase, only DDPG manages not only to preserve its simulated performance but also to improve upon it, demonstrating rapid adaptation to the real environment without any discernible sim-to-real gap. In fact, the initial reward values in the laboratory exceed those at the end of simulation, indicating a

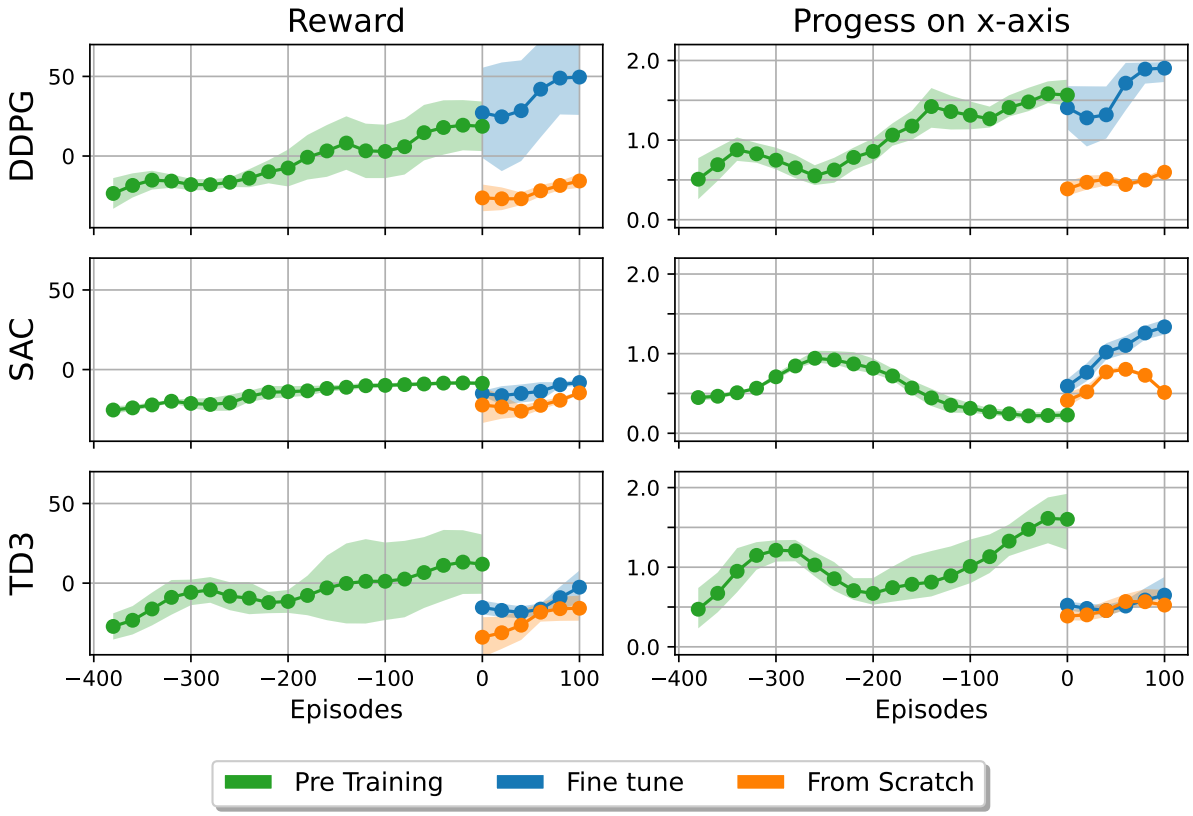

Figure 5.5: Comparison of learning and performance curves for Task 2.

robust transfer capability of the learned policy. In contrast, both TD3 and SAC suffer from a noticeable performance drop: their learning curves start at significantly lower reward levels and require many more real-world episodes to approach the simulation benchmarks.

Given the intrinsic difficulty of the path-following task, the 100 real-world episodes allocated for training from scratch proved insufficient to establish an effective policy. The corresponding learning curves remain almost flat, with only marginal episodic increases, and never attain the reward levels achieved by the pre-trained or fine-tuned models.

Finally, SAC once again records the poorest performance overall: its reward curve is particularly flat throughout most of the training, except for an apparent rise during fine-tuning. However, as Fig. 5.7 shows, this apparent improvement is misleading: instead of faithfully following the prescribed sinusoidal trajectory, the robot "cuts through" the path, deviating significantly from the intended curve, which results in persistently low reward values.

Figure 5.6 shows six snapshots of the robot's x–y trajectory at episodes 0, 20, 40, 60, 80 and 100, with four overlaid curves: the ideal reference (dashed black), the pre-trained model (green), the fine-tuned policy (blue) and the from-scratch policy (orange). At Episode 0, only the fine-tuned trajectory (blue) is already able to follow the sinusoidal path consistently, whereas the from-scratch trajectory (orange) fails immediately. The

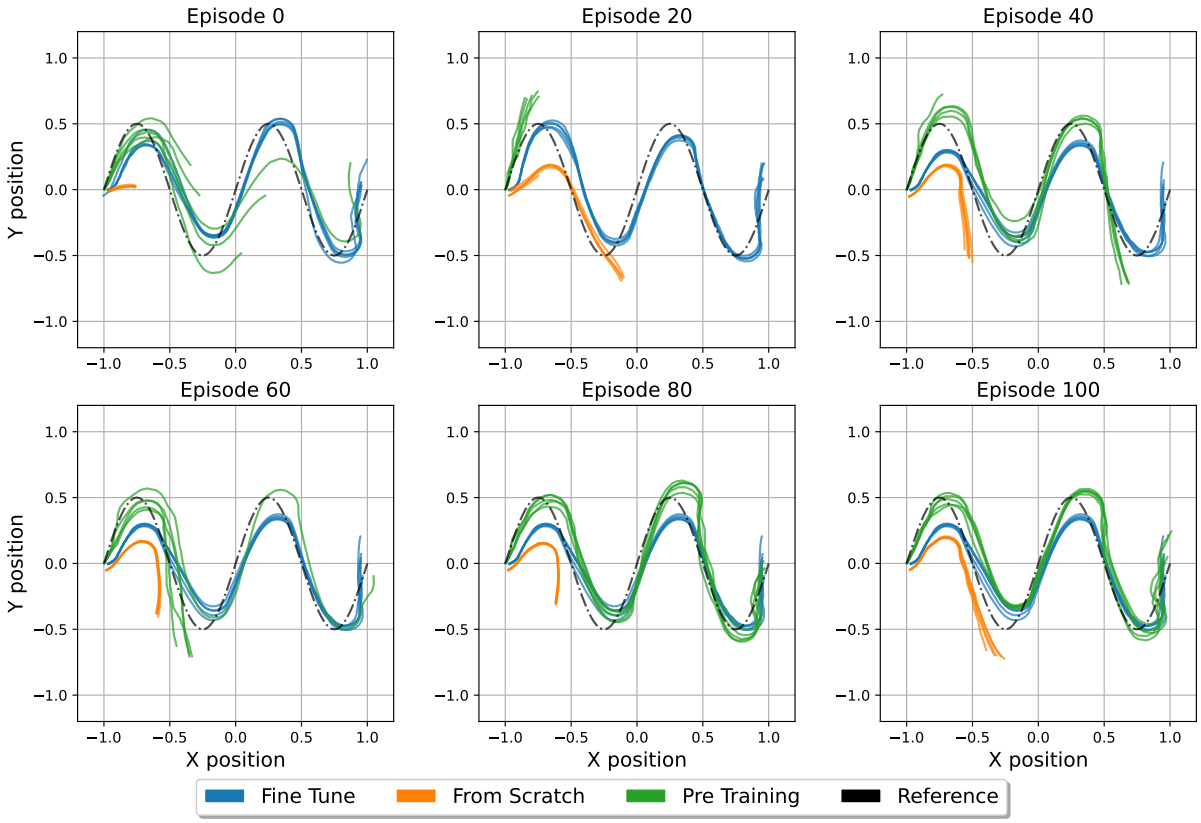

Figure 5.6: Trajectories example of DDPG seed 1 for Task 2.

pre-training trajectories (green) illustrate the progressive improvement: initially rough but increasingly refined across episodes, they demonstrate how simulation-based learning equips the agent with a growing ability to approximate the reference path.

By episode 20, the fine-tuned policy has already "locked on" to the sine shape, closely matching the pre-trained trajectory in the final stage, whereas the from scratch trajectory reaches the first peak but still cuts the first curve, coming on track on the descending segment. Next it momentarily regresses, veering outside the limited region, highlighting the high variance of early exploration. Although the orange trajectories show marginal improvement in later episodes and even briefly traverse the second half-wave, the allotted training is insufficient for mastering the full path: ultimately, only about 30 % of the intended sinusoid is reliably followed.

Fig. 5.7 presents SAC trajectories for seed 0 reporting the same previous curves. At episode 0, none of the SAC policies approximate the sinusoid: both fine-tuned and from-scratch trajectories cut diagonally through the troughs, while the pre-trained model skirts the first peak before diverging sharply off-course.

By episode 20, the fine-tuned policy begins to climb the rising slope of the first half-wave, yet still "short-cuts" the valleys; the from-scratch policy remains erratic, failing to complete even the first part. At episode 40, all SAC trajectories deviate substantially: the

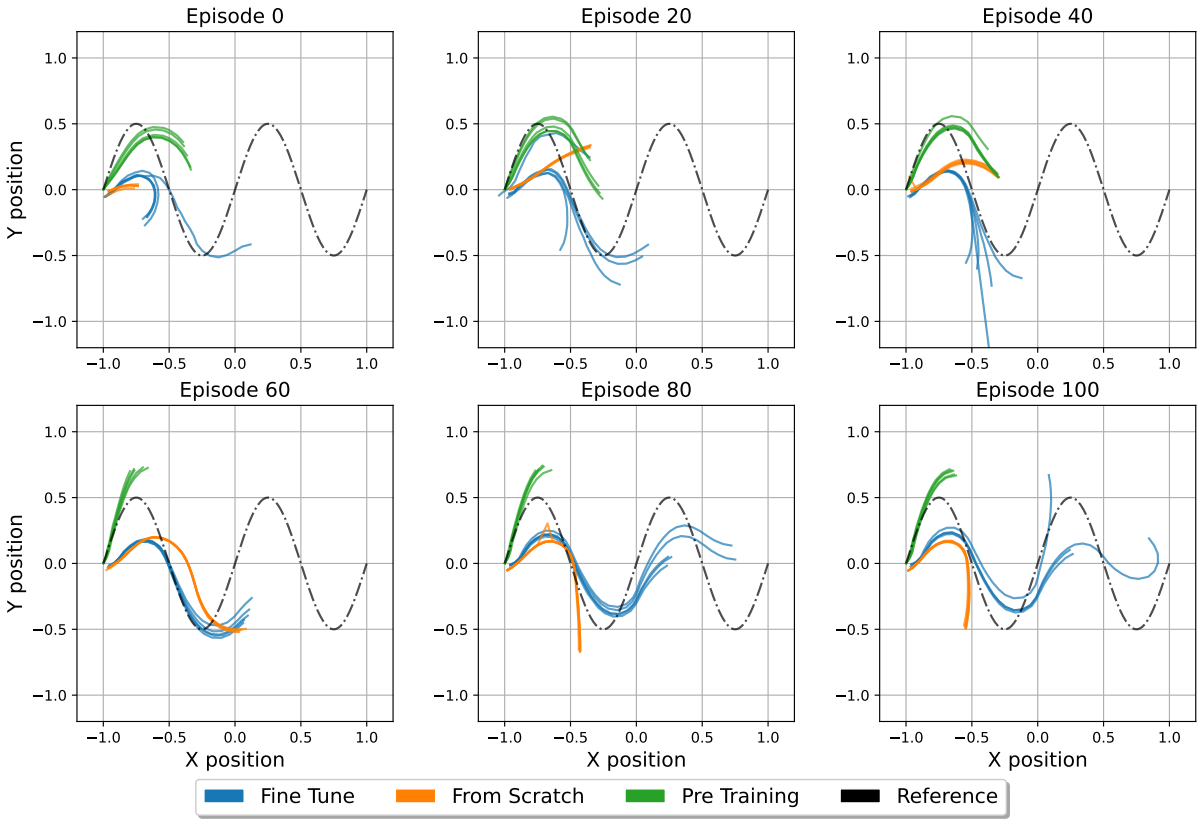

Figure 5.7: Trajectories example of SAC seed 0 for Task 2.

fine-tuned path aggressively slices through troughs, the pre-trained path overshoots the peaks, and the from-scratch path strays further from the intended curve.

During episodes 60 and 80, the fine-tuned policy produces a recognizably sinusoidal segment, but each pass continues to slice across the intended valleys. Both the pre-training and from-scratch models exhibit intermittent successes, occasionally following small arcs of the sine, but cannot sustain consistent path completion. Even by episode 100, the best SAC policy tracks at most two-thirds of the sine wave before reverting to corner-cutting behavior, and all curves have plateaued into suboptimal, jagged shapes.

As with the DDPG algorithms, the pre-training phase is long enough to demonstrate strong progression. For instance, in Fig. 5.8 the agent almost completes the entire trajectory, failing only at the final curve, suggesting that, given more training time, it could fully master the task. However, this apparent success does not translate into generalization: both the fine-tuned and from-scratch agents achieve similar performance, effectively rendering the initial simulation gains useless. This shortcoming is likely due to the limited generalization capacity of the neural network, which becomes especially evident when the policy is transferred out of the pre-training regime.

In summary, Task 2 highlights how critical reward-function design and task complexity are in shaping the behavior of reinforcement-learning algorithms. It underscores the necessity

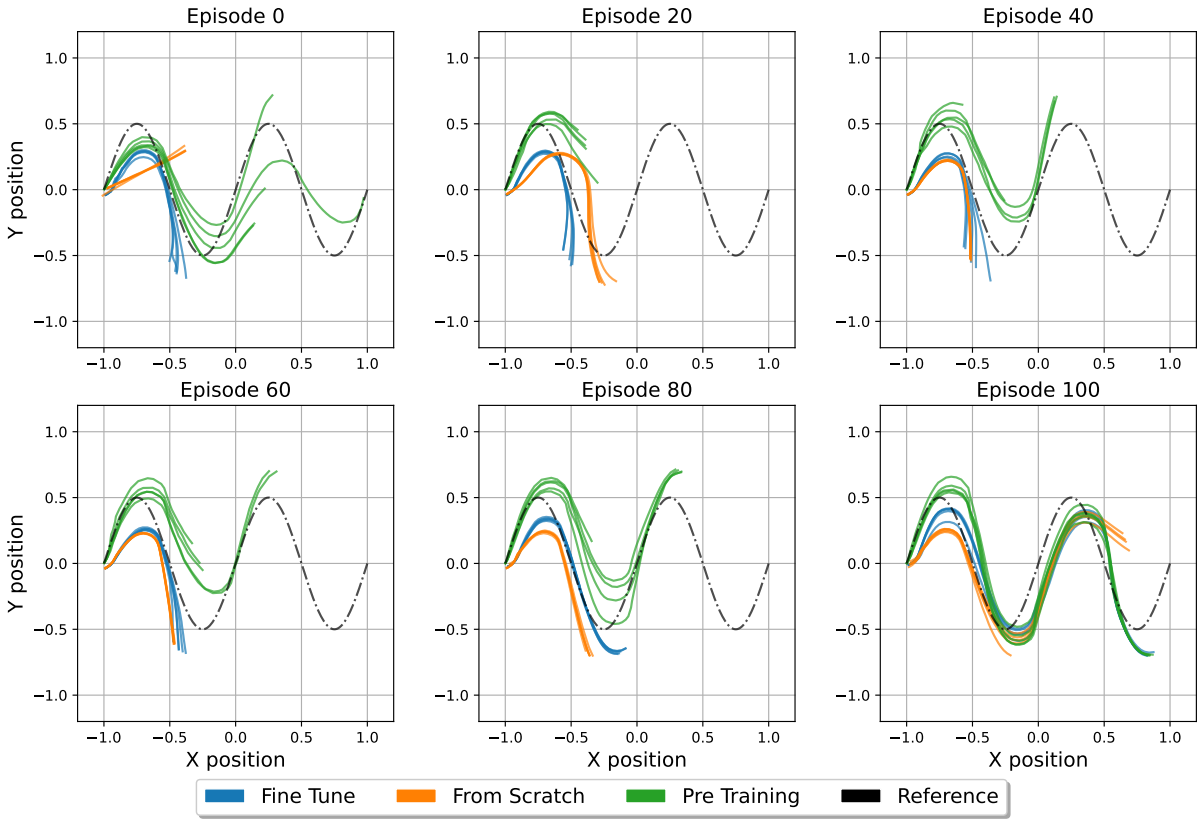

Figure 5.8: Trajectories example of TD3 seed 3 for Task 2.

of thorough simulation studies and an adequate number of real-world training episodes to ensure successful sim-to-real transfer.

5.3 Task 3: Corridor Navigation

The last task was by far the most engaging and enjoyable to develop. Although the objective, navigating continuously around a circular corridor without colliding may appear straightforward at first glance, it hides a number of subtle challenges. First, the robot must maintain a precise balance between angular velocity and translational drift; even a slight oversteer or understeer can cause a gradual drift that results in a collision with the walls. Second, sensor noise and actuation latency introduce unpredictability into the control loop, requiring the agent to learn a robust policy that tolerates small perturbations. Third, the reward shaping had to be carefully tuned: too sparse a reward led to aimless spinning, while too dense a penalty for proximity to obstacles discouraged exploration of the full turning radius.

Fig. 5.9 shows the evolution of both reward and traveled distance for this task. During the pre-training phase, all algorithms exhibit consistently high performance, achieving comparable reward values and covering similar distances, often completing more than one

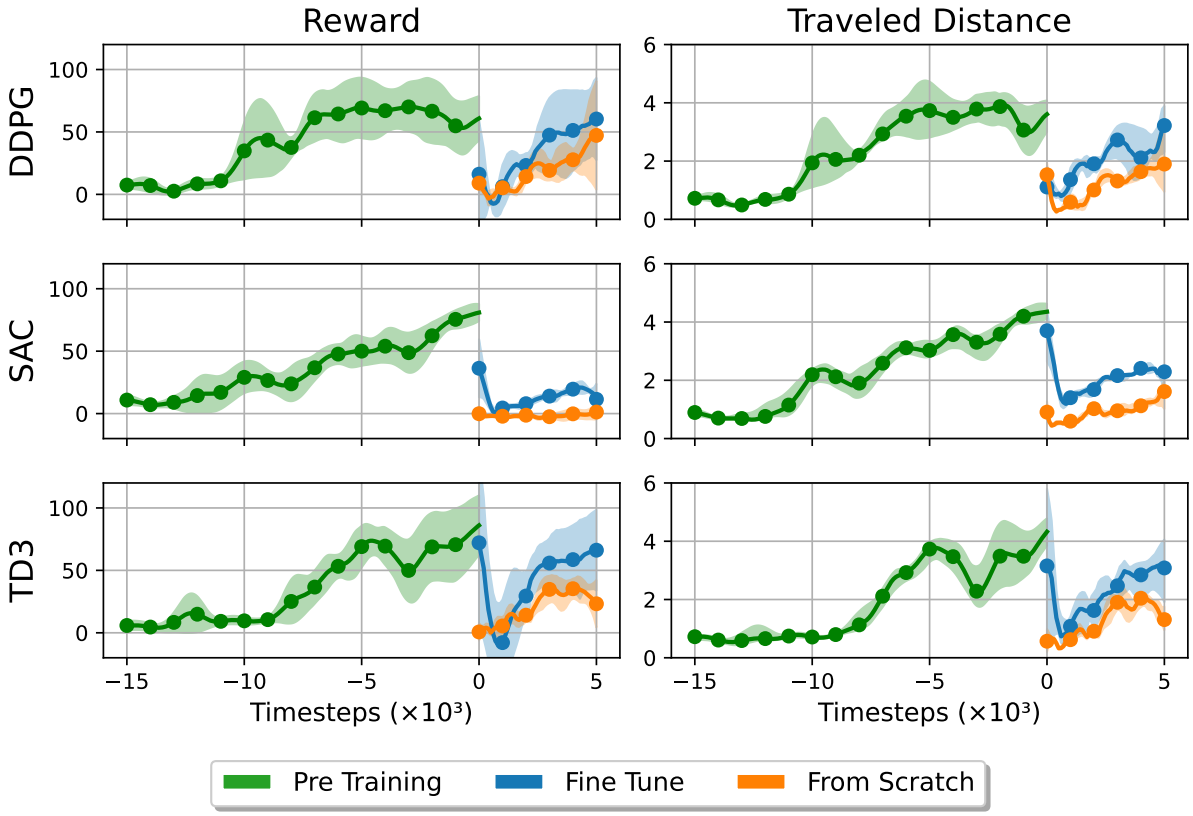

Figure 5.9: Comparison of learning and performance curves for Task 3.

full lap of the circular corridor, with termination only due to the default episode time limit. However, once deployed on the real robot, significant discrepancies in the LiDAR measurements (which were not modeled in the Gazebo simulation) become apparent. These unmodeled sensor errors cause a pronounced sim-to-real gap, initially erasing the benefits gained during pre-training for the fine-tuned policies.

Despite this setback, the robot is able to adapt rapidly: both DDPG and TD3 regain nearly their simulation-level performance after only a few episodes in the real environment. By contrast, the policies trained from scratch start without any built-in advantage from simulation and thus show a slower learning curve, only DDPG eventually matches the performance of its fine-tuned counterpart. The SAC algorithm, on the other hand, fails to recover from the real-world discrepancies and consistently underperforms.

As shown in Figure 5.10, which focuses solely on the comparison between from-scratch and fine-tuned policies, all three algorithms gradually increase their minimum LiDAR readings over the course of training, reflecting a learned tendency to stay farther from the corridor walls and thus operate more safely. At the same time, the average angular velocity steadily declines, as the agents come to avoid sharp turns that incur a penalty in the reward function. In contrast, the linear velocity rises continuously, driven by the positive reinforcement for maintaining high forward speed. Notably, both DDPG and TD3

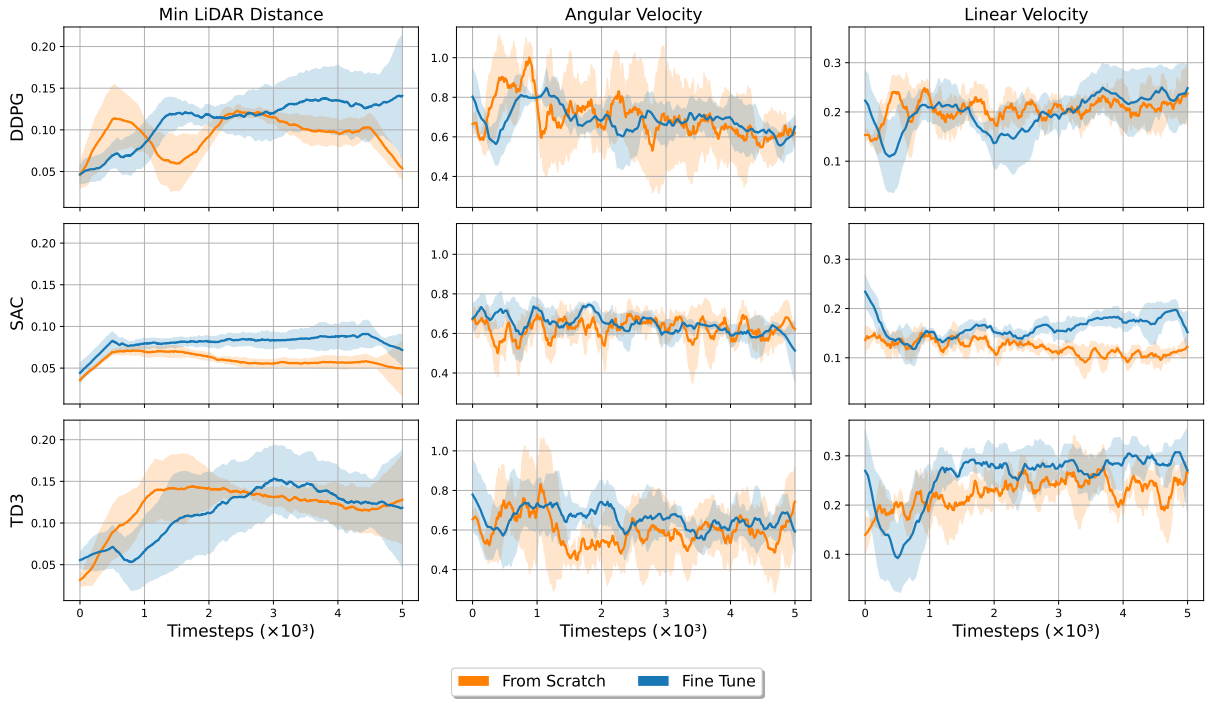

Figure 5.10: Metrics comparison for Task 3.

ultimately reach the imposed speed limit of 0.3 m/s, whereas SAC converges to a more conservative cruising speed, indicating an inability to discover the optimal high-velocity policy under these conditions.

Ultimately, the resulting policies not only complete the 360° turn reliably but also exhibit graceful, human-like motion, slowing down around the curve and accelerating along the straight segments. This task highlights the delicate interplay between classical control principles and modern reinforcement learning, and underscores the importance of careful environment and reward design, even for tasks that appear deceptively simple.

Figure 5.11 presents a qualitative comparison between the best fine-tuned RL trajectories (TD3 and DDPG) and the NAPVIG corridor navigation baseline introduced in Lissandrini et al., 2023. The RL policies maintain a tighter corridor centerline, dynamically modulating speed to preserve safe clearance on bends, whereas NAPVIG follows a fixed-speed profile that can lead to either drift or overly conservative steering in tighter sections.

5.4 Discussion

Across all tasks, fine-tuning accelerates early real-world learning, while it is reasonable that the from-scratch approach yields lower initial performance (due to the lack of prior knowledge). Fine-tuned policies experience a transient dip relative to their simulation-trained peak, yet rapidly recover and sometimes surpass the pre-training performance, as in Fig. 5.1. In conditions where it is not feasible to run a large number of episodes

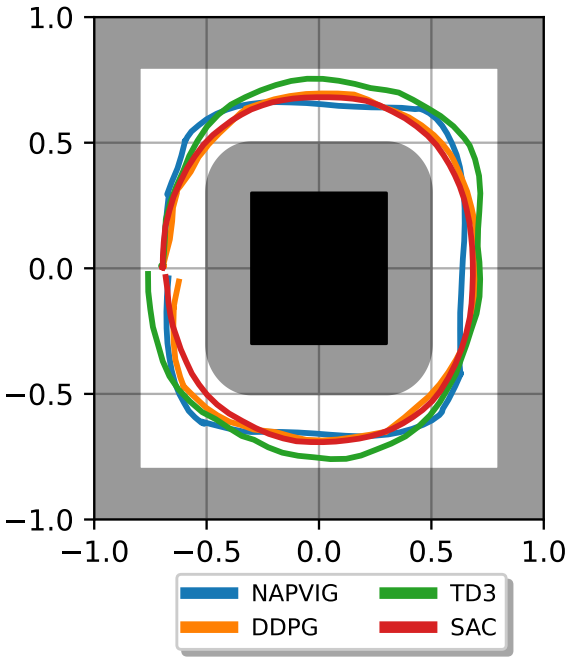

Figure 5.11: Comparison of trajectories for Task 3.

or timesteps in the real world, simulation pre-training followed by real-world fine-tuning proves particularly advantageous.

Pre-trained policies demonstrate significant improvements over those trained from scratch, even when the number of fine-tuning episodes is limited. The performance gap between the two approaches varies significantly across tasks (as shown in Fig. 5.12), but it is clear that simulation pre-training provides substantial gains, and that subsequent deployment and fine-tuning further boost real-world performance. In particular, update frequency plays a pivotal role: higher update rates magnify the benefits of pre-training. Although SAC's relatively slow update rate of 3.3 Hz hinders its ability to fully exploit simulation-derived knowledge and causes it to underperform compared to DDPG and TD3 on every task, lighter algorithms such as DDPG and TD3 benefit from higher update frequencies, enabling them to leverage pre-training more effectively and consistently achieve the best results in our experiments.

These observations underscore the critical interplay between simulation pre-training, fine-tuning strategies, and the computational characteristics of learning algorithms when aiming for efficient, robust sim-to-real transfer under tight resource constraints.

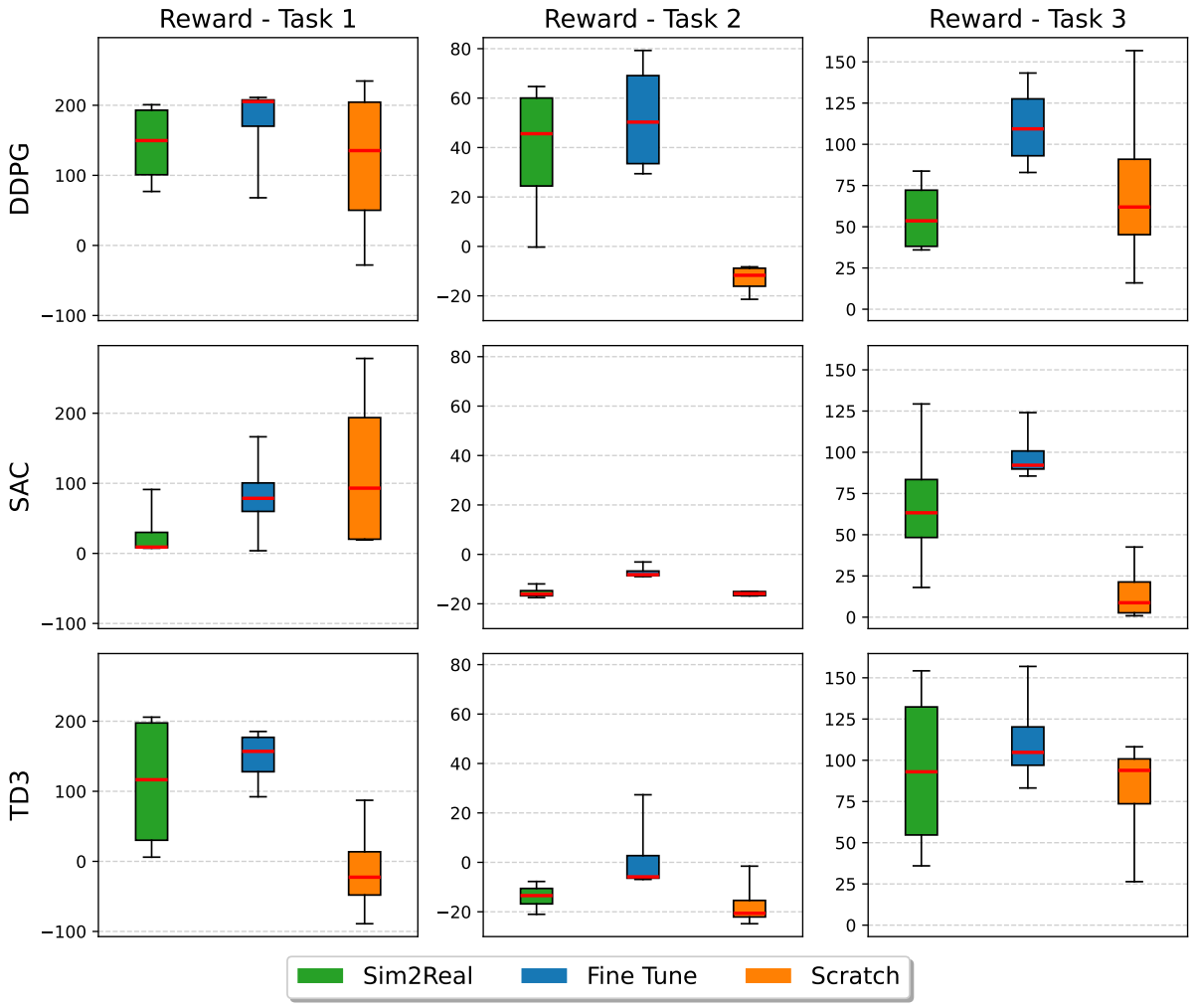

Figure 5.12: Final Policies Evaluation across all Tasks.

Chapter 6

Conclusions

This study has investigated the application of Deep-RL algorithms: DDPG, TD3, and SAC for autonomous navigation tasks, with a focus on their "Sim2Real" transferability on resource-constrained TurtleBots. A structured methodology progressing from simulation to real-world deployment is employed to highlight the practical trade-offs between algorithmic complexity and computational feasibility. Results show that simulation pre-training followed by real-world fine-tuning provides a significant advantage in learning efficiency compared to training from scratch and algorithms with higher control frequencies (DDPG and TD3) can perform better than slower ones (SAC). These findings underscore the importance of lightweight yet effective algorithms for real-world robotic applications and point to promising directions for future work to close the "Sim2Real" gap.
As part of these efforts, a novel lightweight algorithm, recently theorized within the DEI, will be evaluated on similar navigation tasks to asses its on-device learning capabilities Sinigaglia et al., 2024.

Appendix A

Preliminary Simulation Studies

A.1 Task 1

Initial tests were conducted in a simplified 2D Gym environment to validate the reward functions, neural network architecture, and training stability. These experiments provided the foundational framework prior to advancement to a 3D Gazebo simulation.

Reward Shaping & Hyperparameters Selection

The objective of this phase was to develop a reward function capable of directing the agent toward the target while avoiding the rectangular obstacle at the origin, and to achieve this in minimal time. Several reward formulations were implemented and evaluated to determine the most effective approach, facilitating also the fine-tuning of critical hyperparameters, specifically network width and batch size.

Reward shaping began with a basic distance-based component that encouraged the agent to approach the goal. This was gradually refined to include penalty terms for collisions and bonus rewards for proximity to the target.

To further encourage time efficiency, step-wise penalties were introduced for prolonged episodes, while small positive shaping rewards were added to reinforce incremental progress toward the goal and mitigate the risk of the agent becoming stuck in local optima. After extensive empirical testing a baseline reward was found to offer the best balance between training stability and exploratory behavior.

Performance was quantified by the cumulative reward accrued over elapsed simulation time (hours). In the initial prototyping stage, real-time constraints were not enforced: the environment operated at 100 Hz to maximize experimental throughput. A fully connected actor–critic network with a single hidden layer of 256 neurons was sufficient to confirm the validity of the state and reward definitions, and to demonstrate basic policy convergence (see Fig. A.1 (above)).

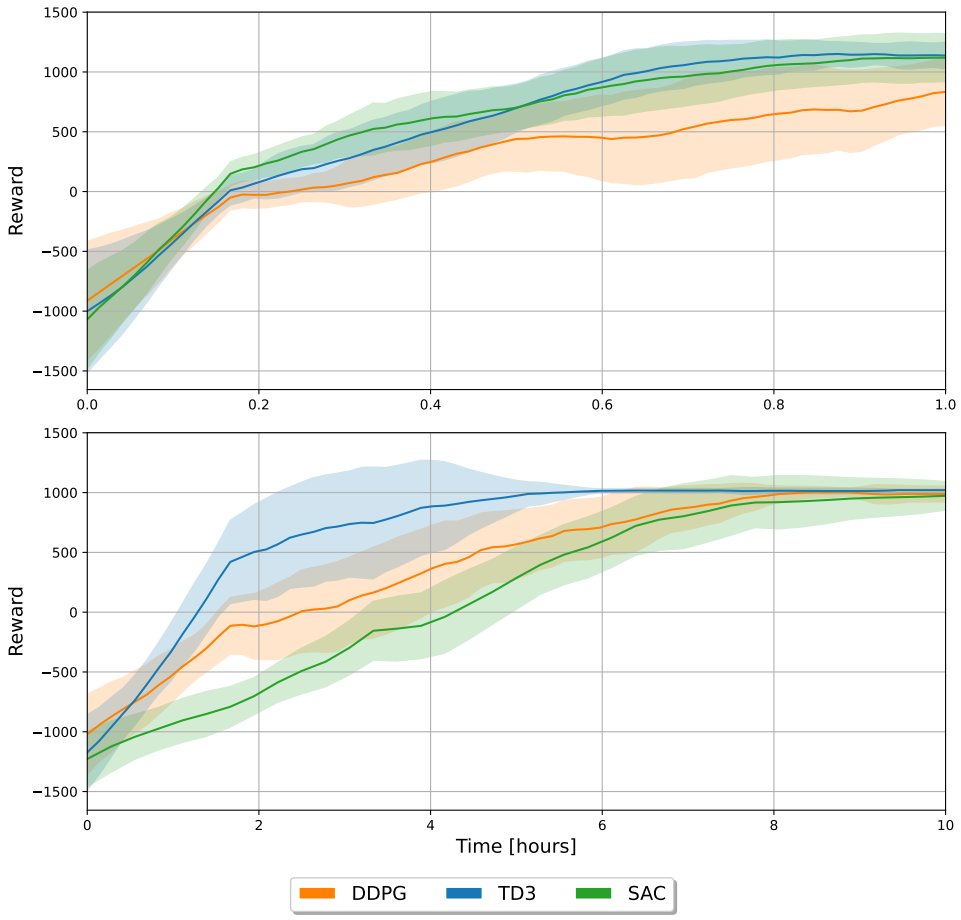

Figure A.1: Reward curves at 100 Hz (above) and at reduced update frequencies (10 Hz for DDPG and TD3; 5 Hz for SAC (below); network width 256, batch size 256).

Subsequently, the control update frequency was reduced from 100 Hz to 10 Hz for DDPG and TD3, and to 5 Hz for SAC, while preserving the previously defined architecture and hyperparameters. The results depicted in Fig. A.1 (below) illustrate the impact of lower update rates on learning performance: training durations increased by an order of magnitude compared to the 100 Hz baseline, and the performance ranking of the algorithms was altered under these conditions.

This degradation highlighted the sensitivity of off-policy deep reinforcement learning algorithms to update frequency and sampling resolution. At lower frequencies, fewer samples are collected per unit time, diminishing the learning signal and increasing the time needed to converge.

The feasible control frequencies achievable on a Raspberry Pi 3B were then evaluated. Network width and batch size were varied to assess their effects on training performance across model configurations. Specifically, network widths of 64 and 32 were tested in conjunction with batch sizes ranging from 32 to 128.
The primary criteria for selection were convergence speed, final reward plateau, and sta-

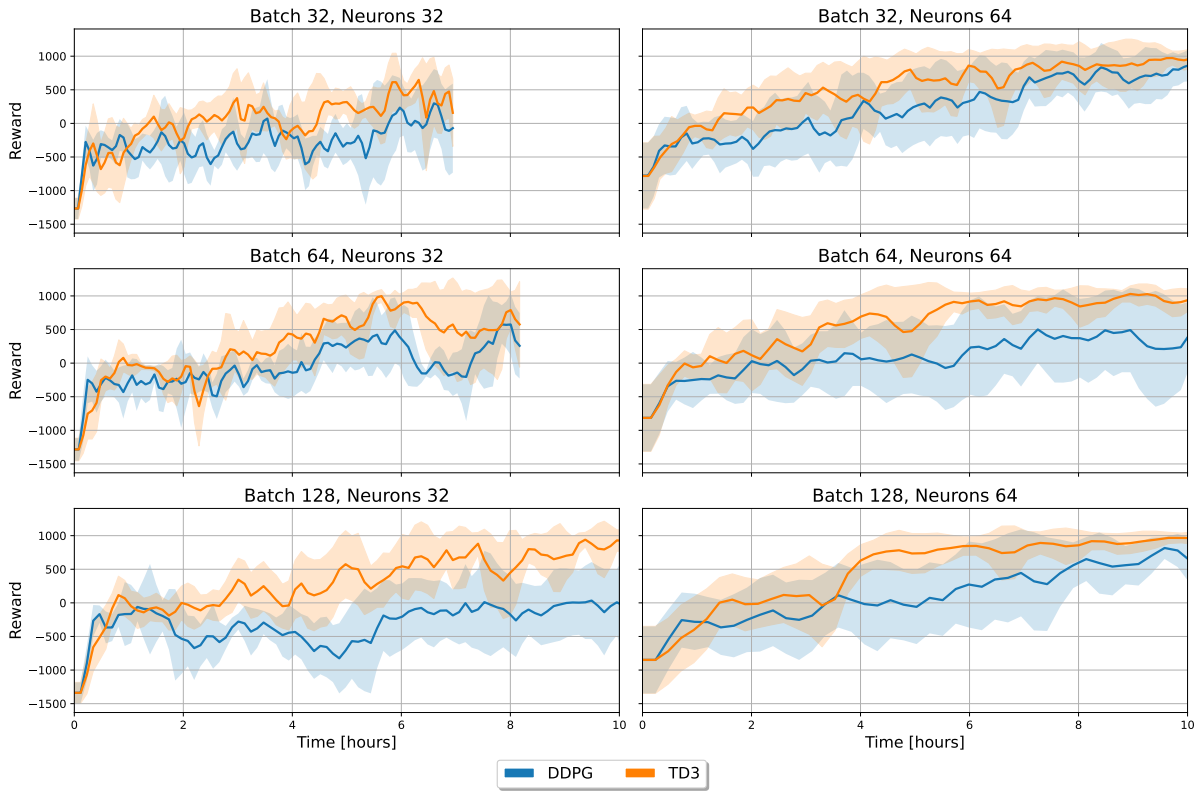

Figure A.2: Effect of network width and batch size on training stability and convergence speed. Each subplot shows the mean and standard deviation of rewards across multiple runs, with increasing batch sizes from top to bottom and increasing network widths from left to right.

bility across multiple training seeds. Network width and batch size were observed to have a significant influence on learning dynamics. Figure A.2 presents an overview of the effect of varying network widths and batch sizes on training progression and stability.

Based on these observations, a network configuration with 64 neurons per hidden layer and a batch size of 128 was selected. This combination offered a practical balance between expressiveness, convergence reliability, and computational efficiency suitable for real-time execution on Edge devices.

The insights from this reward and architecture tuning stage formed the basis for scaling the approach to the Gazebo environment.

A.2 Task 2

In the preliminary design stage, several alternative trajectories were evaluated, and a spatially-variant reward function was devised to encourage the robot to remain close to a desired planar path. Let

$$\mathcal{T} = \{\mathbf{p}_i = (x_i, y_i) \mid i = 1, \ldots, N\} \tag{A.1}$$

be the set of N points discretizing the target trajectory (in our case, a sinusoid of amplitude A over $[-0.5, 0.5]$). For any robot position $\mathbf{x} = (x, y)$, we compute a localized Gaussian reward over the k nearest trajectory points.

$$d_i(\mathbf{x}) = \big\|\mathbf{x} - \mathbf{p}_i\big\|, \tag{A.2}$$

$$g_i(\mathbf{x}) = \exp\!\left(-\frac{d_i(\mathbf{x})^2}{2\sigma^2}\right), \tag{A.3}$$

where $\sigma > 0$ controls the width of the Gaussian kernel. Let

$$\mathcal{N}_k(\mathbf{x}) = \arg\operatorname*{sort}_i \; d_i(\mathbf{x})\,[1\!:\!k] \tag{A.4}$$

denote the indices of the k trajectory points closest to $\mathbf{x}$. The final reward is then

$$R(\mathbf{x}) = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x})} \exp\!\left(-\frac{\|\mathbf{x} - \mathbf{p}_i\|^2}{2\sigma^2}\right). \tag{A.5}$$

For this implementation the parameters used were:

$$\sigma = 0.05, \qquad k = 20, \qquad N = 10\,000, \tag{A.6}$$

with $\mathcal{T}$ given by

$$x_i = -1 + \frac{2(i-1)}{N-1}, \quad \mathbf{p}_i = \big(x_i,\, A\sin(\omega x_i)\big), \quad \omega = 2\pi, \quad A = 0.5. \tag{A.7}$$

The intuition behind this reward function is simple yet effective. When the robot's position $\mathbf{x}$ coincides exactly with one of the trajectory points $\mathbf{p}_i$, the corresponding distance $d_i(\mathbf{x})$ becomes zero, yielding a Gaussian response $g_i(\mathbf{x}) = 1$ and thus driving the average reward $R(\mathbf{x})$ toward its maximum value of approximately one.

As the robot moves away from the path, each distance $d_i(\mathbf{x})$ grows, causing the Gaussian terms to decay smoothly; the aggregate effect is a smoothly decreasing reward as a function of the robot's deviation from the trajectory. By restricting the sum to the k

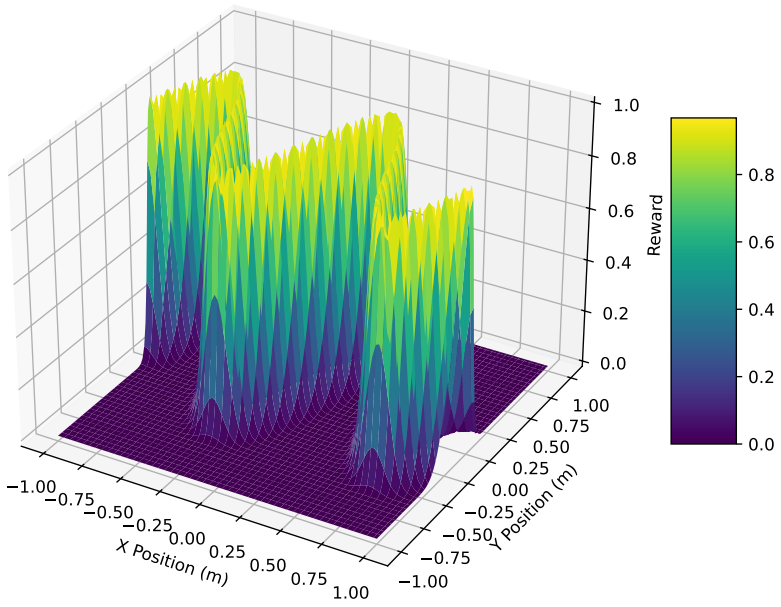

Figure A.3: 3D plot of reward function for Task 2.

nearest trajectory points, the reward computation remains localized: only the segment of the path closest to the robot influences the score, preventing distant parts of the curve from diluting the sensitivity of the function.

Figure A.3 illustrates how these properties combine to produce a sharp, ridge-like reward surface along the desired path and a gradual drop-off on either side.

Appendix B

Other Considerations

The majority of the development effort was devoted to designing and implementing the underlying software and mechanical architecture required to execute all necessary benchmark routines. However, during the transition from simulation to real-world hardware, two primary challenges emerged: the structural durability of the robot and its onboard energy source.

Structural Durability The robotic platform under investigation was originally engineered for short-duration demonstrations rather than extended operational tasks. When subjected to prolonged or intensive use, the excessive load exerted on motor gears and drive shafts led to frequent mechanical failures. In particular, repeated high-torque maneuvers induced gear tooth wear and shaft misalignment, resulting in gear slippage and, ultimately, component breakage. To mitigate these issues, we maintained an extensive stock of replacement gears and shafts. Nonetheless, these ad-hoc repairs introduced downtime and limited the continuous operating cycles we could achieve. A brand-new motor withstood roughly 15 hours of cumulative operation over a period of 1 to 1.5 weeks before exhibiting signs of mechanical fatigue and failure. In Fig. B.2 the first side effects of extensive real world testing.

Energy Source Constraints We equipped the robot with a lithium–polymer (LiPo) battery pack, chosen for its high energy density and compact form factor. Unfortunately, due to both the inherent degradation of LiPo chemistry over time and the specific discharge rates demanded by our benchmark routines, each fully charged battery sustained only about one hour of operation before voltage levels fell below safe thresholds. Consequently, each training session on the physical hardware was limited to roughly 60 minutes to ensure optimal performance, as the battery could potentially last longer but not without risking suboptimal operation and the possibility of failing to complete the predetermined episodes. This frequent interruption not only reduced productivity but also hindered the collection

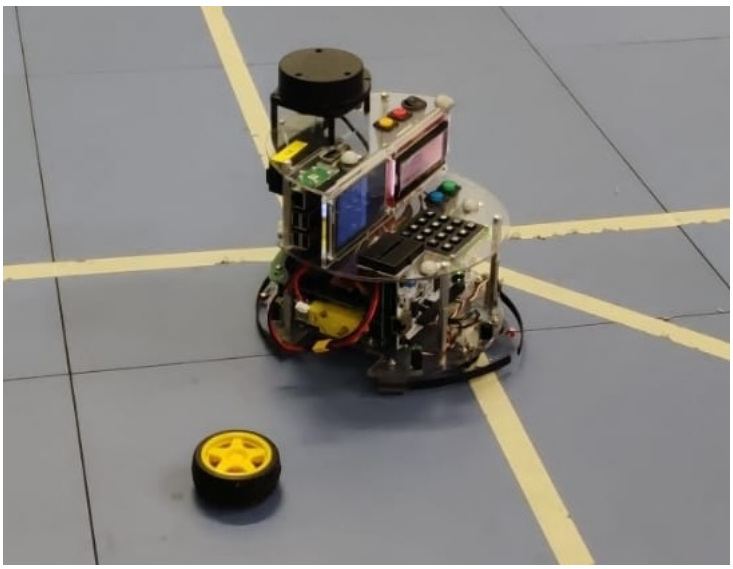

Figure B.1: Intentional shedding of excess components for increased agility.

of long-horizon data, critical for evaluating the stability and convergence properties of our control algorithms.

Together, these hardware constraints significantly curtailed the amount of real-world data we could gather. Whereas our simulations could run uninterrupted for tens of hours, the physical robot required stop–start cycles approximately every hour, not only due to power limitations but also to prevent potential motor damage, as shown in Fig. B.2, introducing manual intervention and extending the overall experimental timeline.

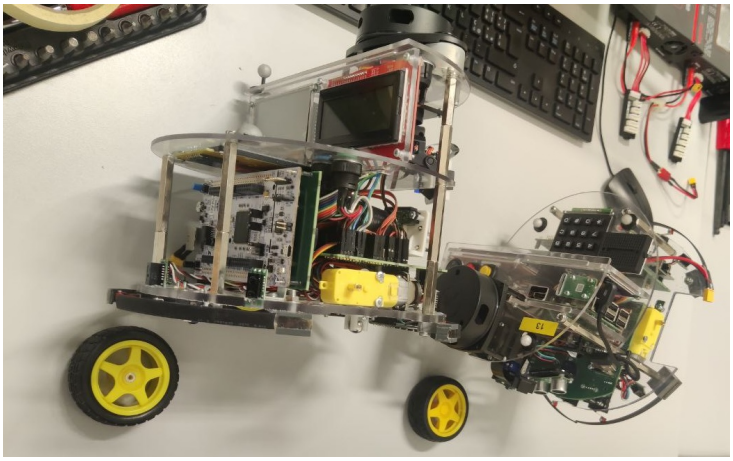

Figure B.2: Pit Stop.

Bibliography

Li, Yuxi (2023). "A comprehensive survey on deep reinforcement learning: Challenges, techniques, and future directions". In: *IEEE Transactions on Neural Networks and Learning Systems*.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Levine, Sergey et al. (2016). "End-to-end training of deep visuomotor policies". In: *Journal of Machine Learning Research* 17.39, pp. 1–40.

Wiebe, Felix, Niccolò Turcato, Alberto Dalla Libera, et al. (Aug. 2024). "Reinforcement Learning for Athletic Intelligence: Lessons from the 1st "AI Olympics with RealAIGym" Competition". In: *IJCAI-24*. Ed. by Kate Larson. Demo Track, pp. 8833–8837.

Wiebe, Felix, Niccolò Turcato, Alberto Dalla Libera, et al. (2025). "Reinforcement Learning for Robust Athletic Intelligence: Lessons from the 2nd 'AI Olympics with RealAIGym' Competition". In: *arXiv preprint arXiv:2503.15290*.

Salvato, Enrico, Gian Fenu, and Eric Medvet (2021). "Crossing the gap: A deep dive into zero-shot sim-to-real transfer for dynamics". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.8, pp. 3679–3693.

Koenig, Nathan and Andrew Howard (2004). "Design and use paradigms for gazebo, an open-source multi-robot simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 3. IEEE, pp. 2149–2154.

Kober, Jens, J Andrew Bagnell, and Jan Peters (2013). "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274.

Kiran, B Ravi et al. (2022). "End-to-end autonomous driving with deep reinforcement learning: Progress and challenges". In: *IEEE Intelligent Transportation Systems Magazine*.

Zhang, Wenke, Chaoyue Chen, Shanghang Zhou, et al. (2021). "An overview of deep reinforcement learning in autonomous driving". In: *IEEE Transactions on Intelligent Transportation Systems*.

Tobin, Josh et al. (2017). "Domain randomization for transferring deep neural networks from simulation to the real world". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, pp. 23–30.

Pinto, Lerrel et al. (2017). "Robust adversarial reinforcement learning". In: *International Conference on Machine Learning*, pp. 2817–2826.

Peng, Xue Bin et al. (2018). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization". In: *IEEE International Conference on Robotics and Automation*. IEEE, pp. 3803–3810.

Sadeghi, Fereshteh and Sergey Levine (2016). "CAD2RL: Real Single-Image Flight Without a Single Real Image". In: *Robotics: Science and Systems*.

Miki, Takahiro et al. (2022). "Learning Robust Perceptive Locomotion for Quadrupedal Robots in the Wild". In: *Science Robotics* 7.62.

Tai, Lei, Giuseppe Paolo, and Ming Liu (2017). "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 31–36.

Kahn, Gregory et al. (2018). "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation". In: *IEEE International Conference on Robotics and Automation*. IEEE, pp. 5129–5136.

Francis, Anthony et al. (2020). "Long-range Indoor Navigation with PRM-RL". In: *IEEE Transactions on Robotics* 36.4, pp. 1115–1134.

Hwangbo, Jemin et al. (2017). "Control of a Quadrotor With Reinforcement Learning". In: *IEEE Robotics and Automation Letters*. Vol. 2. 4. IEEE, pp. 2096–2103.

Kaufmann, Elia et al. (2023). "Champion-level drone racing using deep reinforcement learning". In: *Nature* 611.7937, pp. 928–935.

Haarnoja, Tuomas et al. (2018). "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290*.

Fujimoto, Scott, Herke Van Hoof, and David Meger (2018). "Addressing function approximation error in actor-critic methods". In: *International Conference on Machine Learning*. PMLR, pp. 1587–1596.

Kumar, Ashish et al. (2021). "RMA: Rapid Motor Adaptation for Legged Robots". In: *Robotics: Science and Systems*.

Chen, Guobin et al. (2020). "Learning efficient object detection models with knowledge distillation". In: *Advances in Neural Information Processing Systems*.

Lillicrap, Timothy P et al. (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

ROBOTIS (2020). *TurtleBot3*. `https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`. Accessed: 2023-10-01.

Lissandrini, Nicola et al. (2023). "NAPVIG: Local Generalized Voronoi Approximation for Reactive Navigation in Unknown and Dynamic Environments". In: *ACC-2023*, pp. 28–33.

Sinigaglia, Alberto et al. (2024). *Edge Delayed Deep Deterministic Policy Gradient: efficient continuous control for edge scenarios*. arXiv: 2412.06390 [cs.LG]. URL: https://arxiv.org/abs/2412.06390.