Exploring Multi-Agent Reinforcement Learning for Mobile Manipulation

Master of Science Thesis
University of Turku
Department of Computing
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
2023
Eetu-Aleksi Rantala

Supervisors:
MSc. Wenshuai Zhao
MSc. Xianjia Yu
PhD. Jorge Peña Queralta

UNIVERSITY OF TURKU
Department of Computing

EETU-ALEKSI RANTALA: Exploring Multi-Agent Reinforcement Learning for Mobile Manipulation

Master of Science Thesis, 56 p.
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
October 2023

To make robots valuable in our everyday lives, they need to be able to make good decisions even in unexpected situations. Reinforcement learning is a paradigm that aims to learn decision-making models for robots without the need for direct examples of the correct decisions. For this type of robot learning, it is common practice to learn a single central model that controls the entire robot.

This work is motivated by advances in modular and swarm robotics, where multiple robots or decision-makers collaborate to complete a task. Instead of learning a single central model, we explore the idea of learning multiple decision-making models, each controlling a different part of the robot. In particular, we investigate whether providing the different models with different sensing capabilities helps the robot to learn or to be robust to perturbations. We formulate these problems as multi-agent problems and use a multi-agent reinforcement learning algorithm to solve them.

To evaluate our approach, we design a mobile manipulation task and implement a simulation-based training pipeline to produce decision-making models that can complete the task. The trained models are then directly transferred to a real autonomous mobile manipulator system. Several experiments are performed on the real system to compare the performance and robustness against the usual central model baseline. Our experimental results show that our approach can learn faster and produce decision-making models that are more robust to perturbations.

Keywords: reinforcement learning, multi-agent system, robot learning, mobile manipulation, robustness

Contents

List of Figures

List of Tables

List of Acronyms

RL	Reinforcement learning
SARL	Single-agent reinforcement learning
MARL	Multi-agent reinforcement learning
MDP	Markov decision process
VPG	Vanilla Policy Gradient
PPO	Proximal Policy Optimization
MMDP	Multi-agent Markov decision process
Dec-POMDP	Decentralized partial-observable Markov decision process
CTDE	Centralized training and decentralized execution
IPPO	Independent Proximal Policy Optimization
MAPPO	Multi-agent Proximal Policy Optimization
ROS	Robot Operating System
FCI	Franka Control Interface
MLP	Multi-layer perceptron
ONNX	Open Neural Network Exchange

1 Introduction

One of the most important challenges in artificial intelligence is to create machines that learn to act autonomously from interactions, especially when the optimal behavior is not known in advance. Reinforcement learning (RL) is a principled framework for addressing these kinds of problems [1]. It is a form of machine learning that differs from supervised learning in that there are no direct examples of how to behave in a given situation, only a reward signal that may be delayed in time. In RL, behavior is learned through a trial-and-error process in which different behaviors must be explored to find the ones that lead to the highest reward.

Before the deep learning (DL) boom in the 2010s, RL was limited to low-dimensional problems and lacked scalability due to the computational and sample complexity of high-dimensional problems [2]. After DL became feasible, it provided RL researchers and practitioners with powerful representation learning and function approximation properties of deep neural networks, resulting in so-called deep RL. This allowed RL methods to overcome many of the scalability problems, leading to many impressive applications such as superhuman play in the board games Go and Chess [3] and even in video games from Atari to StarCraft II [4], [5].

In addition to games, RL is used to train neural network controllers for complex robots. The neural network represents a *policy* that specifies the behavior of the robot in a given situation. Interactive learning with live robots is not feasible due to time, cost, and safety considerations. Therefore, the controller is often first trained in a virtual envi-

ronment provided by a simulator. The trained neural network can then be used on a real robot. This so-called sim-to-real approach has made RL a popular paradigm for training robot controllers, especially for legged robots [6].

Simulator-based training isn't without its own challenges. There is a gap between the fidelity of the simulation and the real world. It is impossible to perfectly model the robot's sensing and actuation, or all the unpredictable situations it might encounter. To be successful, robots must be able to generalize to differences between simulation and the real world and be robust to unexpected situations.

In its basic form, RL is concerned with learning for a single agent, i.e., a decision maker. For modeling multiple agents, there exists a subfield of RL called multi-agent RL (MARL). It is concerned with the learning of multiple independent agents interacting with the same world and with each other. The agents can cooperate towards a common goal or they can compete against each other. For robot learning, MARL is not yet as widely adopted an approach as RL. Although it has seen some use for controlling multiple robots in a swarm configuration [7], [8], it has not seen much use in cases where multiple agents control a single robot.

Modeling and controlling a single robot as a multi-agent system is a prevalent area of research in swarm robotics or modular robotics [9], [10]. However, the literature on multi-agent reinforcement learning for such systems is sparse [11]. While there are clear advantages in terms of robustness from using multiple agents cooperating toward a common goal, there is a lack of studies on the performance and quality of single-agent RL versus multi-agent RL approaches. In addition, a new problem arises in such systems when all information previously available to the single agent is now equally available to all sub-agents. In these cases, agents may not need all the information and could benefit from accessing only partial information from local sensing. Finally, it is not well understood how these aspects affect the transfer of control policies from simulation to reality.

In this thesis, we explore modular modeling in RL and show that such methods com-

bined with only local sensing can lead to faster learning and more robust behavior in a real-world mobile manipulation setting.

1.1 Motivation and Significance

This thesis is motivated by the limitations described above in the areas of robust robot learning and the advances in multi-agent learning for complex robots, which are typically modeled as single agents. In particular, the specific topic of this thesis is inspired by recent works on multi-agent MuJoCo and end-to-end policy learning of mobile manipulator [11], [12]. In general words, this thesis is motivated by the problem of learning independent controllers for different parts of a robot with different sensing capabilities that interact with each other, cope with failures, and contribute towards a common goal.

The methods in this thesis make use of multiple collaborating agents within a single robot to increase its generalization and robustness. These methods can lead to more fault-tolerant robots that can function even if some part of the robot malfunctions. In addition, RL for robotics has focused more on single-agent scenarios than multi-agent scenarios. In cases where multiple agents are used, each agent controls a different independent robot. Therefore, it is important to explore the advantages of modular modeling with MARL. Finally, to the best of our knowledge, using MARL to increase the robustness of a single robot is a novel research direction.

1.2 Design Principles and Requirements

To guide the design of our proposed methods, we define a number of principles. First, the methods should be general enough to work with different types of robots and different numbers of agents. Second, the training should not depend on the use of specific simulation or RL software. Finally, the methods should follow a learning approach that produces control policies that are robust to perturbations in the control, such as the malfunctioning

of some of the agents.

In addition, we define requirements for the implementation. On the hardware side, we evaluate our methods with a mobile manipulator built with a Clearpath Husky ground robot and a Franka Emika Panda manipulator arm. The robots are chosen based on availability. On the software side, we use the Nvidia Isaac Sim platform to simulate the virtual training environment and set up the workflow for the sim-to-real transfer. Isaac Sim is chosen because it supports RL training out of the box and provides high performance and scalability thanks to its GPU-based parallelization. We use the Robot Operating System ecosystem to implement the real robot system. The Robot Operating System is used because it is the de facto robotics software stack.

1.3 Research Questions

Based on the above design principles and requirements, the thesis is driven with the objective of answering the following set of research questions:

1. Is it possible to use MARL to train policies for multiple agents controlling a single robot and deploy them in a real-world mobile manipulator?

2. Is zero-shot sim-to-real transfer possible for this multi-agent system?

3. Do the multi-agent policies perform better or more robustly than the single-agent counterpart?

Based on these research questions, we hypothesize that our MARL approach will improve the overall robustness of the system to anomalies in the behavior of some of the agents.

1.4 Contributions

In this thesis, we study how MARL can be used to learn robust control policies for single robots. The main contributions are listed below:

1. Propose two methods for dividing the control of a single robot among multiple cooperating agents which are subsequently trained via MARL.
2. Implement a MARL-based training scheme in Isaac Sim to train control policies using the proposed methods.
3. Design evaluation scenarios to measure the effectiveness of the proposed approaches.
4. Evaluate the trained policies on a real mobile manipulator system, demonstrating viable sim-to-real transfer and the benefit of local sensing for increased robustness.

1.5 Related Works

In recent years, RL has been used in many applications for real robots. Due to the large amount of interaction required for learning and the potential safety risks of training on a real robot, learning is usually performed by first training a control policy in a simulated environment and then deploying the learned policy on the real robot.

Hwangbo et al. [13] used a simulator to learn a robust control policy for a legged robot. The input observations for the policy included the robot's pose and velocity, and joint states such as the position and velocity of each leg joint. The output of the policy was the desired joint positions. To reduce the gap between simulation and reality, the authors used an approach where they modeled the dynamics of each joint's actuator with neural networks and used them during training. This learning approach allowed them to produce policies that enabled the real robot to perform a wide variety of locomotion skills, from fast running to recovering from falls.

In [14], the authors used RL to learn a policy that allowed a robotic humanoid hand to manipulate a colored cube to a target configuration using visual information. They trained the policy in many parallel simulations with randomized settings and visuals to bridge the reality gap between simulation and the real world. After training, they successfully applied the policy to a real robotic hand, although the performance was worse than in the simulation.

More recently, Haarnoja et al. [15] used RL to learn agile soccer skills for bipedal robots. They first trained two teacher policies in simulation, one for playing soccer and another for recovering from falls. They then used a method called policy distillation to combine these two policies into a single policy. The combined policy was then made to play against earlier versions of itself to learn to play two-player soccer. The trained policy was transferred zero-shot to real robots and demonstrated a variety of soccer skills, from shooting to recovering from falls.

MARL research has mostly focused on simulated tasks. For example, Huang et al. [8] used a MARL algorithm to locate a mobile target with radio signals using a swarm of drones. They provided simulation-based results showing that this algorithm can converge to the optimal solution and outperform standard approaches. In addition to simulation-based research, some MARL studies have been conducted with real robots. In one study, Candela et al. [16] used a MARL algorithm to learn a policy that can autonomously drive around a track while avoiding obstacles. The policy was first trained in a simulator with added domain randomization and then successfully transferred to real robots.

There is a mismatch between simulated and real environments that needs to be addressed. Policies must be able to generalize to unseen environmental states and be robust to unexpected events. For example, in [6], a locomotion policy trained in simulation was able to control the robot in a variety of natural environments that were never encountered during training. The authors attributed their success to the following three ingredients. First, they switched from a feed-forward neural network architecture to a sequential ar-

chitecture and provided a history of proprioceptive signals as input. Second, they used an approach called privileged learning. In this approach, a teacher policy was trained with additional information that was only available in the simulator. They then trained a student policy that had access to the same limited information as the real robot. Third, they implemented an automatic curriculum that adapted the environment to become more challenging as the neural network learned. Although the controller was trained in environments with only rigid structures, it was able to handle deformable terrain such as mud and dynamic footholds, as well as situations where the ground was obstructed by dense vegetation or water.

Learning mobile manipulation through RL has been studied in several works, but they have been limited to single-agent tasks. In [17], an RL algorithm PPO was used to train a mobile manipulator system for picking up objects. The training was done in simulation and the system was deployed on a real robot. During training, the object poses were given directly by the simulator, while during deployment, the object poses were estimated from the camera. For the real-world deployment, the Robot Operating System (ROS) was used to control the mobile base and the manipulator.

More recently, Fu et al. [12] used RL to learn a whole-body controller for a legged mobile manipulator. The controller was trained to simultaneously bring the end effector of the arm to the desired pose and move the robot at the desired velocity. The controller was trained in the Isaac Gym simulator and output the desired torques for each of the joints in the legs and arm.

An important inspiration for this work comes from Peng et al. [11], who introduced Multi-Agent MuJoCo, a simulation-based benchmark for continuous cooperative multi-agent robot control. The scenarios of this benchmark involve training multiple agents to control different parts of a single robot. The authors proposed an approach where a single robot is represented as a graph consisting of the robot's body parts and connecting joints. The graph is divided into disjoint subgraphs, one for each agent. Each subgraph contains

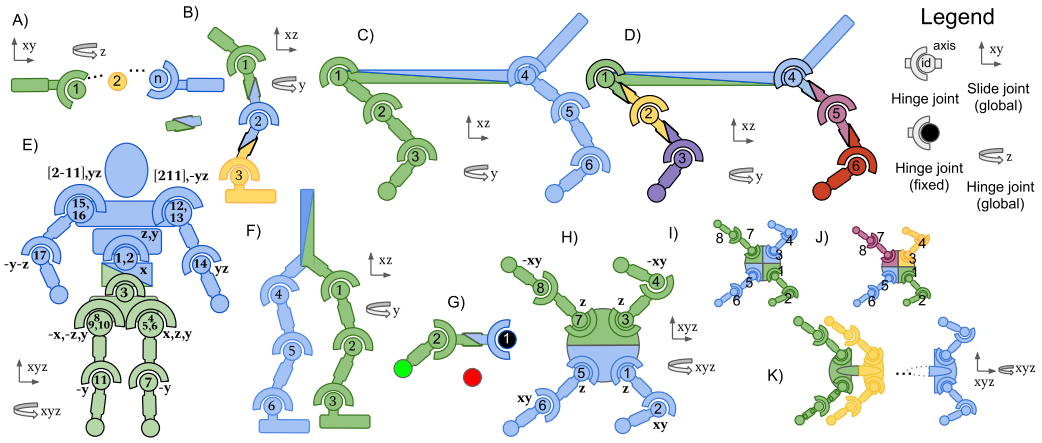

Figure 1.1: Illustration of the Multi-Agent MuJoCo, where each robot can be controlled by several independent agents, visualized here in different colors [11].

one or more joints that the corresponding agent could control. The agents always observe their own subgraph, and can be configured to observe other subgraphs that are within distance $k \geq 0$ from the agent's own subgraph. The approach of Multi-Agent MuJoCo is illustrated in Figure 1.1. The authors used this approach as a benchmark for MARL algorithms. Similar to our work, they use multiple agents to control a single robot, but unlike our work, they focused on creating a benchmark to compare MARL algorithms and didn't have scenarios to test the robustness of the robot to failures of some of the agents. They also focused on simulation and didn't do any policy transfer to a real robot.

1.6 Structure

This thesis has the following structure:

- Chapter 2 introduces RL in both single-agent and multi-agent settings and generalization in the context of sim-to-real.
- Chapter 3 describes the software and robots used in this thesis.
- In Chapter 4, we define our methods for sharing control of the robot among multiple

cooperating agents.

- In Chapter 5 we describe in detail our implementation of the methods, the training, and the real-world evaluation setup.
- Chapter 6 presents the experiments and results.
- Finally, Chapter 7 concludes this thesis and outlines future work directions.

2 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning in which an agent maximizes long-term accumulated rewards by interacting with its environment [18]. The agent has no examples of correct actions to take and must learn the optimal behavior through trial and error from the rewards it receives. An important feature of RL is that the reward can be delayed in time. This means that the action the agent takes at one moment may not be rewarded immediately, but only after some time. Thus, the agent must learn to associate rewards with behaviors that have long-term effects. In addition, the agent must strike a balance between exploration and exploitation. Exploration refers to trying new behaviors that lead to unexplored environmental states that may potentially lead to higher rewards. Exploitation, on the other hand, refers to performing familiar behaviors and visiting environmental states that have already been explored and currently have the highest associated reward.

2.1 Single-agent Reinforcement Learning

Markov Decision Processes

In the classical view of RL, there is only one agent interacting with the environment [2], [19]. This interaction can be expressed mathematically as a Markov Decision Process (MDP), a model for sequential decision making. It describes the environment, how it changes based on the agent's actions, and the rewards the agent receives. An MDP con-

sists of

- a set of states (state space) $\mathcal{S}$,
- a set of actions (action space) $\mathcal{A}$,
- a transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ denoting the probability of transition from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ given action $a \in \mathcal{A}$,
- a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ that determines the reward the agent receives for transitioning from state s with action a to the next state s',
- a discount factor $\gamma \in [0, 1]$ that controls the trade-off between current and future rewards.

States, actions, and rewards are the core concepts of an RL problem. States represent the information from the environment that the agent needs to make informed decisions. Actions, on the other hand, represent the choices the agent can make to interact with and change the environment. Finally, rewards guide the agent to improve its behavior and are used to define the goal of the task.

In an MDP, the agent interacts with the environment in discrete time steps t. At each time t the agent observes the environment state s_t and based on that chooses an action a_t to perform. This causes the environment to transition to the next state $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$. After the transition, the agent receives a reward $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$. The agent continues to interact with the environment until a terminal state is reached at time T. After reaching the terminal state, the environment can be reset and the interaction can be restarted. The segment of interaction from the beginning to time T is referred to as an *episode*, and the sequence of states, actions, and rewards is referred to as the agent's *trajectory*. The interaction within an MDP is shown in Figure 2.1.

MDPs have three important properties or assumptions. First, each state in an MDP has the Markov property. This means that future states depend only on the current state

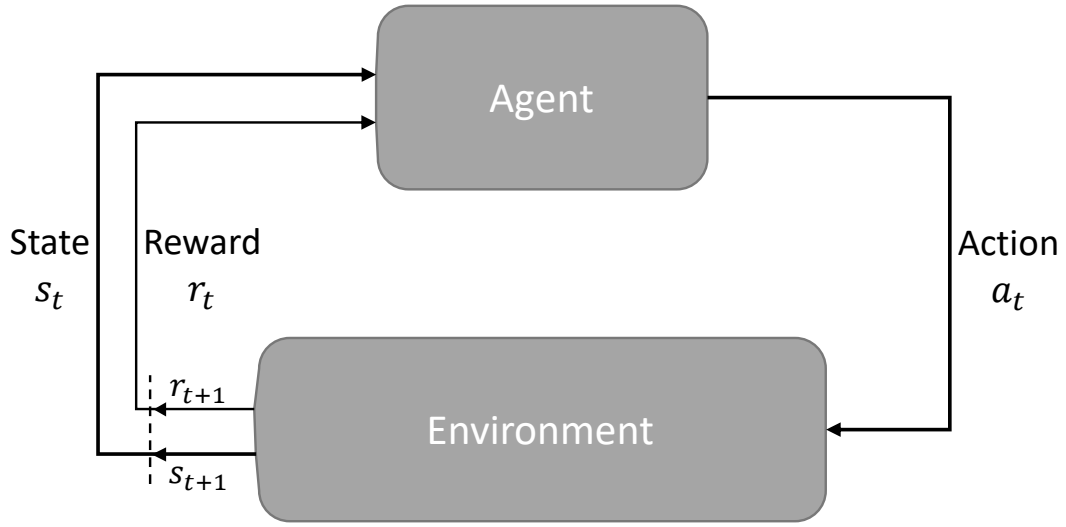

Figure 2.1: Agent-environment interaction in a Markov decision process

and are independent of any previous states or actions taken. In other words, the current state is sufficient for optimal decision making. RL algorithms for solving MDPs assume that the Markov property holds, and may not perform well if it is violated. Therefore, when formulating a task as an MDP, it is important to include sufficient information in the states. Second, MDPs assume full observability. An MDP with full observability means that the agent has complete knowledge of the current state of the environment. If some information is hidden, we have a partially observable MDP. Finally, MDPs assume stationarity of the environment. This refers to the fact that the transition probabilities and the needs of the reward function remain unchanged over time. If the stationarity assumption is violated, the agent may learn an optimal behavior and later, when some aspect of the environment changes, the same behavior is no longer optimal.

Policy

In RL, the behavior of an agent is determined by a policy. A policy π is a function that maps states to actions. An agent uses its policy to choose which action to take given the

current state. A policy can be either deterministic or stochastic, where the former maps each state to a single action, and the latter maps each state to a probability distribution over actions.

In deep RL, the policy is specified by parameters θ, which are usually the weights and biases of a neural network. The specific parameterization of the policy depends on whether the action space is discrete or continuous. If the action space is discrete, it is common to use a neural network that outputs the probability of each action. For continuous actions, the neural network outputs the mean of a Gaussian distribution, and a separate set of parameters is used to specify a diagonal covariance matrix for the distribution. This distribution is then sampled to obtain the action for each dimension in the action space.

Return

The performance of a policy is measured as a return. It is defined as a discounted sum of the rewards from time t onwards:

$$R_t = \sum_{k=0}^{T} \gamma^k r_{k+t}.$$

The discount factor γ determines the relative weight of future rewards in an agent's decision-making process. When γ is closer to 0, the agent becomes more shortsighted and prioritizes immediate rewards, while a value closer to 1 indicates that the agent assigns a higher value to future rewards.

The goal of RL is to learn an optimal policy that maximizes the expected return from all states:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \, \mathbb{E}_\pi \left[R_t \right].$$

2.2 Methods for Reinforcement Learning

There are several different approaches to using RL to solve MDPs. These RL methods or algorithms can be broadly categorized into two groups: value-based methods and policy-

based methods. This categorization is based on whether the algorithm optimizes a value function or a policy. In this section, we briefly review these two approaches.

2.2.1 Value-based Methods

When solving RL problems, it is often useful to assign values to states that are encountered during the interaction. The value of a state is the measure of how much return the agent is expected to get after visiting the state. Value-based methods aim to improve the policy by learning value functions.

There are two different value functions: the state-value function and the action-value function. The state value function describes how good it is to be in a particular state. It is defined as the expected discounted return that is obtained from being in a state s and then following the policy π:

$$V_\pi(s) = \mathbb{E}_\pi \left[R_t | s_t = s \right].$$

The state value function does not specify what action the agent will take after the state. If the policy is stochastic, the state value function specifies only the expected return, not the value of any particular action. Therefore, it cannot be used to select the best action. For that, we need the action value function.

The action value function describes how good it is to take a particular action in a particular state. It is defined as the expected discounted return from being in a state s, performing an action a, and then following the policy π afterward:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[R_t | s_t = s, a_t = a \right].$$

The action value function *can* be used to select the best action. The best action can be selected by testing which action has the highest action value: $a = \operatorname{argmax}_a Q(s, a)$.

It is possible to use the action value function to improve the policy. By selecting the best action, the policy is improved. This improves the return. The improved return can then be used to update the action value function. This process can continue until the

optimal action value function Q_{π^*} is reached. The optimal policy π^* can be retrieved by using Q_{π^*} to select the best action in each state. This iterative process of alternating between updating the action value function and improving the policy is called policy iteration, and it is the foundation of value-based methods [1].

The value functions are usually represented by a function approximator, such as a neural network. For example, the value function of an action can be represented by ϕ and is denoted by $Q_\phi(s, a)$. The value function can then be modified by updating the parameters. There are several algorithms for learning the parameters of an action value function. For example, in Deep Q-learning [4], the parameters are learned iteratively by minimizing the sequence of loss functions, where the ith loss function is defined as:

$$L_i(\phi_i) = \mathbb{E}\left[r + \gamma \max_{a'} Q_{\phi_{i-1}}(s', a') - Q_{\phi_i}(s, a)\right]^2.$$

2.2.2 Policy-based Methods

Policy-based methods follow an approach where the policy is optimized directly [20]. The policy is often represented by a function approximator with parameters θ. The parameterized policy π_θ is a mapping from the state space to a probability distribution over the actions. The probability of choosing an action a in a state s given the parameters θ is denoted as $\pi_\theta(s|a)$. The policy can be changed by modifying the parameters. Policy-based methods aim to optimize the policy parameters via gradient ascent with respect to the objective, which is defined as the expected discounted return per episode:

$$J^{PG}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R_t | s_t = s_0\right]$$

The policy gradient theorem states that the gradient of this objective, called the policy gradient, can be expressed as follows:

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

REINFORCE, also known as the Vanilla Policy Gradient (VPG), is a policy-based method in which the policy gradient is estimated via Monte Carlo sampling, and the

parameters θ are optimized to make the agent more likely to choose actions that lead to higher returns. The gradient can be estimated by collecting a trajectory using the policy π_θ and computing the estimate as

$$\hat{g} = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t.$$

Next, the parameters can be updated in the direction of the estimate

$$\theta_{i+1} = \theta_i + \alpha \hat{g}.$$

The estimated gradient is unbiased but often has a large variance. It is possible to reduce this variance by subtracting a baseline from the return. A common choice for the baseline is the learned value function V_ϕ which is a near-optimal choice for reducing the variance. The pseudocode for VPG with baseline is provided in Algorithm 1.

VPG with a learned value function baseline is an example of an Actor-Critic method. This is a group of methods that combine policy-based methods with learned value functions to address some of the limitations of pure policy-based methods, such as high variance. Examples of actor-critic methods include A2C, PPO, TD3, and SAC. In the next section, we take a closer look at PPO as it is the algorithm we use in this thesis.

2.2.3 Proximal Policy Optimization

Standard policy gradient methods, such as VPG, only update policy parameters once per sample, as more frequent updates can result in policy updates that are too large, causing policy performance to collapse. Proximal Policy Optimization (PPO) is a policy-based method that improves on this by allowing multiple steps of gradient ascent per policy update [21]. PPO does this by following the idea of Trust Region Policy Optimization (TRPO), which maximizes a surrogate objective that constrains the size of the policy update by limiting how different the new and old policies can be. In this way, TRPO achieves a monotonic improvement in policy performance at each optimization iteration.

Algorithm 1 Vanilla Policy Gradient with Baseline

Require: Policy parameters θ, Value function parameters ϕ

1: **for** iteration $= 0, 1, \ldots$ **do**

2: Collect a set of trajectories $D = \{\tau_0, \ldots \tau_N\}$ by running the policy π_θ in the environment for T timesteps

3: Compute returns R_t

4: Estimate the policy gradient as

$$\hat{g} = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - V_\phi(s_t))$$

5: Update the policy parameters using gradient ascent

$$\theta \leftarrow \theta + \alpha \hat{g}$$

6: Fit value function on mean-squared error

$$\phi \leftarrow \underset{\phi}{\operatorname{argmin}} \frac{1}{|D|T} \sum_{\tau \in D} \sum_{t=0}^{T} (V_\phi(s_t) - R_t)^2$$

7: **end for**

The disadvantage of TRPO is that it can be computationally expensive and relatively complicated to implement.

PPO uses a different optimization objective that retains many of the benefits of TRPO, such as stability and data efficiency, while being much simpler to implement. In addition, empirical experiments suggest that PPO provides better overall performance for many tasks. To understand the PPO objective, we must first define the probability ratio and the advantage function.

We use $\rho(\theta)_t$ to denote the probability ratio $\rho(\theta)_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ at timestep t. It measures whether or not action a_t becomes more likely with the updated policy parameters θ than it was with the old policy parameters θ_{old}. If we assess that the action is good, we should increase the probability of selecting it.

The advantage function $A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$ measures whether it is better to choose action a_t in state s_t over what the policy would currently choose [22]. In practice, the advantage function needs to be estimated. For example, the difference $R_t - V_\phi(s_t)$ in VPG (Algorithm 1) is actually an estimate of the advantage function. This estimate has a high variance due to the variability of empirical returns. This variance can be reduced by using an estimation method such as Generalized Advantage Estimation (GAE). We use $\hat{A}_t$ to denote the estimator of the advantage function $A_\pi(s_t, a_t)$ at time step t.

PPO maximizes the following objective

$$J^{PPO}(\theta) = \mathbb{E}_t \left[\min \left(\rho(\theta)_t \hat{A}_t,\ \text{clip}\left(\rho(\theta)_t,\ 1-\epsilon,\ 1+\epsilon\right) \hat{A}_t \right) \right]$$

where $\rho(\theta)_t$ is the probability ratio, $\hat{A}_t$ is the estimate of the advantage, and ϵ is the clipping coefficient. The clipping coefficient ϵ controls how much the policy is allowed to change. The intuition behind the objective is based on how to change the probability ratio to improve the policy while keeping it relatively close to the old policy. More specifically, the objective changes differently depending on whether the advantage is positive or negative. If the advantage is positive, that is, the action was better than expected, increasing the probability of that action causes the objective to increase. Conversely, if the advantage is negative, decreasing the probability of the action causes the objective to increase. To ensure that the policy doesn't change too much, the objective uses minimum and clipping operations to limit how much the objective can be increased by changing the probability ratio.

To use PPO in practice, we can take a standard policy gradient method, such as VPG (Algorithm 1), and modify it to use the PPO objective instead of the regular policy gradient objective. The pseudocode of the PPO algorithm is presented in Algorithm 2.

Algorithm 2 PPO [21]

Require: Policy parameters θ, Value function parameters ϕ

1: **for** iteration $= 0, 1, \ldots$ **do**

2: Collect a set of trajectories $D = \{\tau_0, \ldots \tau_N\}$ by running the policy π_θ in the environment for T timesteps

3: Compute returns R_t

4: Compute advantage estimates $\hat{A}_t$ based on the value function V_ϕ

5: Optimize objective J^{PPO} wrt θ, with K epochs and minibatch size $M \leq NT$

6: $\theta_{old} \leftarrow \theta$

7: Fit value function on mean-squared error

$$\phi \leftarrow \underset{\phi}{\operatorname{argmin}} \frac{1}{|D|T} \sum_{\tau \in D} \sum_{t=0}^{T} (V_\phi(s_t) - R_t)^2$$

8: **end for**

2.3 Multi-agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) is concerned with the learning of multiple agents that interact with the environment and with each other to achieve their goals. The formulation of multi-agent problems requires a different representation than MDP because the dynamics of the environment change based on the actions of each agent. This makes the environment non-stationary from the point of view of a single agent, violating the stationary assumption of the MDP.

There are several ways to formulate multi-agent problems. For fully observable environments, Markov games are a common representation. For cooperation in partially observable environments, decentralized POMDPs are a good choice. In this section we review these frameworks.

2.3.1 Markov Games

Markov game is a generalization from MDPs to multi-agent domains [19]. A Markov game consists of

- set of $N > 1$ agents $\mathcal{N} = \{1, ..., N\}$,
- set of states $\mathcal{S}$ observed by all agents,
- set of actions $\mathcal{A}^i$ for each of agent $i \in \mathcal{N}$, which can be combined into a joint action $\mathcal{A} = \mathcal{A}^1 \times ... \times \mathcal{A}^N$,
- transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ denotes the probability of transition from state $s \in \mathcal{S}$ to next state $s' \in \mathcal{S}$ for any joint action $a \in \mathcal{A}$,
- reward function $\mathcal{R}_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ which determines the reward received by the agent i for a transition from state s with action a to the next state s',
- a discount factor $\gamma \in [0, 1]$ with the same purpose as in the single-agent case.

At each time step t, each agent $i \in N$ observes state s_t and executes action a_t^i. The environment changes to state s_{t+1} based on the actions of all agents, and each agent i receives a reward of $\mathcal{R}_i$. The goal of each agent i is to find a policy π^i that maximizes its expected return. Since the reward is a function of joint action, each agent must learn a policy that takes into account the actions of the other agents.

If the task is cooperative, the agents can share a common reward function. This specialization of Markov games is called Multi-agent MDP (MMDP) in the literature [23]. The consequence of the rewards being the same for all agents is that the value function and the action value function are identical for each agent. This allows the direct use of single-agent RL algorithms by having a central policy that maps from the state to the joint action.

2.3.2 Decentralized Partial-observable Markov Decision Process

A decentralized partially observable Markov decision process (Dec-POMDP) is a specialization of MMDPs for tasks with partial observability [24]. Partial observability means that agents face uncertainty about the true state. In multi-agent problems, it often arises when each of the agents has access only to its local observations and not to the full state.

Dec-POMDP shares almost all characteristics with MMDPs. The set of agents, states, joint actions, transition function, and reward function are equivalent to MMDP. The difference lies in what the agents observe. At each time step t the agent i observes one observation $\mathcal{O}_i$ from the set of observations $\mathcal{O} = \mathcal{O}_0 \times \mathcal{O}_1 \times \cdots \times \mathcal{O}_N$. The observations follow the observation function $O : \mathcal{O} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, which, like the transition function, gives the probability of the observations given the joint action and the next state.

2.4 Methods for Multi-agent Reinforcement Learning

Multi-agent RL algorithms need to account for the fact that there are multiple agents interacting simultaneously with the same environment and possibly with each other [18]. This introduces a new set of challenges that the algorithms must address.

The first challenge is sample efficiency. Because of the trial-and-error nature of RL, it needs a lot of samples to learn effectively. With MARL, this problem is made worse because multiple agents are learning and interacting with the environment at the same time.

Second, there is the non-stationarity of the environment. The stationary assumption of MDPs does not hold for multi-agent environments from an individual agent's point of view, since the next state also depends on the decisions of the other agents. Agents must adapt to the constantly changing behavior of other agents and deal with the fact that the optimal policy may change when the policies of other agents change.

The third challenge is the introduction of partial observability. In many multi-agent

tasks, agents don't have access to the full, global state. In these cases, the agents can only observe their local information. This partial observability means that the agents have to learn from incomplete information, which makes learning more difficult.

Finally, a credit assignment problem arises in collaborative tasks. Credit assignment refers to the problem of figuring out which action led to rewards. In collaborative multi-agent tasks, individual credit assignment may be difficult if each agent receives the same reward. In these cases, it may not be possible to determine whether an individual agent's contribution was beneficial or not if other agents' behaviors also change the reward.

Several approaches have been proposed to address these challenges. One of the simplest is to use a central controller that sees the entire state and outputs a common action. For example, the state can be the concatenation of the state spaces of the individual agents. Basically, this turns the MARL task into a single-agent task and allows the direct use of single-agent algorithms. This approach eliminates the problem of non-stationarity, but has a scaling problem; the state space and joint action spaces grow exponentially in size as more agents are added. The central controller is also a single point of failure, which can be a major problem for real-world multi-agent systems.

On the other hand, there is independent learning. In this approach, each agent learns an independent policy that directly translates local observations into local actions. This is a fully decentralized approach, but it completely ignores the non-stationarity problem. Nevertheless, independent learning has achieved good results in some applications. It also facilitates the use of single-agent algorithms, since each agent can have its own network that it learns. An example of independent learning is Independent PPO (IPPO) [25], which is a variant of PPO for multi-agent settings. In IPPO, each agent learns individual decentralized policy and value functions using only local information. The authors of IPPO implement the agents' policy and value functions by sharing the network weights among the agents. This approach is called parameter sharing, and it has been shown to improve the policy learning of single-agent algorithms adapted for MARL [26].

One of the most common approaches is Centralized Training and Decentralized Execution (CTDE). In CTDE, agents have access to more information about the global state during training. This is usually a fair assumption, since agents are often trained in simulations where the global state is known. After training, execution of the learned policies is fully decentralized. This approach helps with both non-stationarity and partial observability because the agents can take advantage of centralized information during training.

Some CTDE algorithms adopt an actor-critic structure and learn a central critic. An example of this is the Multi-Agent PPO (MAPPO) [27]. It is a modified version of PPO for multi-agent settings. Similar to single-agent PPO, both policy and value functions are learned and represented as neural networks. Each agent learns a decentralized policy using only local observations. The value function, on the other hand, is centralized and takes global information as input. This is possible because the value function is used only during training and can be discarded during execution.

Value decomposition methods, such as VDN [28], are another class of CTDE algorithms that follow a value-based approach where the joint action-value function is represented as a function of the local action-value functions of individual agents. The idea is that the joint action-value function can be learned centrally during training and then factored into individual value functions for each agent for execution. QMIX [29] is a value decomposition method that improves on the earlier VDN method by not requiring full factorization of the joint action-value function. The idea behind QMIX is that it is sufficient that all individual action-value functions produce the same action as the joint action-value function. This is achieved by enforcing a monotonicity constraint between the joint action-value function and the individual action-value functions.

2.5 Sim-to-real and Zero-shot Generalization

To learn good policies, RL requires a large amount of data from the interaction, typically in the range of hundreds of thousands to millions of timesteps. This is because learning is a trial-and-error process, where the agent must explore different behaviors to figure out which ones are rewarded. Therefore, training directly on the real robot hardware is not feasible due to the time it would take to collect enough data [30]. In addition, there are safety concerns with training on real robots because the agents behave erratically during training, i.e., their behavior is random at the beginning. During learning, the erratic behavior of the robots can easily damage the robots themselves or their environment.

The greatest achievements of RL have been in simulated virtual environments such as games. This motivates the idea of simulation-based robot learning. In this sim-to-real paradigm, the policy is first trained in a virtual environment and then transferred to a real robot. Simulation-based training has many advantages, such as faster training and lower cost. In simulation, learning is not limited to real-time, and it is also possible to simulate parallel environments and run them simultaneously to collect large amounts of data.

Although simulators provide a faster, safer, and more cost-effective way to train RL policies, there are often various mismatches or differences between the simulated environment and its real-world counterpart. Policies may overfit the data from the simulated environment. The gap between simulation and reality may be too large for the trained policy to handle. Even small differences in environment parameters such as mass or friction can cause the policy to fail.

The differences between simulation and reality occur in both the sensing and actuation of the robot. For example, the pose information provided by simulation is exact, but the pose information provided by real-world systems is an approximation that always contains some noise or error. Another example is visual information from a camera. It is still difficult to simulate photorealistic environments and all the imperfections of reality. Similarly, the dynamics of the robot are often hard to model accurately, and as a result,

the commands to the robot that were optimal in the simulation may not be optimal in the real world. To achieve a successful sim-to-real transfer, these differences must be taken into account.

To cope with these differences, the agent has to learn to generalize. Generalization in machine learning refers to the ability of the trained model to adapt and respond correctly to previously unseen data. In the context of RL, it refers to the ability of the policy to respond correctly to unseen observations [31]. Commonly, zero-shot generalization is pursued. It refers to the problem setting where the policy is evaluated in an environment different from the one on which it was trained. In this setting, it is not allowed to perform additional training or use additional data from the evaluation environment. In the context of sim-to-real, this means that the policy is trained in the simulation and then deployed in the real world without further training. The goal is for the policy to generalize well to situations for which it hasn't been explicitly trained.

There are several methods that can be used to improve generalization. The most common one is domain randomization. It works by adding random noise to input observations or output actions to make the policy generalize better to variations between simulation and reality. Domain randomization can also be applied to the environment by randomizing physical parameters such as gravity or friction. It is also possible to add random perturbations to the environment such as pushing the robot with random impulses of force [15].

System identification is another method where the goal is to match the dynamics of the real robot as closely as possible in the simulation. For example, a model of the actuation can be learned. The model can be a classical controller or a neural network. It can be learned by first sending predefined actions to the real robot and recording how the robot moves. The model can then be optimized so that performing the same actions in simulation causes the same movement of the robot as in reality [6], [15].

3 Software and Hardware

3.1 Robot Operating System

The Robot Operating System (ROS) is a widely used open-source software framework used for building robotics applications [32]. It has been applied to various areas of robotics, including autonomous navigation, visualization, control, and simulation.

The functionality of ROS is built around processes called nodes, which are used to communicate with other nodes via messages. A node can send a message by publishing it to a particular topic, while another node can receive the message by subscribing to the same topic. Each topic allows only one type of message. For example, the message can be the current pose of the robot or a command to move the robot.

Although this way of communication is very versatile, it is not always suitable for every use case. Therefore, ROS also supports request-response style communication via its services. A ROS service requires a server that listens for requests from a service client. When a request arrives, the server sends an appropriate response. To visualize how communication works in ROS, see Figure 3.1.

In addition to providing a communication framework, ROS provides several software packages to help build robotics applications. These include a visualization tool RViz and a transformation system TF.

RViz allows users to visualize data published in specific topics. It provides panels that can be dynamically opened to view different types of data, such as images and point

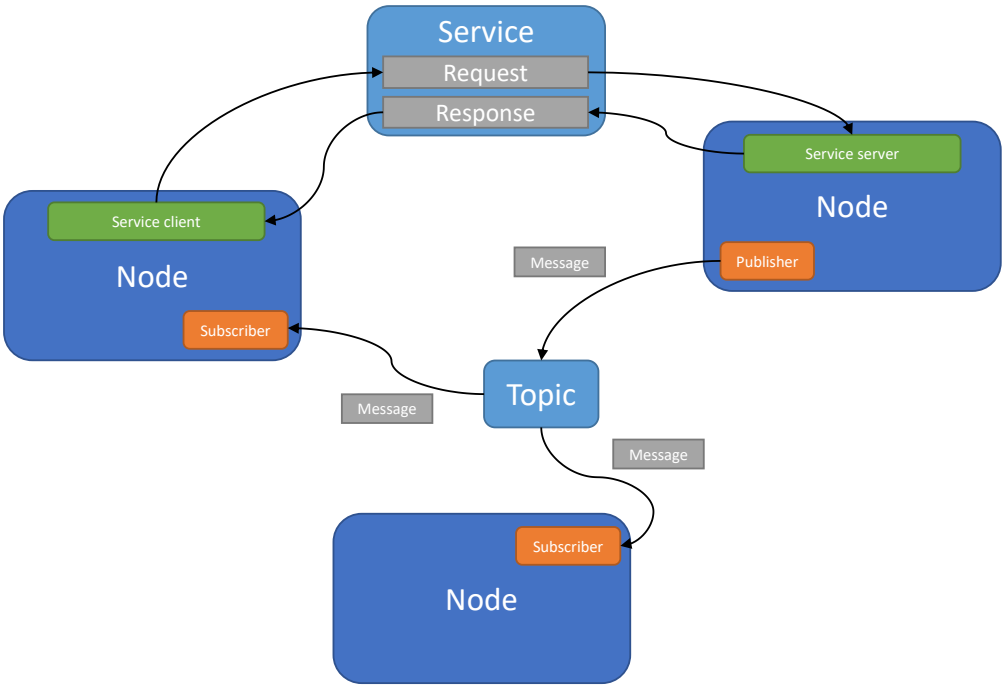

Figure 3.1: Communication between nodes in ROS

clouds, or to render the pose of a robot.

TF is a transformation system that can track and relate different frames of reference. For example, it can determine the position of the manipulator's gripper frame relative to the manipulator base. TF constructs a dynamic transformation tree that contains all the spatial relationships between the system's reference frames. This tree can be used to obtain the transformation from one frame to another.

Finally, ROS promotes modularity and reusability. Nodes, configurations, and other essential files can be organized into packages that promote code reuse and collaboration among multiple individuals.

Figure 3.2: Graphical user interface of Isaac Sim

3.2 Isaac Sim

NVIDIA Isaac Sim is a simulation platform for creating virtual environments that can be used to train AI-based robots and generate synthetic data [33]. It provides advanced GPU-based physics simulation, physically accurate rendering, and photorealistic graphics. A screenshot of Isaac Sim's graphical user interface is shown in Figure 3.2.

Isaac Sim supports RL via the Omniverse Isaac Gym extension. It provides an interface for training and executing RL policies within Isaac Sim. With this interface, it is possible to use Isaac Sim to train thousands of RL agents in parallel. Each parallel agent collects training data simultaneously, and together all the data can be used to train a single shared policy. This makes training much faster because the interaction data can be collected at scale. It also has the benefit of making the policy more robust, as each agent can explore slightly different behaviors, making the estimates of the policy and value function more accurate.

To use RL in Isaac Sim for a robot learning task, the task has to be modeled and implemented in Omniverse Isaac Gym. The task serves as an interface between the RL algorithm provided by an RL library and Isaac Sim. It specifies what kind of robots and what kind of world should be simulated. After the task is implemented, it is possible to read the current state of this environment, calculate rewards from it, and pass all this information to the RL algorithm. In turn, the RL algorithm will output actions that will be used to control the robots in the simulated environment.

As a starting point, many examples of robot task implementations are provided in a repository called OmniIsaacGymEnvs [34]. These tasks include classic tasks such as Cartpole balancing, as well as more complex tasks such as opening a cabinet drawer with the Franka Emika Panda manipulator.

To train an RL policy for these tasks, an RL library is needed. These libraries require a certain Application programming interface (API) to communicate with Isaac Sim. An environment wrapper is a way to encapsulate the task interface of Omniverse Isaac Gym to provide a specific API needed by an RL library. OmniIsaacGymEnvs includes an environment wrapper and training scripts for the RLGames library, making it easy to train policies.

3.3 Clearpath Husky

Clearpath Husky is a differential drive robot designed for rugged terrain. It weighs 50 kg and can carry payloads up to 75 kg. It is used for many applications including mapping, localization and autonomous navigation. The robot communicates through an on-board computer that interfaces with low-level controllers and sensors. The on-board computer runs ROS, which can be used, for example, to command the robot to move at a desired speed or to retrieve wheel odometry data. Controlling Husky is easy, simply publish the desired linear and angular velocity to its command velocity (cmd_vel) topic.

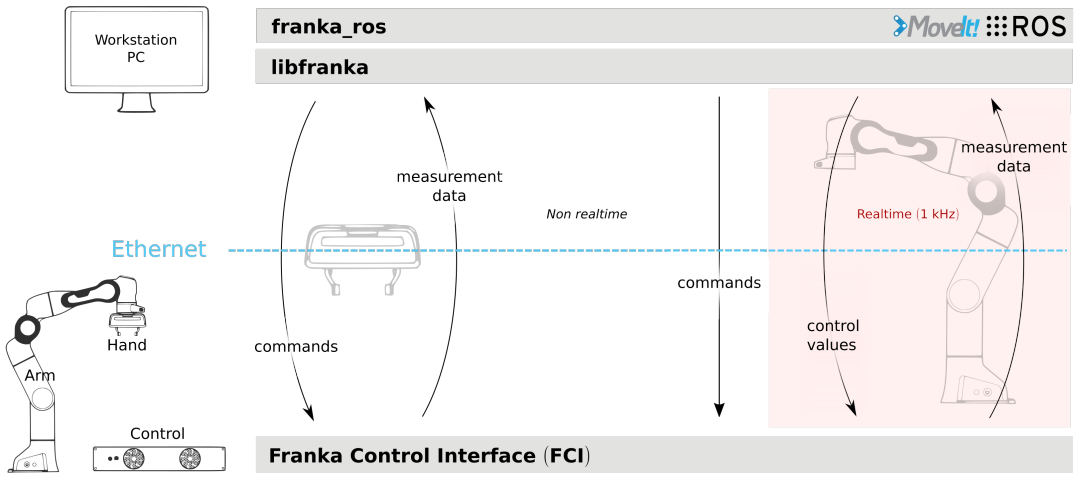

Figure 3.3: Overview of the Franka Control Interface [35]

3.4 Franka Emika Panda

Franka Emika Panda is a collaborative robot arm with 7 degrees of freedom. It is designed to assist humans in various tasks and is also popular in the robotics community for research purposes. Panda has 7 controllable revolute joints, each equipped with a torque sensor. It weighs around 18 kg and can lift payloads up to 3 kg.

The arm is controlled by the Franka Control Interface (FCI) [35]. It allows an external workstation to connect to the Panda system via Ethernet. With its fast bi-directional communication, it provides access to the current status of the robot and allows its direct control. Communication via FCI is illustrated in Figure 3.3.

A C++ interface libfranka can be used on the workstation to read data and send commands. It is possible to get real-time measurements of joint data such as position, velocity and torque sensor signals. Many different modes can be used to control the robot. The user can choose between sending the desired torque to the motors, the desired joint position or velocity, or the desired Cartesian position or velocity of the end effector, i.e. the hand of the arm. The FCI controller takes these command values and manipulates them so that the motors can generate the appropriate torque to achieve the desired configuration.

Another package, franka_ros, provides integration between libfranka and ROS. This

allows the use of ROS nodes and topics to control the robot and read its status. It includes detailed 3D models of the robot for visualization in RViz and simulators. It also integrates with the MoveIt motion planning framework, providing an easy way to control the robot and its end effector.

4 Methodology

In this chapter, we present our methods to turn a single-agent robot task into a multi-agent robot task by dividing the robot's control among multiple cooperating agents. We hypothesize that this has advantages such as faster learning or that the resulting policies are more robust to perturbations such as malfunctioning of some parts of the robot. We consider two different methods that differ in how the single-agent task is formulated as a MARL problem and how much information is available to the agents.

4.1 Multi-agent Reinforcement Learning with Global Observations

Our first method is motivated by the fact that to turn a single-agent task into a multi-agent task in any meaningful way, the action space needs to be divided among the agents.

To start, we assume a robotic task that is formulated as an MDP. In this task, we consider the robot as the agent. The agent can observe states from the state space $\mathcal{S}$ and perform actions from the action space $\mathcal{A}$. There is also a transition function that specifies how the environment changes based on the agent's actions, and a reward function that specifies the rewards from the agent's behavior. We further assume that the task has a structure that allows us to divide the task meaningfully among $N > 1$ agents.

The agents can correspond to different sections or parts of the robot. For example, in the case of a manipulator arm like Panda, each agent may be responsible for controlling a

single joint of the arm. Another example is a legged robot, where each agent can control a separate leg. These agents must work together to control the entire robot to accomplish the given task.

To turn the single-agent task into a multi-agent task we need to assign each agent a subset of the dimensions of the action space. Each action dimension should be associated with only one agent. Although it might be interesting to allow the same action dimension for multiple agents, it would require additional consideration of how to interpret two agents giving different values for the same action. We don't consider this here, as it is not the focus of this work.

After assigning the action dimensions to the agents, we formulate the task as a multi-agent MDP (MMDP). We denote the set of all agents by $\mathcal{N} = \{1, ..., N\}$. The state space remains unchanged and all agents observe the same states. We partition the original action space into non-overlapping subsets $A_1, A_2, ..., A_N$, one for each agent. Note that the original action space can be constructed as the Cartesian product of the action space subsets: $\mathcal{A} = A_1 \times A_2 \times ... \times A_N$. In MARL nomenclature, this is called the joint action space. The transition function remains the same as in the original MDP because the state space and the joint action space are the same. Similarly, the reward function stays the same because the agents are working toward a common goal and we can treat the reward as a team reward. This means that the reward will be the same for all agents.

Solving this task amounts to learning a policy for each agent that maximizes the expected discounted team return. The MMDP formulation is actually equivalent to the original MDP formulation if we learn a joint policy that maps from the state space to the joint action space. Thus, to treat the task as a true multi-agent problem, we learn individual policies π^i for each agent. The policies can be learned with any MARL algorithm, e.g., Multi-Agent PPO.

4.2 Multi-agent Reinforcement Learning with Partial Observations

The second method is motivated by the idea that the agents may not need to observe the same states. As an example, let's consider a mobile manipulation task where one agent controls the mobile base and another agent controls the manipulator arm. The agent controlling the arm may not need to know all the information about the mobile base and vice versa. Limiting the information that the agents observe introduces partial observability into the system. Although this partial observability can make the task more difficult, it can be helpful in certain situations. For example, if one of the agents malfunctions, the state information associated with that agent will be corrupted. This corrupted information can negatively affect the other agent if it uses the full state information, since it has not encountered such situations during training. Conversely, a policy using partial observations may be more robust to this type of perturbation because it is decoupled from the extraneous information.

The previous method is adapted by introducing partial observability for the agents. This is done by dividing the full state among the agents and formulating the problem as a Dec-POMDP. The set of agents, the state space, the action spaces for each agent (and thus the joint action space), and the team reward function remain the same as in the MMDP of the previous method in Section 4.1. The dimensions of the state space are divided into observations for each agent. Unlike action spaces, observation spaces can overlap. It may be beneficial for agents to observe common information. This can be thought of as them sharing information with each other.

The goal of this method is to learn a policy π^i for each agent that maps from the agent's local observation space to its action space. Since we are dealing with local observations, it is possible to use the centralized training and decentralized execution approach introduced in section 2.4. The value function that uses the full state is used only during training, and

the execution of the policy can still be decentralized. This helps the value function to better estimate the expected return, which can make learning more robust and faster.

5 Implementation

This chapter presents the implementation of the methods. First, we design a mobile manipulation reach task and frame it as an RL problem. Then, we model this task in Isaac Sim and implement the proposed methods that are used to train MARL policies. Finally, we present the real-world hardware and software setup in which we deploy the trained policies and which is subsequently used for experiments. Parts of the text and figures of this chapter are reproduced from our previous work [36].

5.1 Mobile Manipulation Reach Task

We chose a relatively simple learning task to evaluate the methods. In the task, two agents work together to control a mobile manipulator to reach a given target point in space with the manipulator's end-effector. One of the agents controls the mobile base and the other controls the manipulator. We refer to these agents as the base agent and the manipulator agent, respectively. Specifically, the agents are tasked with minimizing the distance between the end-effector and the target point. The target point should be far enough from the start position to force both robots to move in order to complete the task. The task is shown in Figure 5.1.

Next, we formulate this task as an RL problem for each of the methods. In addition to the MARL methods with full and partial observations, we will also use a single-agent PPO method as a baseline in our experiments.

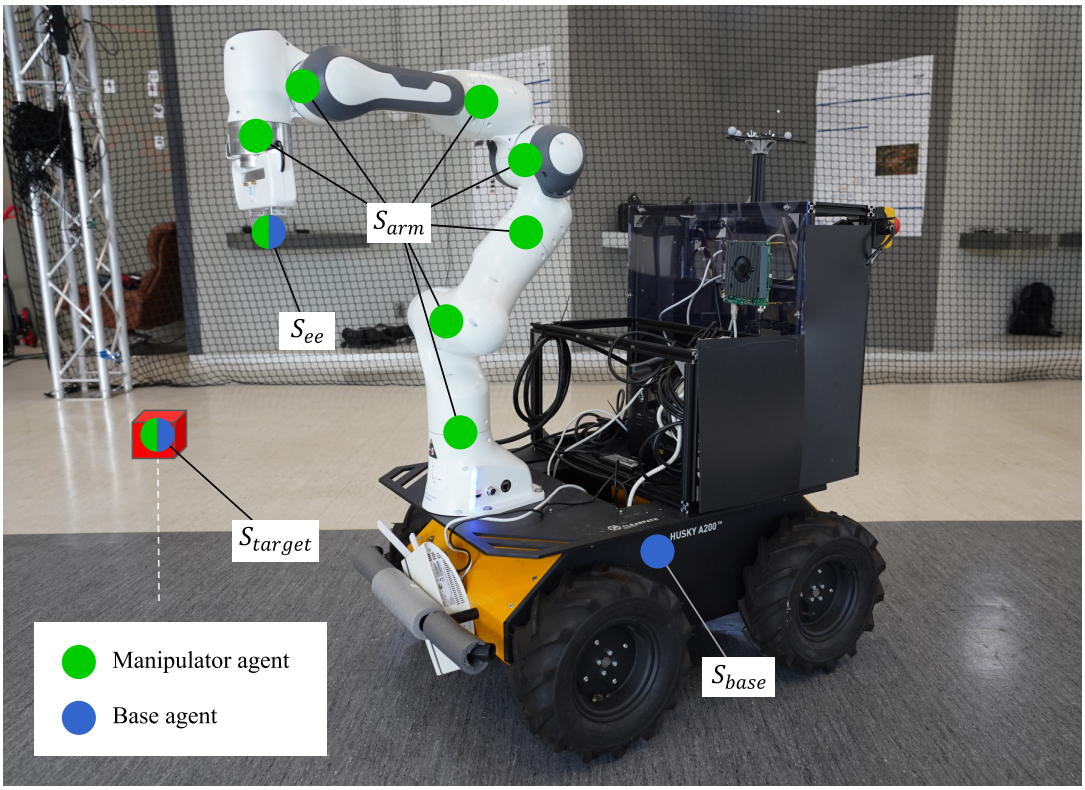

Figure 5.1: In the mobile manipulation reach task the base agent and manipulator agent need to collaboratively control the mobile manipulator to reach the target point with the end-effector. The observations of the agents are depicted in the figure. The actions are the desired joint angles for the arm and the linear and angular velocity for the base.

Observation Space The state or observation space depends on the method. Full state information $(s_{\text{base}}, s_{\text{arm}}, s_{\text{ee}}, s_{\text{target}})$ is available for the fully-observable MARL and the single-agent baseline. For partially-observable MARL, the base agent observes $(s_{\text{base}}, s_{\text{ee}}, s_{\text{target}})$ and the manipulator agent observes $(s_{\text{arm}}, s_{\text{ee}}, s_{\text{target}})$. The state of the mobile base $s_{\text{base}} \in \mathbb{R}^3$ is its translation in x and y coordinates relative to the starting position of the base and its yaw in the global frame. The state of the manipulator $s_{\text{arm}} \in \mathbb{R}^{18}$ is composed of its current joint angles and the angular velocities of each of its joints. The position of the manipulator's end-effector $s_{\text{ee}} \in \mathbb{R}^3$, and the target position $s_{\text{target}} \in \mathbb{R}^3$. The positions s_{ee} and s_{target} are relative to the start position of the base.

Action Space The actions for the base agent are the linear and angular velocity commands for the mobile base $(a_{\text{base\_x}}, a_{\text{base\_yaw}})$. For the manipulator agent, the actions are the joint commands for the manipulator $a_{\text{arm}} \in \mathbb{R}^9$ which control the desired change for the joint angles. All the actions are continuous and are clipped to a range of $[-1, 1]$.

Reward Function We design the reward function with a number of constraints in mind. Most importantly, we want to reward the agents more the closer they can position the gripper to the target point. To do this, we use the following exponential function

$$r_d = \exp(-\alpha \| s_{\text{target}} - s_{\text{ee}} \|)$$

where $\alpha \geq 0$ is used to scale the amount of the reward. This reward increases as the distance between the end-effector and the target point decreases. Additionally, we want to keep the arm movements small and controlled to avoid jerky movements. To this end, we denote r_d as the squared sum of each arm action a_{arm} and use it to penalize large arm movements. Finally, we don't want the manipulator arm to move to the back of the mobile manipulator, because the real robot has computers, batteries, and other hardware there. And we don't want the arm to twist into limiting positions or hit the ground or the mobile base. To ensure these constraints, we design a function r_l that penalizes joint configurations that differ greatly from the neutral configuration. To tune the importance of each reward component, we scale them with weights $w_p, w_d, w_l \geq 0$. The full reward function is defined as

$$r = w_d r_d - w_p r_p - w_l r_l.$$

In summary, to maximize the expected return, the agents should learn policies that will collaboratively position the manipulator's end effector as close as possible to the target point while simultaneously keeping the manipulator arm close to a neutral position to avoid hitting the computers on the back sections of the Husky or the ground.

5.2 Simulating the Task

Nvidia's Isaac Sim is used to model the mobile manipulator reaching task and to perform the RL training. The mobile manipulation reaching task is implemented for Omniverse Isaac Gym using the OmniverseIsaacGymEnvs repository as a starting point. The repository also includes an interface to RL algorithms provided by the RLGames library. Neither the Omniverse Isaac Gym nor the examples repository explicitly support multi-agent RL, so some modifications are required.

The implementation starts with setting up the robots and the world. Isaac Sim already includes the model for the Franka Emika Panda robot, but not the model for the Clearpath Husky. Due to the problems of importing the Husky model into Isaac Sim and combining it with the Panda, we use a mobile manipulator robot that is already included in Isaac Sim. This robot is a combination of the Clearpath Ridgeback and the Franka Emika Panda. The robot has the same Panda manipulator as our robot, but a different mobile base. Fortunately, the observations and actions for the base are quite simple and should not affect the sim-to-real transfer too much. While the model looks realistic, it is essentially a floating base that isn't affected by gravity. Nevertheless, we consider this robot sufficient for our task, since we can assume that the task won't be significantly affected by gravity.

The final environment includes the mobile manipulator robot and the specified target point. The target point is not a physical object in the simulator but we add a visual cube to visualize the target point in the simulator's user interface. To scale up the training and provide more varied training data, we use parallel environments. The functionalities provided by Omniverse Isaac Gym allow the cloning of the environment. The number of parallel environments is a parameter that can be changed, in this case we choose to use 512 parallel environments. During training, these cloned environments have no interaction between them, but they all provide the training data to train the agents. To reduce the memory usage of these environments, we use the instanceable assets feature of Isaac Sim. It allows each copy of the robots to reference a single copy of the 3D model's mesh instead

of creating multiple copies of the same mesh. This makes training much faster and more efficient.

The observations and actions of the robots are modeled as described in the previous section. The simulator provides all the information necessary to create the observations and rewards. The poses of the robots and the positions and velocities of the joints are queried by the simulator and used to build the observations and rewards. These are then passed to the RL algorithm provided by RLGames. The RL algorithm then sends back the actions of the policies and these are used to set the desired joint positions and velocities for Isaac Sim's built-in controllers to move the mobile manipulator.

The simulation runs at a frequency of 120 Hz, while the RL interaction runs at 60 Hz. The maximum length of an episode is set to 500 steps of RL interaction. Each step of RL interaction in the simulator consists of

1. Read the state of the robots and the environment and build an observation tensor.
2. Send the observation tensor to the RL algorithm.
3. The RL algorithm processes the observation tensor and produces an action tensor.
4. The action tensor is divided into actions for the base and the arm.
5. We set the target velocity for the base robot and the target positions for the joints of the arm using the base and arm actions, respectively.
6. The simulation physics are stepped, causing the robots to move according to the actions.
7. The time limit is checked and the environments are reset accordingly.

After the time limit is reached, the environment is reset. This returns the robot to its initial position and randomizes its yaw. The Panda arm is also reset to its initial configuration. Finally, the coordinates of the target point are randomized.

The simulated mobile manipulation reach environment is shown in Figure 5.2. The figure shows a screenshot which is taken in the middle of the training and illustrates the mobile manipulator robot and the parallel environments that were used in the simulator.

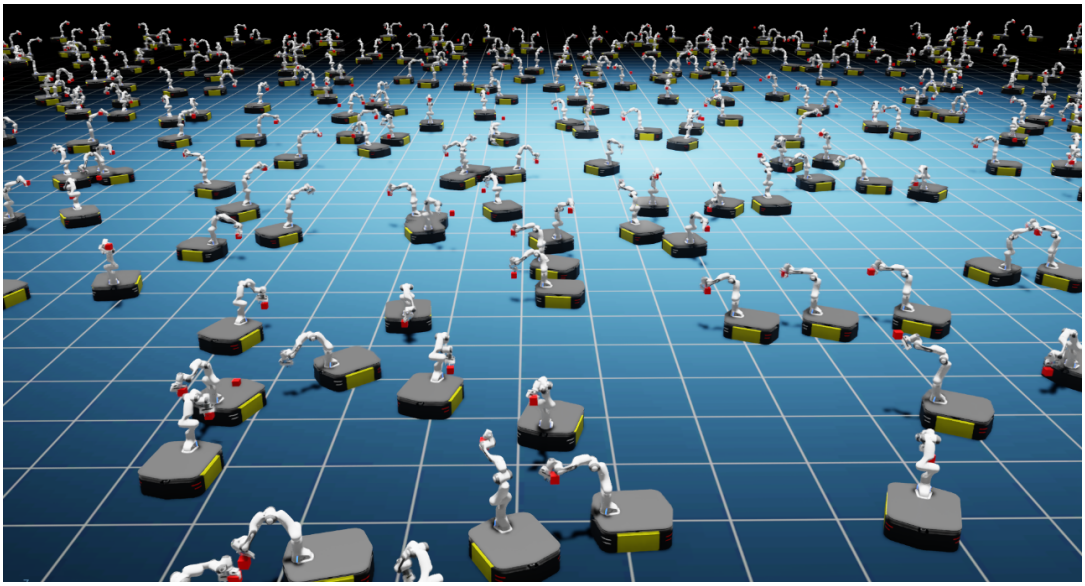

Figure 5.2: MARL training for the mobile manipulation reach task in Isaac Sim using parallel environments. The agents, one controlling the base and the other controlling the arm, are tasked to collaboratively operate the mobile manipulator to position the gripper as close as possible to the specified target point, visualized here as a red cube.

5.3 Multi-agent Proximal Policy Optimization

To train the agents, we follow an actor-critic approach where both the policy and value function parameters are learned. The value function is used for variance reduction during training, but is discarded during deployment. We parameterize the policies with parameters θ, which correspond to the weights and biases of a Multi-Layer Perceptron (MLP). The policy is modeled as a Gaussian distribution with diagonal covariance. The output of the MLP is the mean of each action, i.e. the velocities for the base and the desired angles for the manipulator. The diagonal covariance is given by a separate set of parameters that

are learned to represent the standard deviations of the actions.

Instead of using different parameters θ for each policy, we share the parameters between policies. This allows us to use one MLP for multiple policies. Parameter sharing has been shown to speed up the learning process for MARL and possibly help with non-stationarity [26]. In previous work, parameter sharing is typically used only in cases where the agents are homogeneous, i.e., the agents are similar entities with the same observation and action spaces. For heterogeneous agents, such as the base and manipulator in our task, parameter sharing is not as straightforward. The MLP does not have information about which agent the observation belongs to, and thus cannot output meaningful actions. To handle this, we follow an approach suggested by Terry et al. [37], where a one-hot encoded agent indicator is added to the input. This indicator is `01` for the base agent and `10` for the arm agent. The agent indicator conditions the MLP to output different actions for each agent's observations, allowing it to represent two different policies with the same parameters. An important detail is that the number of dimensions in the observation and action spaces of the agents can be different depending on the method. The extra dimensions in the observations are solved by filling in zeros instead of the extra dimensions. For the dimensions of the action spaces, we simply take the number of dimensions that belong to the agent and discard the rest.

The representation of the value function depends on the method. For the single-agent baseline and the MARL with global observations, we take advantage of the learned features of the MLP. That is, we use the same network as the policies and have separate output weights for the value function. The MLP is able to output different values for different agents based on the agent indicator. For the MARL with partial observations, we take the opportunity to use CTDE by training a centralized value function that takes the full state as input. This allows agents to benefit from centralized information during training, but be decentralized during deployment. The central value function is represented by its own MLP and does not share weights with the policy MLP. The complete architectures

of the MLPs for different methods are shown in Figure 5.3.

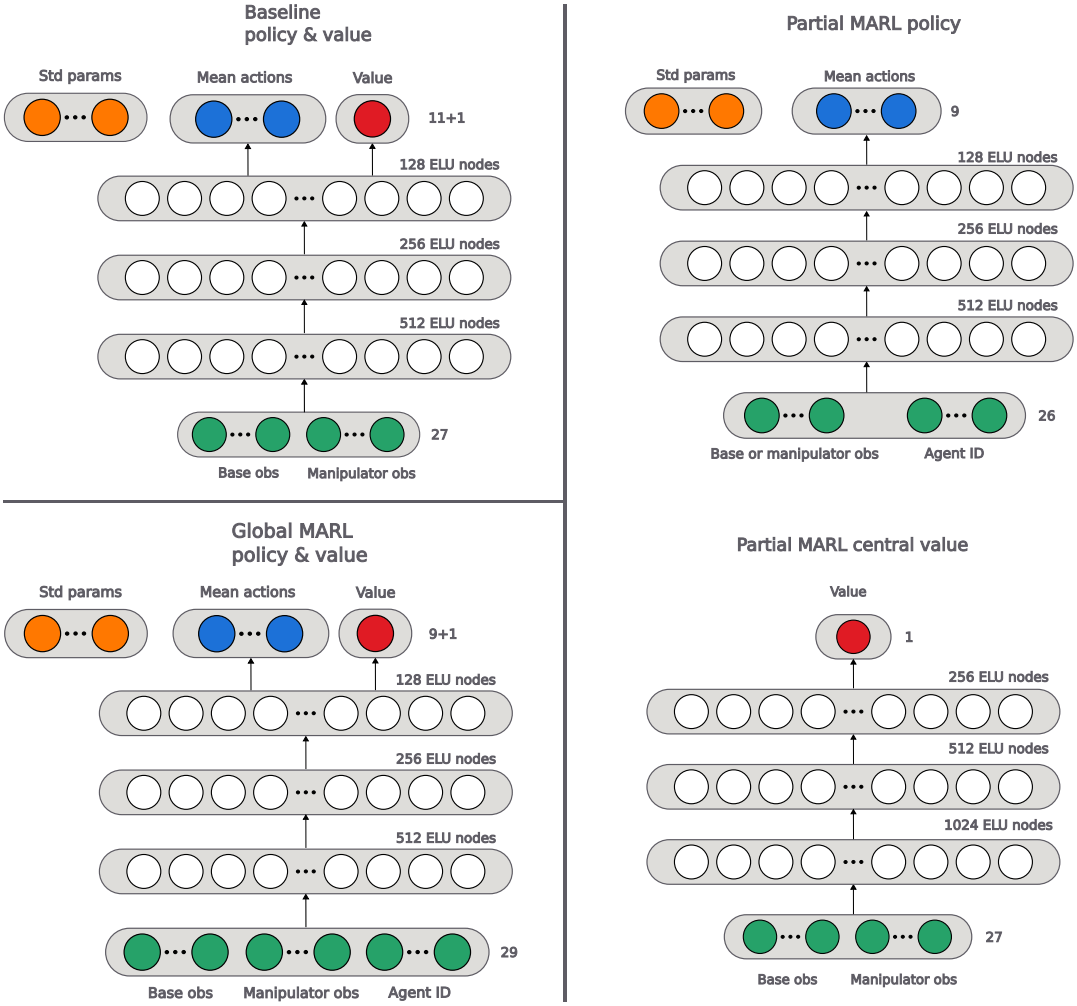

Figure 5.3: Neural network architectures for each method

To optimize the policies of the agents, we use a multi-agent variant of PPO (MAPPO). We implement the algorithm by adapting the single-agent PPO (Algorithm 2) by changing the data collection step. The data is collected from the parallel simulation environments using the multi-agent policies and consists of observations o_t^i, actions a_t^i, rewards r_t, boolean values for terminal states done_t, action probabilities $\pi_\theta^i(a_t^i|o_t^i)$, and estimated values $V^i(o_t^i)$ for each agent $i \in \{base, arm\}$. Agent indicators are added to the observations before they are passed to the MLP. The output of the MLP is used to control the agent corresponding to the indicator. The pseudocode for the algorithm is provided

Algorithm 3.

Algorithm 3 Multi-agent PPO with Parameter Sharing

for each training epoch **do**
  for each rollout step t **do**
    Query arm and base observations, o_t^{base}, o_t^{arm} from the environment
    Add Agent IDs to observations $o_t^{base} \leftarrow \{o_t^{base}, 0, 1\},\ o_t^{arm} \leftarrow \{o_t^{arm}, 1, 0\}$
    Compute policy for the agents $a_t^i \sim \pi_\theta^i(o_t^i),\ i \in \{\textit{base}, \textit{arm}\}$
    Perform the actions in the environment
    Store in a rollout memory the samples
    $(o_t^i, a_t^i, r_t, \text{done}_t, \pi_\theta^i(a_t^i|o_t^i), V^i(o_t^i))\ ,\ i \in \{\textit{base}, \textit{arm}\}$
  end for
  Estimate returns R_t^i and advantages A_t^i using $(r_t, \text{done}_t, V^i(o_t^i))$
  Optimize PPO objective wrt θ for K mini-epochs and minibatch size M
  using $(\pi_\theta^i(a_t^i|o_t^i), A_t^i)$
  Fit the value functions V^i by minimizing $(R_t^i - V^i(o_t^i))^2$
end for

During the training, we use various settings and hyperparameters to tune the performance. We normalize the input observations, values, and advantages by subtracting the mean and dividing by the standard deviation using statistics collected during training. Additionally the observations are clipped to $[-7, 7]$ and actions are clipped to $[-1, 1]$. The hyperparameters for the training are provided in Table 5.1.

5.4 Real-world Experimental Setup

To evaluate the policies produced by the MARL methods, we use a ROS based system to allow communication between the neural network policy and robot. The overview of the system is provided in Figure 5.5. The mobile manipulator robot is built with a

Hyperparameter	Value
Training epochs	1500
Rollout steps	16
Minibatch size	4096
Mini-epochs	8
Discount factor	0.95 (0.9 for Partial MARL)
GAE-Lambda	0.95
Learning rate	0.003
PPO ratio clip	0.2
Gradient norm clip	1.0

Table 5.1: Training hyperparameters

Clearpath Husky as the mobile base and a Franka Emika Panda as the manipulator arm. The Panda arm is attached to the front of the Husky, and its control computer (FCI) is mounted vertically on the back of the Husky. The mobile manipulator system is shown in Figure 5.1.

Both Husky and Panda have dedicated computers that run their respective ROS nodes and other programs. The computer for Husky runs all the Husky-specific software, such as the interface for sending motion commands to the motors. On the computer for Panda, we run the Franka Control node from the frankaros package. This node provides an interface between ROS and the Franka control interface. It reads and publishes the state of the arm joints and listens for commands to control them. Additionally, it publishes TF2 transforms for all joints, which can be used to get the transformation from the base of the arm to the end effector. Communication between the computers is handled by a router that was on board the system.

To capture the robot's pose, we use the OptiTrack motion capture system. Several motion capture cameras are pointed at the test area and detect markers placed on the robot.

The system is then able to calculate the pose of the markers with respect to the frame of the OptiTrack system. A separate computer handles the OptiTrack system and provides the pose information through a VRPN server. We use a package called vrpn_client_ros to connect to the VRPN server and make this pose available as a ROS topic.

Another node uses TF2 to transform between different coordinate frames. First, we create a dynamic transformation that continuously reads the OptiTrack pose topic and makes it available to TF2. Then we use manually measured transforms from the OptiTrack frame to the Husky and Panda base frames. The transforms are visualized in Figure 5.4. In the figure, the yellow arrows visualize the transforms from the child frame to the parent frame. The axes visualize the orientation of the frame with respect to the global OptiTrack frame.

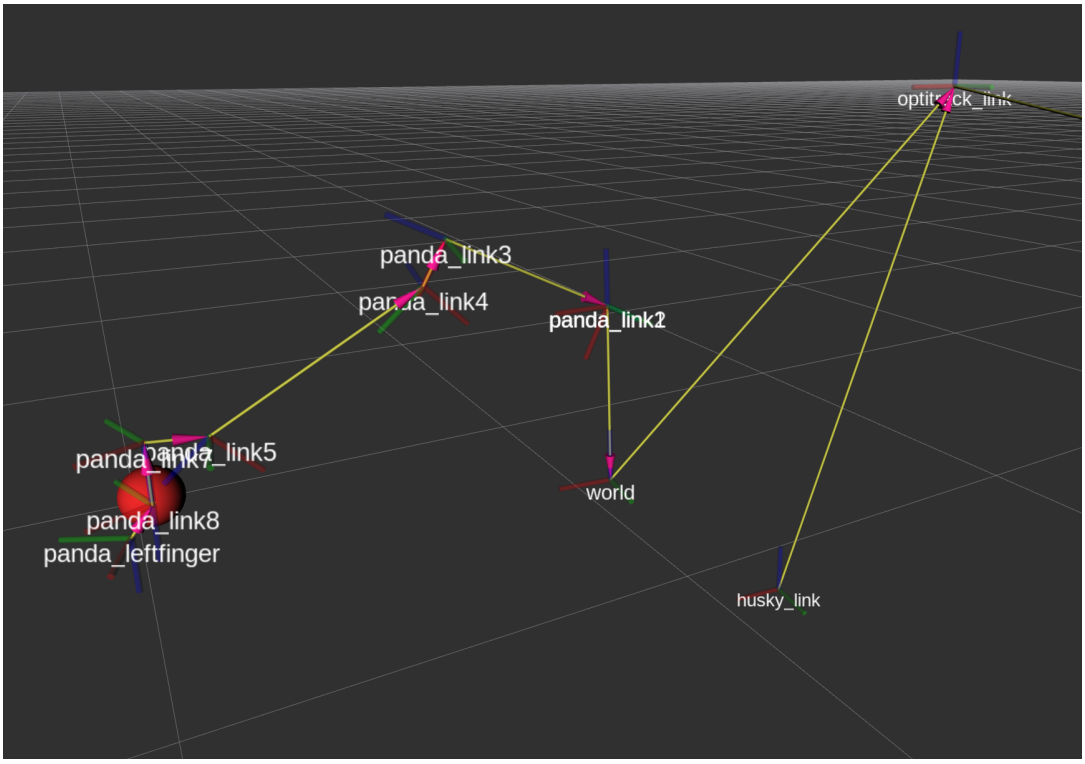

Figure 5.4: Transforms from Optitrack frame to Husky's base frame and to Panda's world frame, and from world frame all the way to Panda's left finger. The red sphere shows the current target point.

The central component of the setup is the ROS node, which uses the trained neural network to control the robots. The neural network is exported from Isaac Gym in Open Neural Network Exchange (ONNX) format. With the ONNX runtime, we can use the neural network model inside the ROS node. We use a TF transform listener to get the poses of the base and end-effector. The observations for the arm, i.e. joint positions and velocities, are read from the topic published by the franka_control node. All poses and arm observations are scaled in the same way as in the training phase and then combined appropriately into inputs for the neural network. The agent indicators are added to the inputs in the same way as in training. The agent observations are then passed to the neural network to obtain the actions. As a safety feature, the actions are scaled down to ensure that it is possible to stop the robot at any time. The control loop runs at the same 60 Hz frequency as in the simulator.

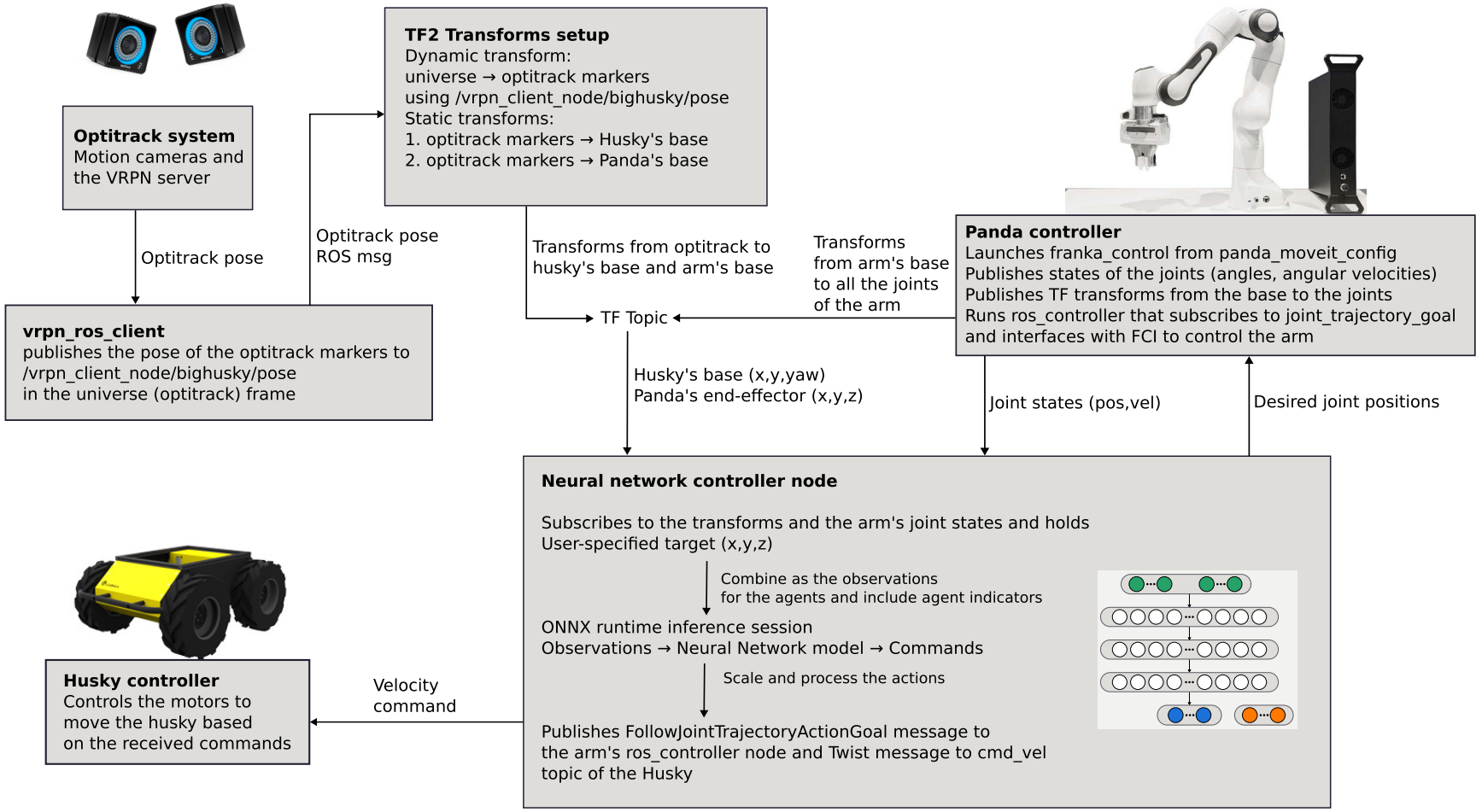

Figure 5.5: Overview of the real-world setup

6 Experiments and Results

Parts of the text and figures of this chapter are reproduced from our previous work [36].

6.1 Training Performance

The training curves for MARL with global observations (Global MARL), MARL with partial observations (Partial MARL), and the single-agent PPO baseline (SARL) are shown in Figure 6.1. We train each method for three different random seeds and plot the mean and standard deviation of the episodic return over 1500 training epochs. The results suggest that the multi-agent methods can lead to better performance. The MARL method with partial observations learns the fastest, suggesting that it can be more sample efficient. Finally, for each method, a training run of 1500 epochs takes only about 15 minutes of wall-time to complete.

6.2 Real-world Experiments

The methods are evaluated in several real-world experiments using the setup introduced in the previous chapter. We compare the multi-agent methods to a single-agent PPO baseline trained on the same task. In addition, we compare global observability to partial observability for the multi-agent methods. The experiments are designed to test the methods under different conditions, which are: (1) nominal conditions, (2) manipulator agent disabling, and (3) manipulator agent disabling and arm repositioning. These were chosen

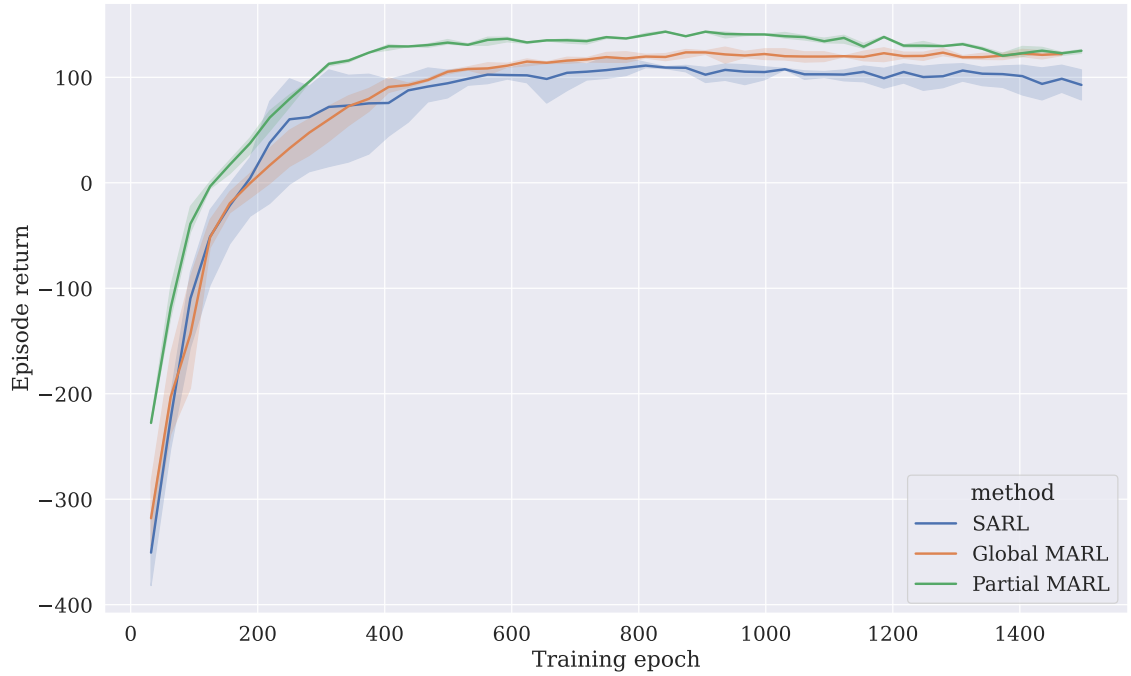

Figure 6.1: Mean episodic returns reached during training for MARL with global observations (Global MARL), MARL with partial observation (Partial MARL), and the single-agent PPO baseline (SARL).

to test the viability of the MARL approaches and the robustness of the resulting policies.

It is important to note that the policies may perform differently depending on where the target point is relative to the robot's starting position and yaw. To get a more realistic evaluation, we run each experiment over a set of target points and vary the initial orientation of the robot. These are chosen to force the robot to move forward, backward, and rotate. The target points can be seen in 6.1.

id	x	y	z
1	0.4	-0.6	0.5
2	0.3	2.0	0.7
3	-2.0	-1.0	0.4

Table 6.1: Set of target points

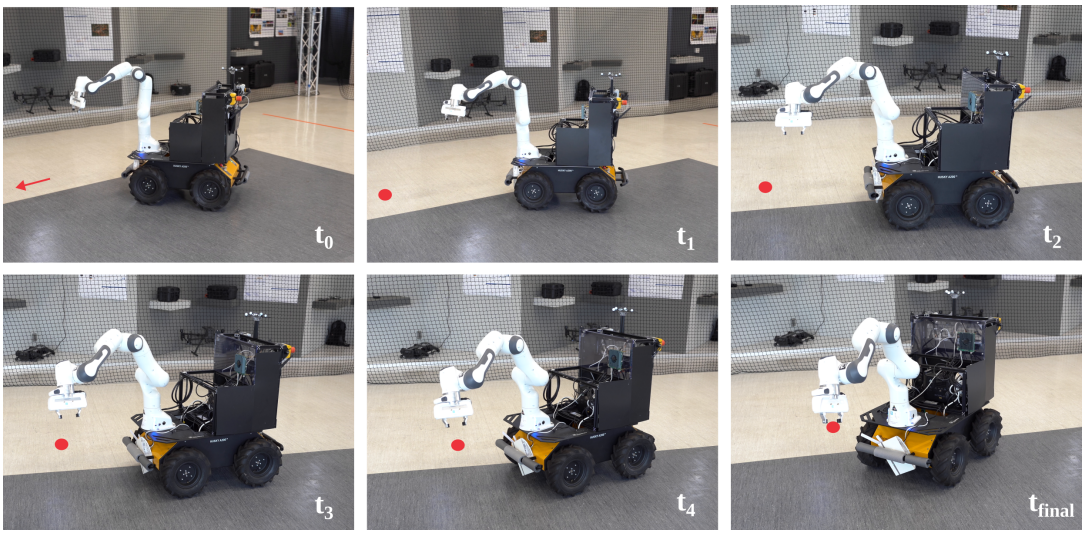

Figure 6.2: Time-lapse of a single experiment for Partial MARL under nominal conditions. The neural network policy trained simulation can perform the task in the real world.

6.2.1 Nominal Conditions

In this experiment, we evaluate the different methods under nominal conditions, i.e. the same conditions for which they were trained. The results are shown in Figure 6.3. The baseline single-agent PPO policy performs best for all three points, resulting in the closest distance to the target point. Both MARL approaches converge to a similar distance as the baseline. Even the introduction of partial observability has minimal impact on performance. The results show that it is feasible to use our multi-agent approach to train policies for real robots. Importantly, we are able to transfer the policies from the simulation to a real system in a zero-shot fashion. Time-lapse of an experimental run is shown in Figure 6.2.

6.2.2 Disablement of the Manipulator Agent

Now we test how well the policies perform when one of the agents is disabled and not contributing. We disable the manipulator agent, which in practice means that its actions

are not used to control the arm. The manipulator arm remains in its initial state, and only the base can move. The base agent has to move the mobile manipulator as close as possible to the target point without any help from the manipulator agent. The results of this experiment are shown in Figure 6.4.

The MARL method with global observations diverges in all cases, which may be due to the base agent's policy being unable to decouple the arm's observations from its actions. Surprisingly, the PPO baseline using the same global observations is able to perform well. It is important to note that further experiments in the simulation show that the single-agent baseline is not consistently successful. MARL with partial observations converges to the closest distance for each point and demonstrates the most robust behavior.

6.2.3 Disablement of the Manipulator Agent and Additional Perturbation

In this experiment, in addition to disabling the manipulator agent, we add another perturbation to the previous experiment by manually repositioning the arm to a different initial joint configuration. This emulates the case where the arm stops working in the middle of the task and the base has to compensate to complete the task as well as possible. The results are shown in Figure 6.5.

The methods that rely on global observations, namely baseline and global MARL, are no longer able to complete the task. Partial MARL with its local observations is able to cope with the additional perturbation and succeed in the task. The results suggest that the introduction of partial observability for the agents allows for increased robustness to perturbations in the system, such as the malfunctioning of some agents.

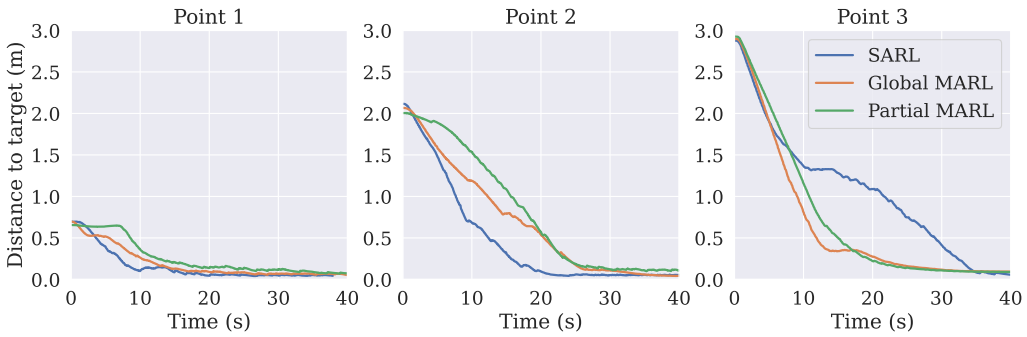

Figure 6.3: Results with a mobile manipulator under nominal conditions. We show the viability of the MARL approach and the zero-shot sim-to-real transfer. Single-agent baseline performs the best but MARL approaches achieve near optimal performance.

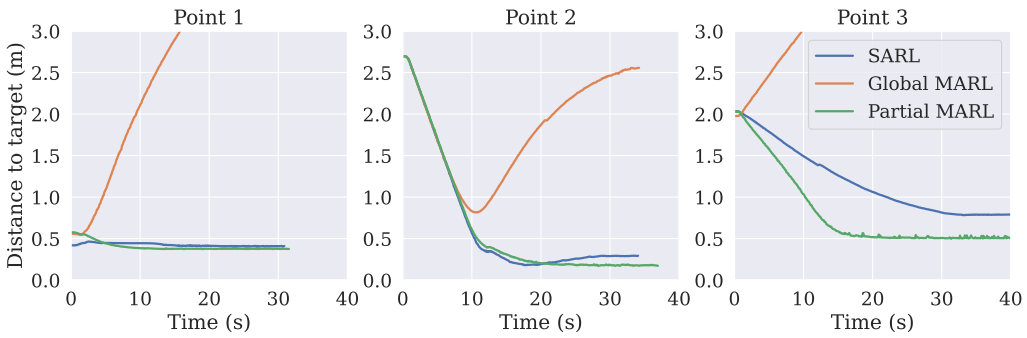

Figure 6.4: Results with a mobile manipulator with the manipulator agent disabled. MARL approach with global observation fails. Single-agent baseline and Partial MARL succeed with the latter performing better.

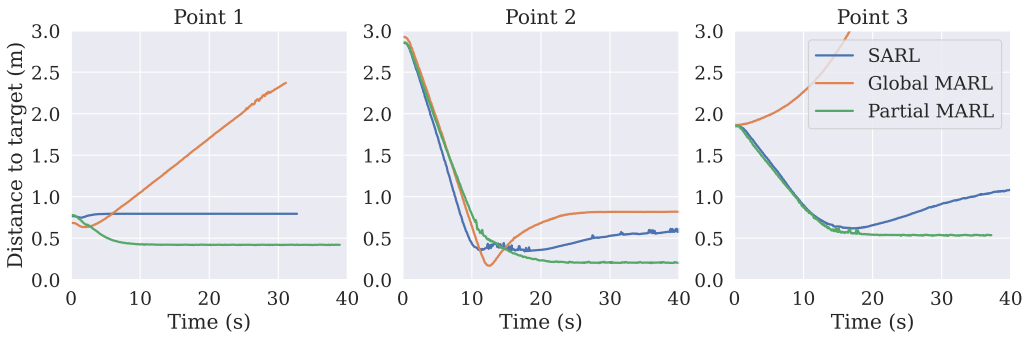

Figure 6.5: Results with a mobile manipulator with the manipulator agent disabled, and the arm repositioned. Only the MARL with partial observations converges. This shows that the introduction of partial observability enables greater system robustness.

7 Conclusion

In this thesis, we explored the idea of dividing the control of the robot among multiple cooperating agents and training them via MARL. The idea was motivated by the limitations of robust robot learning and the advances in multi-agent systems such as modular and swarm robotics. Based on these motivations, we selected the following research questions:

1. Is it possible to use MARL to train policies for multiple agents controlling a single robot and deploy them in a real-world mobile manipulator?
2. Is zero-shot sim-to-real transfer possible for this multi-agent system?
3. Do the multi-agent policies perform better or more robustly than the single-agent counterpart?

To answer these questions, we developed two methods for modeling and training this type of multi-agent system. We implemented a training scheme for a mobile manipulation reach task that uses Isaac Sim for zero-shot sim-to-real transfer. This implementation uses multi-agent PPO to train neural network policies that use parameter sharing to improve multi-agent learning. In our experiments, we compared the learning speed of the multi-agent approaches to the single-agent approach. We conducted a series of real-world experiments to evaluate the zero-shot transfer performance and the robustness of the methods.

Our experiments show that the multi-agent methods can lead to better performance

during training and that the multi-agent approach using partial observations can lead to greater robustness. However, more and different experiments are needed to draw solid conclusions about performance gains or increased robustness.

Potential areas for future research include investigating how the proposed methods perform in situations that require a different kind of generalization or robustness; a more formal analysis of why and in which cases partial observability helps; and studying the contribution of different implementation choices to performance gains and increased robustness.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey", *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[3] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602*, 2013.

[5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning", *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[6] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain", *Science robotics*, vol. 5, no. 47, eabc5986, 2020.

[7] Y.-J. Chen, D.-K. Chang, and C. Zhang, "Autonomous tracking using a swarm of uavs: A constrained multi-agent reinforcement learning approach", *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 13 702–13 717, 2020.

[8] Y. Huang, S. Wu, Z. Mu, X. Long, S. Chu, and G. Zhao, "A multi-agent reinforcement learning method for swarm robots in space collaborative exploration", in *2020 6th international conference on control, automation and robotics (ICCAR)*, IEEE, 2020, pp. 139–144.

[9] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: A review from the swarm engineering perspective", *Swarm Intelligence*, vol. 7, pp. 1–41, 2013.

[10] J. Liu, X. Zhang, and G. Hao, "Survey on research and development of reconfigurable modular robots", *Advances in Mechanical Engineering*, vol. 8, no. 8, p. 1 687 814 016 659 597, 2016.

[11] B. Peng, T. Rashid, C. Schroeder de Witt, *et al.*, "Facmac: Factored multi-agent centralised policy gradients", *Advances in Neural Information Processing Systems*, vol. 34, pp. 12 208–12 221, 2021.

[12] Z. Fu, X. Cheng, and D. Pathak, "Deep whole-body control: Learning a unified policy for manipulation and locomotion", in *Conference on Robot Learning*, PMLR, 2023, pp. 138–149.

[13] J. Hwangbo, J. Lee, A. Dosovitskiy, *et al.*, "Learning agile and dynamic motor skills for legged robots", *Science Robotics*, vol. 4, no. 26, eaau5872, 2019.

[14] OpenAI, M. Andrychowicz, B. Baker, *et al.*, "Learning dexterous in-hand manipulation", *CoRR*, 2018. [Online]. Available: `http://arxiv.org/abs/1808.00177`.

[15] T. Haarnoja, B. Moran, G. Lever, *et al.*, "Learning agile soccer skills for a bipedal robot with deep reinforcement learning", *arXiv preprint arXiv:2304.13653*, 2023.

[16] E. Candela, L. Parada, L. Marques, T.-A. Georgescu, Y. Demiris, and P. Angeloudis, "Transferring multi-agent reinforcement learning policies for autonomous driv-

ing using sim-to-real", in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022, pp. 8814–8820.

[17] C. Wang, Q. Zhang, Q. Tian, *et al.*, "Learning mobile manipulation through deep reinforcement learning", *Sensors*, vol. 20, no. 3, p. 939, 2020.

[18] A. Wong, T. Bäck, A. V. Kononova, and A. Plaat, "Deep multiagent reinforcement learning: Challenges and directions", *Artificial Intelligence Review*, pp. 1–34, 2022.

[19] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms", *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[20] J. Schulman, "Optimizing expectations: From deep reinforcement learning to stochastic computation graphs", Ph.D. dissertation, UC Berkeley, 2016.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *arXiv preprint arXiv:1707.06347*, 2017.

[22] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation", *arXiv preprint arXiv:1506.02438*, 2015.

[23] C. Boutilier, "Planning, learning and coordination in multiagent decision processes", in *TARK*, Citeseer, vol. 96, 1996, pp. 195–210.

[24] L. Canese, G. C. Cardarilli, L. Di Nunzio, *et al.*, "Multi-agent reinforcement learning: A review of challenges and applications", *Applied Sciences*, vol. 11, no. 11, p. 4948, 2021.

[25] C. S. de Witt, T. Gupta, D. Makoviichuk, *et al.*, "Is independent learning all you need in the starcraft multi-agent challenge?", *arXiv preprint arXiv:2011.09533*, 2020.

[26] J. K. Gupta, M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning", in *Autonomous Agents and Multiagent Systems: AAMAS 2017 Workshops, Best Papers, São Paulo, Brazil, May 8-12, 2017, Revised Selected Papers 16*, Springer, 2017, pp. 66–83.

[27] C. Yu, A. Velu, E. Vinitsky, *et al.*, "The surprising effectiveness of ppo in cooperative multi-agent games", *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 611–24 624, 2022.

[28] P. Sunehag, G. Lever, A. Gruslys, *et al.*, *Value-decomposition networks for cooperative multi-agent learning*, 2017. arXiv: 1706.05296 [cs.AI].

[29] T. Rashid, M. Samvelyan, C. S. De Witt, G. Farquhar, J. Foerster, and S. Whiteson, "Monotonic value function factorisation for deep multi-agent reinforcement learning", *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7234–7284, 2020.

[30] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey", in *2020 IEEE symposium series on computational intelligence (SSCI)*, IEEE, 2020, pp. 737–744.

[31] R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel, "A survey of zero-shot generalisation in deep reinforcement learning", *Journal of Artificial Intelligence Research*, vol. 76, pp. 201–264, 2023.

[32] Open Source Robotics Foundation, *Robot Operating System*, https://www.ros.org/, Accessed: 2023-10-10.

[33] NVIDIA, *Isaac Sim*, https://developer.nvidia.com/isaac-sim, Accessed: 2023-03-15.

[34] NVIDIA, *Omniverse Isaac Gym Reinforcement Learning Environments for Isaac Sim*, https://github.com/NVIDIA-Omniverse/OmniIsaacGymEnvs, Accessed: 2023-05-29.

[35] Franka Emika, *Franka Control Interface*, `https://frankaemika.github.io/docs/overview.html`, Accessed: 2023-09-19.

[36] W. Zhao, E.-A. Rantala, J. Pajarinen, and J. P. Queralta, "Less is more: Robust robot learning via partially observable multi-agent reinforcement learning", *arXiv preprint arXiv:2309.14792*, 2023.

[37] J. K. Terry, N. Grammel, A. Hari, L. Santos, and B. Black, "Revisiting parameter sharing in multi-agent deep reinforcement learning", *arXiv preprint arXiv:2005.13625*, 2020.

[38] J. Moos, K. Hansel, H. Abdulsamad, S. Stark, D. Clever, and J. Peters, "Robust reinforcement learning: A review of foundations and recent advances", *Machine Learning and Knowledge Extraction*, vol. 4, no. 1, pp. 276–315, 2022.

[39] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning", in *Machine learning proceedings 1994*, Elsevier, 1994, pp. 157–163.