Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

A.y. 2024/2025

Graduation Session December 2025

Deep Reinforcement Learning for Robotic Manipulation on UR10e

From Simulation to Real Deployment

Supervisor
Prof. Giuseppe Bruno Averta

Candidate
Federico Pretini

Alla mia famiglia

Abstract

Robotic manipulation is widely adopted in industry but typically assumes predictable, tightly structured workspaces, where classical motion-planning pipelines can work safely, reducing as much as possible the occurrence of failures when used to control robots. However, when a failure occurs e.g. missed grasps, object slippage, or unforeseen perturbations, explicit detection and handling are required, making this approach fragile in unstructured settings. In parallel, advances in hardware such as graphics processing units (GPUs) and in machine learning have made Reinforcement Learning (RL) a practical option to learn, via trial and error, policies that directly plan and control motion while exhibiting recovery behaviors. In fact, unlike classical pipelines, an RL policy acts as a closed-loop controller that adapts online to perturbations and unexpected events without the need to exhaustively hard-coding failure cases.

This thesis presents an end-to-end pipeline for training, simulation-based validation, and deployment to a real robot of manipulation policies for a UR10e with a Robotiq 2F-140 gripper. In simulation, training is performed in Isaac Lab/Sim using the Proximal Policy Optimization (PPO) algorithm implemented in the RSL-RL library on three tasks of increasing complexity (Reach, Lift, OpenDrawer). The approach leverages manager-based environments for reusability and portability, and domain randomization to improve policy robustness. Intermediate validation uses a containerized digital environment based on URSim, orchestrated with Docker and ROS 2, which mirrors the controller-level stack and enables deterministic policy replay without hardware. Finally, deployment to the physical robot reuses the same code used for simulation validation, with minor adjustments for gripper integration, on a host running a real-time Linux kernel to improve timing determinism.

The results demonstrate successful transfer of the reach and lift task policies, with effective execution of the tasks on the real robot. The open-drawer tasks show promising initial results, with successful policy execution in simulation indicating potential for further development. Overall, this work beypond demonstrating a zero-shot RL pipeline from simulation to reality, highlight that sequential reward terms can interact and conflict as task complexity increases. Future work will therefore explore Imitation Learning and Inverse Reinforcement Learnin to reduce manual reward design and assess UR's new direct torque control interface for torque-level, contact-aware control.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

A2C Advantage Actor-Critic

A3C Asynchronous Advantage Actor-Critic

AOUSD Alliance for OpenUSD

API Application Programming Interface

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DDPG Deep Deterministic Policy Gradient

DH Denavit-Hartenberg

DOF Degrees of Freedom

DQN Deep Q-Network

DRL Deep Reinforcement Learning

EE End-Effector

ELU Exponential Linear Unit

FK Forward Kinematics

GAE Generalized Advantage Estimation

GPU Graphics Processing Unit

IO Input/Output

IP Internet Protocol

KL Kullback-Leibler divergence

LAN Local Area Network

MDP Markov Decision Process

MJCF MuJoCo XML format

MLP Multi-Layer Perceptron

PPO Proximal Policy Optimization

PREEMPT_RT Real-time preemption patch for the Linux kernel

RL Reinforcement Learning

RSL-RL Robotic Systems Lab Reinforcement Learning library

RTDE Real-Time Data Exchange

ROS Robot Operating System

ROS 2 Robot Operating System 2

RT Real-Time

Sim2Real Simulation-to-Reality transfer

Sim2Sim Simulation-to-Simulation transfer

TCP Tool Center Point

TD Temporal Difference

TRPO Trust Region Policy Optimization

UR Universal Robots

URCap UR Capsule

URDF Unified Robot Description Format

URSim Universal Robots Simulator

USD Universal Scene Description

Chapter 1

Introduction

Industrial robotic manipulators are most effective when they operate in structured workspace, where the environment, the obstacles, the objects, and the sequence of actions are known in advance and motion can be generated with classical planning and control pipelines [1, 2, 3, 4, 5]. In these scenarios, inverse kinematics and sampling-based planners provide safe and repeatable trajectories and have therefore become the default solution in industrial practice and in ROS-based frameworks such as MoveIt [2]. The success of these pipelines, however, relies on the assumption that the scene does not change in unexpected ways and that the task can be fully encoded in advance. Recent surveys on motion planning for industrial manipulators and on intelligent control report that, as soon as the robot must tolerate small variations such as slightly displaced objects, imperfect grasps, contacts with the environment, purely pre-programmed solutions become harder to maintain, because every possible deviation must be explicitly detected and handled [1, 6, 7]. Classical model-based controllers are accurate but mathematically heavy and not always convenient in dynamic environments; for this reason, data-driven or learning-based techniques are increasingly adopted to cope with nonlinearities, uncertainties, and changing conditions [6]. At the same time, integrated task-and-motion planning approaches show that, when manipulation becomes multi-stage, the amount of explicit logic the engineer has to write grows quickly [8]. This is exactly the type of brittleness that motivates learning-based, closed-loop policies. In parallel, the availability of high-fidelity, GPU-accelerated simulators such as NVIDIA Isaac Sim [9] and the robot-learning framework Isaac Lab [10] has made it practically feasible to train manipulation policies directly in simulation and then validate them before deployment on the real platform. On the learning side, modern deep reinforcement learning (DRL) algorithms such as Proximal Policy Optimization (PPO), a well-known on-policy actor–critic method, provide a stable and widely adopted way to learn closed-loop controllers from trial and error [11]. PPO and closely related variants have been successfully used in several recent works that achieve non-trivial

real-world performance in contact-rich manipulation, e.g. industrial assembly [12] and dexterous in-hand object reorientation [13, 14]. Unlike classical pipelines, a DRL policy observes the current state and can react online to modest perturbations or execution inaccuracies, without exhaustively hard-coding all exceptional cases, as already demonstrated in learning-based manipulation works on visuomotor control and robust grasping [15, 16]. Very recent surveys on DRL in real robotic settings further report that the most mature applications follow exactly this simulator-centric, dense-reward workflow, which aligns with the approach adopted in this thesis [17]. At the same time, classic overviews of RL in robotics emphasize that reward shaping and sample collection remain key practical bottlenecks [18]. In this perspective, the thesis aims to demonstrate that it is possible to start from policies trained entirely in Isaac Lab on top of Isaac Sim and bring them, with minimal changes, onto a real UR10e. Concretely, the three manipulation policies are first trained and checked inside Isaac Lab, where the full task environment (e.g. objects, rewards, terminations) is available. Once their general behavior is verified in simulation, the same policies are executed in a containerized Universal Robot Simulator (URSim) instance integrated with the Robot Operating System (ROS 2): this step does not reproduce the task environment,but it allows us to develop and test the deployment code that will later communicate with the real robot through ROS 2 nodes, and to verify that the UR controller executes the arm motion consistently, without risking damage to equipment or harm to people. After this controller-level validation, the very same code is run on the physical UR10e, adding only the ROS 2 node that manages the Robotiq gripper. The main contributions of this thesis are threefold.

First, as previously mentioned, it implements a complete training-to-deployment workflow that starts from policy training and task definition in Isaac Lab/Sim, continues with a first functional check in the same simulator, and then reuses the learned policies inside a containerized URSim instance integrated with ROS 2. Once the deployment code is tested on the simulator it's directly executed on the real UR10e.

Second, it defines three classes of manipulation tasks with increasing complexity in terms of required observations, reward shaping, and training time. Two of these tasks (*Reach* and *Lift*) are taken all the way through the pipeline up to the real robot, thus validating the proposed workflow end-to-end, while the third task (*OpenDrawer*) is validated only in simulation and serves to motivate future work.

Third, it provides a detailed analysis of the training dynamics, sim-to-sim transfer, and sim-to-real transfer for each task, discussing the main challenges encountered and the lessons learned during the project.

1.1 Thesis outline

This work is structured as follows:

- **Chapter 2** introduces the theoretical background on reinforcement learning and robotic manipulation, and surveys the literature most relevant to this work.
- **Chapter 3** presents the methodological framework, describing the simulation and deployment tools, the PPO-based learning setup, and the manager-based environment design in Isaac Lab.
- **Chapter 4** details the training in simulation, including the agent configuration and the definition of the three manipulation tasks (Reach, Lift, OpenDrawer), together with their observation, action, reward, and termination models.
- **Chapter 5** describes the validation pipeline and the sim-to-real transfer strategy, from policy replay in Isaac Sim to controller-level tests in URSim and deployment on the physical UR10e.
- **Chapter 6** reports the experimental results, comparing performance across simulation and the real robot. Furthermore, it analyzes sim-to-real discrepancies.
- **Chapter 7** summarizes the main conclusions and contributions of the thesis. It also outlines possible directions for future work and extensions of the proposed pipeline.

Chapter 2

Background and Literature Review

The goal of this chapter is to provide the theoretical and methodological background needed to frame the thesis within the broader landscape of learning-based robotic manipulation. Rather than giving a purely historical survey, the material is organized around three questions: (i) what makes Reinforcement Learning (RL) different from other machine-learning paradigms, (ii) which classes of RL algorithms are practically relevant for robotic control, and (iii) how existing deep RL approaches have been used for manipulation, and where this thesis fits within that space.

We begin with an overview of RL, highlighting its main characteristics in comparison with supervised and unsupervised learning, and introducing the key elements of any RL problem. We then move to a more formal treatment based on multi-armed bandits and Markov Decision Processes (MDPs), introducing the notions of return, value functions, and Bellman equations. These concepts will later be used to explain how actor-critic methods, and PPO in particular, estimate and exploit value information during training.

Building on this foundation, the chapter reviews a taxonomy of RL algorithms that is commonly adopted in robotics, distinguishing model-free from model-based approaches, and value-based from policy-based (policy-gradient) methods. Within this taxonomy, we motivate the choice of Proximal Policy Optimization (PPO) as the learning algorithm used in this thesis, and we summarize its main ingredients: policy gradients with advantage estimation, trust-region ideas, the clipped surrogate objective, and entropy regularization. The focus is on the aspects that are most relevant for training continuous-control policies for robotic manipulators.

The last part of the chapter connects this algorithmic view with recent applications of deep RL to robotic manipulation. Drawing on recent surveys on real-world DRL [17], we outline the main dimensions along which manipulation tasks are

classified and we position the present work. Finally, a brief recap of manipulator kinematics is provided, to fix the notation used in later chapters when defining task frames and targets for the considered manipulation tasks.

2.1 Reinforcement Learning Overview

To learn, one must experiment and learn from the feedback provided by the environment. This is the idea behind any learning mechanism we are familiar with, and Reinforcement Learning (RL) translates this idea into a *computational* form. It enables an agent to learn how to perform the best action given its current state, with the goal of maximizing a reward defined with respect to a particular objective to be achieved. RL represents the third paradigm of machine learning and its two main characteristics that differentiate it from the other two (Supervised Learning and Unsupervised Learning) are:

- **Trial-and-error search**: the agent is not given the correct actions that maximize its reward; instead, it discovers them by trying.
- **Delayed reward**: in more interesting and complex cases, the agent's choice of an action affects not only the immediate reward but also all the rewards it may obtain in the future.

The concept of delayed reward brings a non-trivial problem: the agent must be induced not to always prefer actions that deliver immediate reward, because such behavior does not necessarily maximize the total return. Rather, it must choose actions that, in the *long term*, lead to maximal return. To see why, note that in non-trivial cases certain action choices can preclude the agent from achieving the maximum reward, even if up to that point it has always selected the action yielding the highest immediate reward.

This problem is one of the major challenges in reinforcement learning and is addressed by balancing two key aspects when choosing actions:

- **Exploitation**: the agent should select actions it has already tried in order to obtain rewards;
- **Exploration**: the agent should search for new actions that may lead to a larger cumulative reward.

2.1.1 Learning Paradigms in Comparison

What has been said about reinforcement learning helps clarify that the well-known paradigms of Supervised and Unsupervised Learning share some traits with RL

but also have substantial differences. This supports the view that RL constitutes a third paradigm of Machine Learning. **Supervised Learning** trains on a set of labeled examples provided by an expert. The goal is to learn a general rule that enables correct responses even on examples not present in the training set. In Reinforcement Learning, by contrast, where learning occurs via interaction, it is often impossible to obtain examples of a desired behavior that are both correct and representative of all the possible situations in which the agent may find itself interacting. As noted above, with Reinforcement Learning we aim to maximize a given reward without initially knowing which actions are correct. At first glance this might seem similar to **Unsupervised Learning**, but the objective is different: unlabeled examples are not used to find actions that maximize a reward function; rather, they are used to categorize and group data.

In the Table 2.1, we summarize the main differences among the three learning paradigms.

	SL	***UL***	***RL***
Input	Labeled data	Unlabeled data	Environment state
Objective	Minimize prediction error	Discover latent structure in data	Maximize cumulative reward
Feedback	Direct (label)	No external feedback	Scalar reward from the environment
Example	Image classification	Customer clustering	Training a robot

Table 2.1: Comparison among supervised, unsupervised, and reinforcement learning.

2.1.2 Key Elements of Reinforcement Learning

Below are the key elements underlying any RL algorithm. As noted, an RL system consists of an **agent** interacting with an **environment**. In addition to these two core components, we have:

- **Policy** (π): maps the current state of the environment to an action to be executed. The policy can be deterministic (maps a state to a single action) or stochastic (maps a state to a probability distribution over actions). The

agent's goal is to learn an optimal policy that maximizes the sum of future rewards.

- **Reward (r)**: a scalar sent by the environment to the agent as feedback for the action executed at the current step. The agent aims to maximize the sum of future rewards. If, following an action, the agent receives a negative reward, the policy is updated to reduce the probability of choosing that action in the future.

- **Value functions (V, Q)**: while the reward indicates the agent's immediate gain, the value function estimates the expected future return from a state (V) or from a state-action pair (Q), under a given policy. A state could yield high immediate reward yet lead to low-reward states later, whereas a state with low immediate reward could lead to states with high future rewards. Value functions help the agent make more informed decisions than those based on immediate reward alone.

- **Model**: when available, it predicts transition dynamics and rewards. *Model-based* methods can plan by simulating the environment, whereas *model-free* methods learn directly from interaction without an explicit model. Hybrid methods build an approximate model during interaction and use it for planning.

2.1.3 Mathematical Foundations of Reinforcement Learning

We start from the simplest setting, the multi-armed bandit, in which there is no notion of state and the agent only has to balance exploration and exploitation. This allows us to isolate the core learning problem and to emphasize the role of trial-and-error search. We then move to the more general case of sequential decision making, modeled as a Markov Decision Process (MDP), where the agent interacts with an evolving environment over multiple time steps and the return is the discounted sum of future rewards. This framework naturally leads to the definition of value functions and Bellman equations, which underpin most modern RL algorithms, including the actor-critic methods used in this thesis.

Multi-Armed Bandits

The *multi-armed bandit* problem represents the simplest form of reinforcement learning. An agent repeatedly chooses among k actions (slot-machine arms), each associated with an unknown reward distribution. The objective is to maximize the sum of obtained rewards.

Let $a_t \in \{1, \ldots, k\}$ be the action chosen at time t and r_t the reward obtained; the **expected value** of an action is

$$q_*(a) = \mathbb{E}[r_t \mid a_t = a].$$

The agent's task is to estimate $q_*(a)$ and choose the best action.

This setting naturally introduces the **exploration-exploitation dilemma**:

- **exploitation**: choose the action with the highest estimated value;
- **exploration**: try other actions to improve the value estimates.

Classic strategies include:

- ϵ-greedy: with probability $1 - \epsilon$ choose the best action; with probability ϵ choose a random action;
- Upper Confidence Bound (UCB): balances exploitation and exploration via a statistical confidence bonus.

Although the manipulation tasks considered in this thesis are far more complex than bandits, the same tension between exploiting known good behaviors and exploring new ones remains central, and will reappear in the discussion of entropy regularization in PPO.

Markov Decision Processes (MDPs)

A general RL problem is modeled as a **Markov Decision Process (MDP)** defined by the tuple

$$M = (S, A, P, R, \gamma)$$

where:

- S: set of possible states;
- A: set of possible actions;
- $P(s' \mid s, a)$: transition probability to state s' when taking action a in state s;
- $R(s, a)$: expected reward for taking a in s;
- $\gamma \in [0,1]$: discount factor balancing immediate and future rewards.

At each time step t, the agent observes a state $s_t \in S$, selects an action $a_t \in A$ according to its policy, receives a scalar reward r_{t+1}, and transitions to a new state s_{t+1} according to P. The expected return from time t is defined as

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

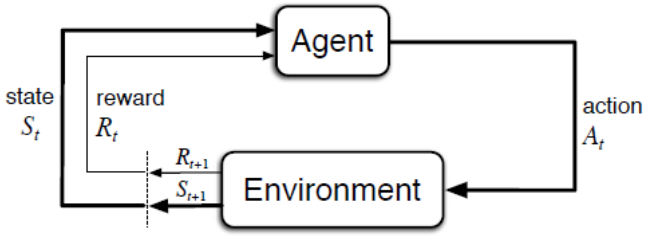


Figure 2.1: The agent-environment interaction in an MDP [19].

The objective is to find a **policy** $\pi(a \mid s)$ that maximizes the expected return. A comprehensive figure illustrating the MDP framework is shown in Figure 2.1.

In the manipulation tasks studied in this thesis, s_t contains low-dimensional proprioceptive and task-related features (e.g., joint positions and object pose), a_t is a continuous joint-space command suitable for the UR10e controller, and rewards are designed to encode intermediate and final task objectives.

To reason about long-term performance in an MDP, it is convenient to introduce the *value functions* associated with a given policy π. The *state-value function* V^π and the *action-value function* Q^π are defined as

$$V^\pi(s) = \mathbb{E}_\pi\Big[\, G_t \,\Big|\, s_t = s\Big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right], \tag{2.1}$$

$$Q^\pi(s,a) = \mathbb{E}_\pi\Big[\, G_t \,\Big|\, s_t = s,\, a_t = a\Big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s,\, a_t = a\right]. \tag{2.2}$$

Intuitively, $V^\pi(s)$ measures how good it is to start from state s and then follow π, while $Q^\pi(s,a)$ measures how good it is to take action a in s and then continue according to π.

These functions satisfy a set of recursive relations known as *Bellman equations*. For the state-value function, the Bellman expectation equation are:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\mathbb{E}_{s'\sim P(\cdot|s,a)}\Big[\, R(s,a) + \gamma\, V^\pi(s') \,\Big]. \tag{2.3}$$

Similarly, the Bellman equation for Q^π is

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim P(\cdot|s,a)}\mathbb{E}_{a'\sim\pi(\cdot|s')}\Big[\, R(s,a) + \gamma\, Q^\pi(s',a') \,\Big]. \tag{2.4}$$

The optimal value functions $V^\star$ and $Q^\star$ are defined as $V^\star(s) = \max_\pi V^\pi(s)$ and $Q^\star(s,a) = \max_\pi Q^\pi(s,a)$. They satisfy the Bellman *optimality* equations:

$$V^\star(s) = \max_{a\in A} \sum_{s'\in S} P(s' \mid s,a)\,\Big[R(s,a) + \gamma\, V^\star(s')\Big], \tag{2.5}$$

$$Q^\star(s,a) = \sum_{s' \in S} P(s' \mid s,a) \Big[R(s,a) + \gamma \max_{a' \in A} Q^\star(s',a') \Big]. \tag{2.6}$$

These relations form the basis of dynamic programming algorithms such as value iteration and policy iteration. In model-free deep RL, and in actor-critic methods in particular, a critic network is trained to approximate V^π so that they approximately satisfy the corresponding Bellman equations, while the actor (policy) is updated using these value estimates as a training signal.

2.2 Taxonomy of reinforcement learning algorithms

A common way to organize reinforcement learning (RL) methods in robotics is to use a layered taxonomy that clarifies which families of algorithms are practical for real or sim-to-real manipulation tasks. A possible taxonomy is the one discussed in [20] where RL algorithms are divided into **model-free** and **model-based**.

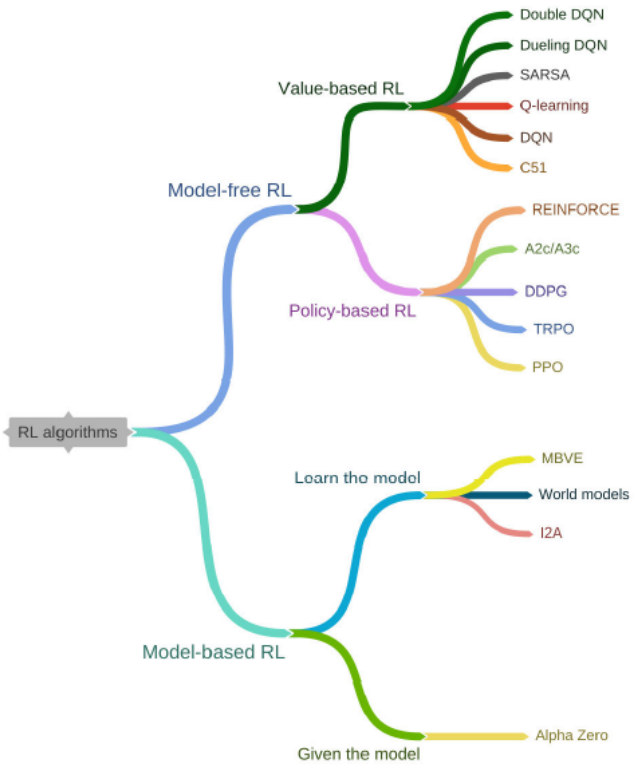


Figure 2.2: Taxonomy of reinforcement learning algorithms proposed by [20]

Model-free methods learn a control policy or a value function directly from interaction, without explicitly constructing a dynamics model. This makes them attractive for robotics, where accurate models are not always available but large amounts of data can be generated in simulation. Within model-free RL, two main subfamilies are typically distinguished:

- **Value-based** methods learn an action-value function $Q(s, a)$ (or a state-value function $V(s)$) and derive the policy by selecting the action that maximizes this value. Classical examples include Q-learning, DQN, and Double DQN. These methods have been very successful in discrete-action domains, but become less natural when the robot has to output continuous joint commands [18].
- **Policy-based** (or policy-gradient) methods directly parametrize the policy $\pi_\theta(a \mid s)$ and optimize it with respect to the expected return[1]. Algorithms such as REINFORCE, A2C/A3C, DDPG, TRPO, and PPO belong to this branch. They are generally better suited for robotic manipulation because they handle continuous action spaces and produce smoother control signals, which match the requirements of industrial and collaborative arms [18, 11].

Model-based methods, on the other hand, assume that a dynamics model $p(s' \mid s, a)$ is available ("given the model") or attempt to learn such a model from data ("learn the model") and then use it for planning, prediction, or to generate additional rollouts. While model-based RL can be more sample-efficient, it typically requires more modeling effort and is less commonly adopted in day-to-day robotic manipulation pipelines [18].

2.2.1 Proximal Policy Optimization (PPO)

In the rest of this work, and consistently with recent surveys on real-world DRL for robotics [17], we focus on the *model-free, policy-based* branch, and in particular on Proximal Policy Optimization (PPO) [11], which has become the de facto standard in several simulator-centric robot learning frameworks. PPO, in particular, is an on-policy actor-critic algorithm: an actor network outputs a distribution over continuous actions, while a critic network estimates the state value $V(s)$ to compute advantages and reduce variance. The algorithm optimizes a surrogate objective designed to keep policy updates within a trust region, thus ensuring stable and

[1] In the broader RL literature, 'policy-based' can also refer to methods that optimize a parameterized policy without explicitly using gradients (e.g., evolutionary strategies). In this thesis, we only consider gradient-based methods, so we use the terms 'policy-based' and 'policy-gradient' interchangeably.

efficient learning. This is achieved by clipping the probability ratio between the new and old policies, preventing excessively large updates that could destabilize training.

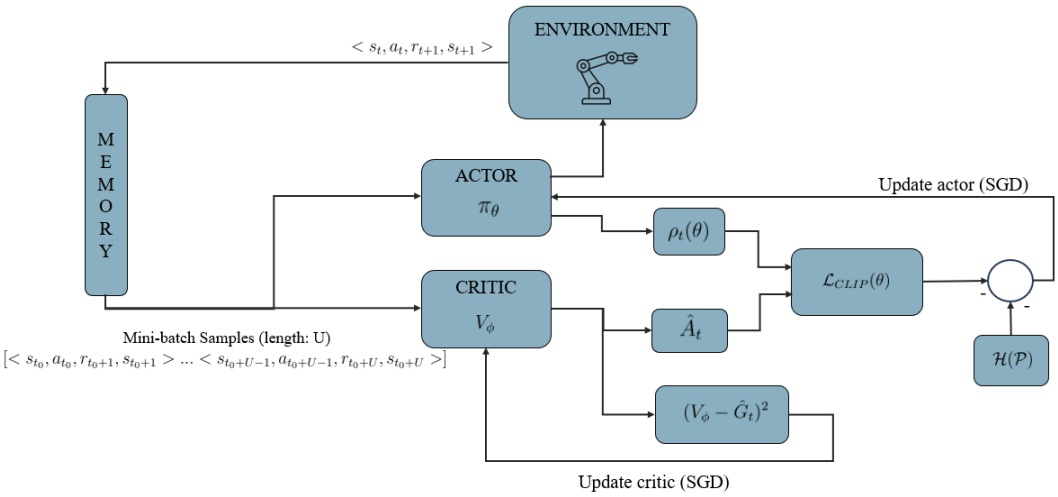


Figure 2.3: Schematic overview of the Proximal Policy Optimization (PPO) update loop, adapted from [21].

Motivation

The evolution of policy-gradient methods has been driven by the need to reduce variance, improve sample efficiency, and stabilize policy updates. The path leading to PPO can be summarized as follows.

1) Policy gradient theorem and variance In its simplest form, the policy gradient theorem states that, for a parametrized policy $\pi_\theta(a \mid s)$, the gradient of the expected return $J(\theta)$ can be written as

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_\pi\Big[\,\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t\,\Big], \tag{2.7}$$

where G_t is the discounted return from time t. This expression is unbiased but typically exhibits high variance, requiring many trajectories to obtain stable updates.

2) Baseline and advantage To reduce variance without introducing bias, it is common to subtract a baseline that does not depend on the action, typically the

value function $V^\pi(s_t)$. This leads to the *advantage* function

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{2.8}$$

and to the practical gradient estimator

$$\nabla_\theta J(\theta) \;\approx\; \mathbb{E}_\pi\Big[\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A^\pi(s_t, a_t) \,\Big], \tag{2.9}$$

which forms the basis of actor-critic methods: a critic network estimates V^π (or Q^π), and the actor is updated in the direction suggested by the estimated advantages.

3) Generalized Advantage Estimation (GAE) In deep RL, advantages are estimated from sampled trajectories. A widely used estimator is *Generalized Advantage Estimation* (GAE) [22], which combines multi-step temporal-difference residuals to balance bias and variance. Let

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{2.10}$$

be the TD residual under the current value approximation V_ϕ. GAE defines

$$\hat{A}_t \;=\; \sum_{l=0}^{\infty} (\gamma\lambda)^l\, \delta_{t+l}, \tag{2.11}$$

where $\lambda \in [0,1]$ controls the trade-off between low-variance, high-bias (small λ) and high-variance, low-bias (large λ) estimates. In practice the sum is truncated at the horizon of the collected trajectories.

4) Trust regions and TRPO Standard policy-gradient updates can still be unstable if the policy changes too much in a single step. Trust-region methods such as TRPO [23] address this by explicitly constraining the Kullback–Leibler (KL) divergence between the old and new policy, ensuring that each update stays within a small region in policy space.

PPO was introduced to mitigate this complexity. Instead of enforcing a hard KL constraint, PPO uses a *clipped* surrogate objective that implicitly limits the size of policy updates, preserving much of TRPO's stability while retaining a simple first-order optimization scheme. Conceptually, PPO belongs to policy-gradient methods and leverages the same intuition as natural gradients (conservative moves in distribution space), but without explicitly computing second-order information. This makes PPO attractive for large neural networks and for simulator-centric robot learning setups such as the one considered in this thesis.

Clipped surrogate objective

The core idea of PPO is to reformulate the surrogate objective into a shape amenable to first-order optimization while indirectly controlling the change in the policy. Let

$$\rho_t(\theta) \;=\; \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{2.12}$$

be the importance sampling ratio between the new and old policy, and consider the classical policy gradient surrogate

$$\mathcal{L}_{\text{CPI}}(\theta) \;=\; \mathbb{E}\Big[\, \rho_t(\theta)\, A^{\pi_{\theta_{\text{old}}}}(s_t, a_t) \,\Big]. \tag{2.13}$$

Without any constraint, $\rho_t(\theta)$ may drift far from 1, leading to unstable updates. PPO introduces the *clipped* objective

$$\mathcal{L}_{\text{CLIP}}(\theta) \;=\; \mathbb{E}\Big[\, \min\Big(\rho_t(\theta)\,\hat{A}_t,\; \text{clip}(\rho_t(\theta),\, 1-\epsilon,\, 1+\epsilon)\,\hat{A}_t\Big)\Big], \tag{2.14}$$

where $\hat{A}_t$ is an empirical estimate of the advantage (e.g., obtained via GAE) and $\epsilon > 0$ is a small hyperparameter (in [11] they advise setting this value to 0.2). If $\hat{A}_t > 0$, the increase in probability for the advantageous action is capped once ρ_t exceeds $1+\epsilon$; if $\hat{A}_t < 0$, the decrease is limited once ρ_t falls below $1-\epsilon$. In both cases, overly aggressive updates are discouraged, and successive policies are kept sufficiently close, as summarised in Figure 2.3.

Exploration vs. Exploitation in PPO

PPO trains a stochastic policy in an on-policy regime: the same policy that is used to collect trajectories is also updated by gradient ascent. At the beginning of training, the policy typically has high stochasticity and therefore explores a wide region of the state-action space. As optimization progresses and the policy is repeatedly updated in the direction of the estimated advantages, its action distribution tends to become more peaked around high-advantage actions, i.e., more deterministic.

This gradual loss of stochasticity is desirable up to a point (it corresponds to *exploitation*), but if it happens too quickly the agent may get stuck in suboptimal behaviors. To counteract this effect, PPO includes an *entropy bonus* that explicitly encourages the policy to retain some randomness during training. The entropy of a discrete distribution P is defined as

$$\mathcal{H}(P) \;=\; \mathbb{E}_{x\sim P}\Big[\, -\log P(x) \Big], \tag{2.15}$$

and measures how spread out the distribution is. For a stochastic policy $\pi_\theta(\cdot \mid s)$, high entropy means that multiple actions are still assigned non-negligible probability.

In practice, PPO combines three terms in a single loss function: the negative of the clipped surrogate objective, a value-function regression loss, and an entropy bonus term. Denoting by $\hat{G}_t$ the estimated return (used as target for the value function) and by $\mathcal{L}_{\text{CLIP}}(\theta)$ the clipped objective in Eq. (2.14), the total loss minimized during training can be written as

$$\mathcal{L}(\theta, \phi) \;=\; -\hat{\mathbb{E}}_t\Big[\mathcal{L}_{\text{CLIP}}(\theta)\Big] \;+\; c_v\,\hat{\mathbb{E}}_t\Big[\Big(V_\phi(s_t) - \hat{G}_t\Big)^2\Big] \;-\; c_e\,\hat{\mathbb{E}}_t\Big[\mathcal{H}(\pi_\theta(\cdot \mid s_t))\Big], \quad (2.16)$$

where c_v and c_e are scalar coefficients that weight the value loss and the entropy bonus, respectively, and $\hat{\mathbb{E}}_t[\cdot]$ denotes the empirical average over time steps and trajectories. The last term encourages higher-entropy policies, thereby sustaining exploration and preventing the policy from collapsing too early to a nearly deterministic mapping. In all the experiments of this thesis, the entropy coefficient c_e is kept fixed and chosen empirically so as to maintain a reasonable level of exploration without destabilizing training.

2.3 Applications of Deep Reinforcement Learning to robotic manipulation

Recent surveys on real-world deep reinforcement learning for robotics show that manipulation tasks solved with DRL can be described along a small set of recurring dimensions [17]:

- **Platform and task scope**: single-robot manipulation vs. multi-robot settings.
- **Problem formulation**: MDP/POMDP formulations with either full-state or partial observations.
- **Action space**: low-level, continuous actions (e.g., joint torques, joint velocities, Cartesian velocities) vs. higher-level or skill actions (e.g., "go from A to B").
- **Observation space**: low-level proprioceptive/task features vs. high-dimensional vision.
- **Reward design**: dense rewards (e.g., a reward inversely proportional to the distance between the end effector and the object) vs. sparse or success-only rewards (e.g., a reward is given only if the object is reached).
- **Data source and transfer**: real-world training, pure simulation, or zero-shot sim-to-real.
- **Learning paradigm**: model-free vs. model-based; policy-optimization vs. value-based; on-policy vs. off-policy.

- **Policy representation**: lightweight MLPs vs. vision-based encoders (CNN/-Transformer) when images are used.

Within this grid, many successful robotic manipulation systems reported in [17] fall into the model-free branch and use policy-optimization methods, often with dense rewards and simulator support, because this combination offers stable training and direct closed-loop control.

2.3.1 Position of this thesis in the taxonomy

According to the above classification, the work presented in this thesis can be placed very precisely:

- **Platform and task scope**: single-robot manipulation on a UR10e with a Robotiq 2F-140 gripper.
- **Problem formulation**: tasks are modeled as MDP instances in Isaac Lab, with structured low-level observations (robot state + task features).
- **Action space**: low-level, continuous actions compatible with the UR controller (the policy outputs directly the control to be sent through ROS 2).
- **Reward design**: manually designed *dense* rewards, which is the regime in which sim-to-real DRL has been reported to work most reliably [17].
- **Data source / transfer**: training entirely in a high-fidelity simulator (Isaac Lab/Sim), functional check in simulation, controller-level validation in a containerized URSim, and then *zero-shot* deployment on the real UR10e.
- **Learning paradigm**: *model-free*, *policy-optimization*, *on-policy*, using PPO as implemented in the Isaac Lab / RSL-RL stack [11].
- **Policy representation**: MLP-based actor-critic with shared backbone, which is sufficient because observations are low-dimensional.

In other words, this thesis targets the same quadrant highlighted as the most mature in [17]: model-free, policy-based, simulator-supported manipulation with dense rewards, transferred zero-shot to real hardware.

2.4 Kinematic of Manipulators

This section provides a brief overview of the kinematic modeling of robotic manipulators, knowledge that will be useful in Chapter 4 when defining the target frame for reach task for the UR10e robot.

The kinematics of a robotic manipulator put in relationships the internal state of the robot expressed in the joint space that can be easily measured (joint positions) with the position and orientation (pose) of the end-effector in the workspace. This relationship is irrespective of the forces and torques acting on the robot, which are instead studied in the field of dynamics.

We can define two types of kinematics:

- **Forward kinematics**: given the joint parameters (angles for revolute joints, displacements for prismatic joints), it computes the position and orientation of the end-effector.
- **Inverse kinematics**: given a desired position and orientation of the end-effector, it computes the necessary joint parameters to achieve that pose.

Since the Inverse Kinematics will not be directly used in this thesis, we will focus on the Forward Kinematics only.

2.4.1 Forward Kinematics

Every robot can be represented as a series of rigid links connected by joints that can be mainly divided in 2 classes:

- **Revolute joints**: allow rotational movement around a fixed axis.
- **Prismatic joints**: allow linear movement along a fixed axis.

Focusing on the Denavit-Hartenberg (DH) convention for representing the spatial relationships between consecutive links.

The DH convention assigns a coordinate frame to each link of the manipulator, allowing for a systematic description of the robot's geometry. Each frame is defined by four parameters:

- a_i: the length of the common normal (distance between Z_{i-1} and Z_i axes).
- α_i: the angle between Z_{i-1} and Z_i axes, measured about the X_i axis.
- d_i: the offset along the previous Z_{i-1} axis to the common normal.
- θ_i: the angle between X_{i-1} and X_i axes, measured about the Z_{i-1} axis.

Using these parameters, the transformation matrix from frame $i-1$ to frame i can be expressed as:

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The overall transformation from the base frame to the end-effector frame is obtained by multiplying the individual transformation matrices:

$$T_n^0 = T_1^0 T_2^1 \cdots T_n^{n-1}$$

This transformation matrix encapsulates both the position and orientation of the end-effector relative to the base frame, which is essential for tasks such as reaching, grasping, and manipulating objects in the robot's workspace. For more detail about the kinematic of robotic manipulators, refer to [24].

The FK of the UR10e robot can be computed using the DH parameters provided in Table 2.2 , those values and the figure 2.4are taken from the official documentation [25].

Joint	a [m]	d [m]	α [rad]	θ
Joint 1	0	0.1807	$\pi/2$	q_1
Joint 2	-0.6127	0	0	q_2
Joint 3	-0.57155	0	0	q_3
Joint 4	0	0.17415	$\pi/2$	q_4
Joint 5	0	0.11985	$-\pi/2$	q_5
Joint 6	0	0.11655	0	q_6

Table 2.2: Denavit Hartenberg parameters for UR10e provided by Universal Robots

The FK can then be computed by multiplying the transformation matrices for each joint using the DH parameters from Table 2.2. This results in a final transformation matrix that describes the position and orientation of the end-effector in the base frame, given the joint angles q_1 to q_6.

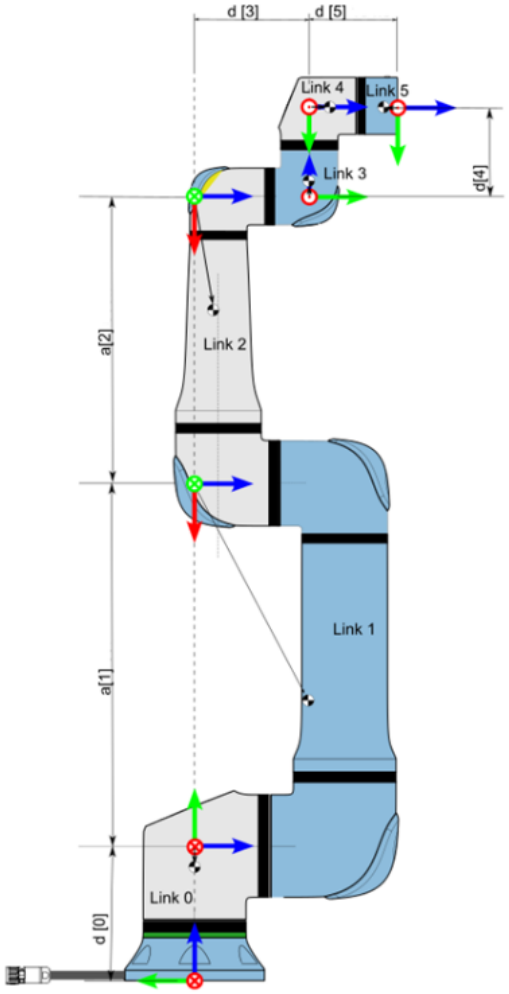


Figure 2.4: Denavit-Hartenberg coordinate frames for UR10e taken from [25]

Chapter 3

Methodologies

In this chapter, we describe the methodologies and tools employed throughout this thesis for training, validating, and deploying deep reinforcement learning policies for robotic manipulation tasks. The section is organized as follows: first, we introduce the training tools and the architecture used to train the policies in simulation; next, we discuss the deployment architecture and tools that facilitate the transfer of learned policies to the real UR10e robotic arm.

3.1 Training Architecture and Tools

The training phase of this work relies on NVIDIA Isaac Sim [9] and the Isaac Lab framework [10] to create realistic simulation environments and to implement the reinforcement learning pipeline. The overall training architecture is shown in Figure 3.1, while the main components used in this process are briefly introduced below.

The components of this architecture are summarised in the following paragraphs. A more detailed configuration of the two main blocks, the *Agent* and the *Environment*, is provided in Chapter 4 for each of the three tasks.

3.1.1 Isaac Sim

Isaac Sim[9] is NVIDIA's robotics simulator built on Omniverse, designed to develop, test, and validate robots and control algorithms in physically plausible virtual environments. At its core is USD (Universal Scene Description), which makes it straightforward to compose complex scenes, version assets, and integrate models from common formats such as URDF and MJCF. Dynamics are handled by PhysX[27] with GPU support for large-scale parallel simulations, enabling rapid data collection for training machine-learning models. Isaac Sim also includes a

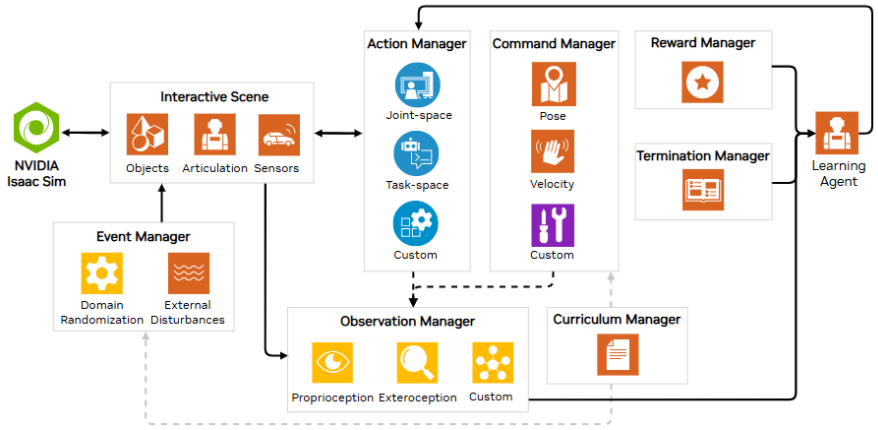


Figure 3.1: Manager-based training architecture in Isaac Lab, reproduced from [26].

suite of tools for sensor simulation, robot kinematics, control, and tuning, making it a comprehensive platform for robotics research and development.

Universal Scene Description (USD) Overview

The *Universal Scene Description* (USD) is a modern framework originally developed by Pixar to manage the creation and exchange of complex 3D scenes. It was designed to address a fundamental challenge in computer graphics and simulation: allowing different applications and users to collaborate on the same virtual world without losing data fidelity or structural consistency. USD provides a flexible and extensible format capable of representing geometric models, materials, lights, animations, and even physical simulation parameters in a unified way.

NVIDIA Isaac Sim adopts USD as its foundational layer for representing every element of the simulation world — from robotic manipulators and sensors to objects, environments, and physics materials. This choice is not accidental: USD offers a non-destructive and highly modular data model that allows individual components to be edited, replaced, or extended without altering the integrity of the whole scene. This makes it particularly suited for robotics, where assets such as arms, grippers, cameras, and fixtures must be combined, configured, and reused across multiple experiments.

In its open form, known as *OpenUSD*, the framework is evolving under the governance of the *Alliance for OpenUSD (AOUSD)* [28]. This open standard extends Pixar's original design and promotes USD as a cross-industry ecosystem for 3D data exchange and collaboration. OpenUSD emphasizes three key principles: it enables **non-destructive** editing, where modifications to a scene do not overwrite existing

data; it supports **collaborative** workflows, allowing multiple users or applications to work concurrently on the same project; and it is inherently **standardized**, ensuring interoperability across different software and hardware platforms.

Within this research, USD serves as the backbone of the entire simulation and training workflow. It enables the UR10e and the Robotiq 2F-140 to be combined into a single, coherent model, while maintaining the ability to edit, parameterize, or replace each component independently. In practice, this means that the same USD structure can be used seamlessly for visualization, physics-based simulation, reinforcement learning, and deployment—making it a cornerstone technology for modern robotics research in virtual environments.

Assembler

To define the robotic model used within the simulation environment, it was necessary to integrate the *UR10e* robotic arm and the *Robotiq 2F-140* gripper into a single USD asset. This operation was performed using the *Assembler Tool* provided by *NVIDIA Isaac Sim*, a graphical interface that allows the modular composition of robotic manipulators starting from individual components available in the asset library. The reference procedure is described in the official documentation provided by NVIDIA [29].

The assembler makes it possible to import the manipulator model (in this case, the UR10e) as the base of the system and subsequently add a compatible *end-effector*, either selected from the available grippers catalog or loaded from an external USD file. Once both components are loaded, the interface allows the user to specify the mounting point, i.e., the joint or flange of the robot on which the gripper should be attached. In the case of the Robotiq 2F-140, the operation was carried out by selecting the terminal wrist link (`ee_link`) of the UR10e as the attachment point, ensuring proper spatial alignment between the two models.

After completing the assembly, the tool automatically generates a composed USD file that includes the full link hierarchy, joint definitions, and the necessary transformation references to maintain kinematic consistency throughout the system. This assembled asset was then exported and used as the reference model for defining the training scenes within Isaac Lab, thus enabling realistic simulation of the interaction between the UR10e arm and the Robotiq 2F-140 gripper.

To ensure proper dynamic and kinematic compatibility between the two elements, the correspondence of reference frames (particularly among `tool0`, `ee_link`, and `base_link`) was verified, and appropriate rotation offsets around the Z-axis were applied to harmonize the coordinate conventions used by the ROS2 driver and the real robotic setup.

The final result is a single, modular USD file that can be easily reused and configured, serving as the foundation for all the Reinforcement Learning training

and validation experiments described in the following chapters.

3.1.2 Isaac Lab

Isaac Lab is the robot-learning framework built on top of Isaac Sim. It was chosen for its capability to define modular observations, actions, rewards, curriculum, and termination conditions, improving code reuse across different scenarios. Furthermore, it provides ready-to-use functions for domain randomization and native integrations. Most importantly, it includes the RSL-RL library [30] within its ecosystem, reducing the likelihood of implementation errors when using the PPO algorithm. These goals and capabilities are described in the Isaac Lab whitepaper [31]. Importantly, Isaac Lab is the successor to the deprecated Isaac Gym framework, which offered an high performance platform for robot learning as shown in [32].

Agent

In Isaac Lab, the `PPORunnerCfg` defines the agent as the combination of (i) a rollout runner, (ii) an actor-critic policy architecture, and (iii) a PPO learning algorithm. In the notation of Section 2.2.1, the policy parameters are denoted by θ and the value-function parameters by ϕ, and the various hyperparameters below control how the empirical expectations $\hat{\mathbb{E}}_t[\cdot]$, the advantages $\hat{A}_t$ and the total loss $\mathcal{L}(\theta, \phi)$ are instantiated in practice.

Runner / Rollout (data collection) The runner specifies how trajectories are collected from the environment before each PPO update. Together with the number of parallel environments, these settings determine the batch size used to approximate the on-policy expectations $\hat{\mathbb{E}}_t[\cdot]$ in the policy gradient and in the loss of Eq. (2.16).

- `num_steps_per_env`: on-policy horizon per update (steps collected in each parallel environment before a PPO update, i.e., the time span over which $\hat{\mathbb{E}}_t[\cdot]$ is computed).
- `max_iterations`: maximum number of training updates (overall training budget).
- `save_interval`: checkpoint frequency for reproducibility and resume.
- `experiment_name`: identifier used for logging and artifact paths.
- `empirical_normalization`: whether to normalize observations based on empirical statistics, which can help the optimization of both π_θ and V_ϕ.

Policy architecture As discussed in Section 2.2.1, PPO is based on an actor-critic setup where the policy (actor π_θ) and the value function (critic V_ϕ) are modeled as multilayer perceptrons. The policy network implements the stochastic mapping $\pi_\theta(a_t \mid s_t)$, while the critic network approximates the state value $V_\phi(s_t)$ used to construct the TD residuals δ_t and advantages $\hat{A}_t$.

- `actor_hidden_dims`, `critic_hidden_dims`: widths/depths of the multilayer perceptrons that parametrize the actor π_θ and critic V_ϕ.
- `activation`: nonlinearity used in the MLPs which affects the expressiveness of both π_θ and V_ϕ. In this case we use the ELU[33] activation function, please refer to the Appendix E for more details.
- `init_noise_std`: initial standard deviation of the Gaussian action head, i.e., the initial exploration level of the stochastic policy π_θ.

Learning algorithm (PPO) The learning block specifies how the empirical data are turned into gradient updates for θ and ϕ. In particular, these hyperparameters instantiate the clipped surrogate objective $\mathcal{L}_{\text{CLIP}}(\theta)$ and the total loss $\mathcal{L}(\theta, \phi)$ of Eq. (2.16), including the coefficients c_v and c_e and the trust-region behaviour via ϵ and the KL target.

- `clip_param`: surrogate ratio clipping parameter ϵ in the PPO objective of Eq. (2.14).
- `value_loss_coef`: weight c_v of the critic loss $c_v\hat{\mathbb{E}}_t[(V_\phi(s_t) - \hat{G}_t)^2]$ in the total objective.
- `use_clipped_value_loss`: optionally applies value prediction clipping to V_ϕ for additional stability.
- `entropy_coef`: weight c_e of the entropy bonus term $-c_e\hat{\mathbb{E}}_t[\mathcal{H}(\pi_\theta(\cdot \mid s_t))]$, which controls the exploration pressure.
- `num_learning_epochs`: number of passes over the collected batch per update (how many times $\mathcal{L}(\theta, \phi)$ is optimized on the same data).
- `num_mini_batches`: partitioning of the batch into mini-batches for stochastic optimization.
- `learning_rate`: optimizer step size for the gradient updates of θ and ϕ.
- `schedule`: learning-rate and/or KL adaptation strategy (e.g., fixed or adaptive around a target KL).

- `desired_kl`: target KL-divergence per update, used in adaptive schedules to moderate policy shifts in the space of π_θ.
- `max_grad_norm`: global gradient clipping threshold applied to $\nabla_\theta \mathcal{L}$ and $\nabla_\phi \mathcal{L}$.
- `gamma`, `lam`: discount factor γ and GAE parameter λ used in the construction of the TD residuals δ_t and of the advantage estimates $\hat{A}_t$.

Manager-based environments

In Isaac Lab, manager-based environments[34] instantiate seven managers that, at each step, follow a stable operational order and exchange only typed tensors. This design improves reproducibility and code reuse across tasks and robots. The managers are described below with the relative configuration classes [35]. It is easy to see that the names of some managers correspond to the main components of a Markov Decision Process (MDP), such as *Observation* (or *State*), *Action*, *Reward*.

Observation Manager The *ObservationManager* builds the agent's observation vector $\mathbf{o}_t$ by evaluating a set of terms (typically `ObservationTermCfg`) organized in groups (e.g., *policy*, *critic*, *eval*). Each term is a callable that returns a tensor (one $\mathbf{o}_t$ per parallel environment); optional modifiers and noise models can corrupt signals for robustness, followed by clipping and scaling.

Action Manager The *ActionManager* receives the action tensor (one $\mathbf{a}_t$ per parallel environment) from the learning library, splits it across the registered terms (e.g., joint position/velocity/effort commands, binary gripper commands, configured via `ActionTermCfg`), applies any pre-processing (offsets, scaling, clipping), and then applies the processed actions to the scene assets **before** the physics step.

Reward Manager The *RewardManager* computes the scalar reward r_t as a weighted sum of the registered terms (`RewardTermCfg`). Each term is configured through its own term config and is evaluated at every step; by design, Isaac Lab multiplies each term's weight by the control time step `dt` so that reward magnitudes remain consistent across different control frequencies. The manager supports per-term getters/setters and returns episodic accumulators for logging.

Curriculum Manager The *CurriculumManager* adapts environment quantities over time (e.g., goal tolerances, mass/friction ranges, initial-state dispersions) through a list of terms (`CurriculumTermCfg`). Terms are evaluated to progressively harden the task as performance improves, stabilizing learning and enabling difficulty schedules without entangling curriculum logic with rewards or resets.

Event Manager The *EventManager* orchestrates operations bound to different simulation modes via `EventTermCfg` terms.

- `prestartup`: applied once before the simulation is instantiated. This mode is used to randomize USD-level properties of the stage (e.g., swapping assets, modifying materials or geometric parameters) before any physics step is taken.
- `startup`: applied once after the simulator has been brought online. Typical uses include initialization that requires a running physics context, such as setting controller gains, seeding random number generators, or configuring run-time properties of the scene.
- `reset`: applied at every environment reset. This mode is used to randomize episode-level conditions (initial joint configurations, object poses, friction coefficients, etc.), so that each rollout starts from a slightly different configuration.
- `interval`: applied periodically at fixed simulation intervals, enabling slower perturbations or periodic interventions when needed.

Typical uses include random pushes, asset swapping, or changes to physical and material parameters. Terms are grouped by mode and applied with optional per-term parameters, so that different sources of variability can be controlled independently.

Command Manager The *CommandManager* generates and updates high-level commands/goals (e.g., target poses) through `CommandTermCfg` terms. It can resample commands in regular intervals, expose commands as part of the observation (when useful), and provides shared descriptors that other managers (primarily Observation and Reward) can reference to build signals consistent with the current objective.

Termination Manager The *TerminationManager* produces the episode-end signal (the `dones` tensor) by taking the logical OR over the registered terms (`TerminationTermCfg`). In line with the Gymnasium API, it separates *truncated* (exogenous limits, e.g., max steps) from *terminated* (MDP terminal states such as success, failure, or safety violations). Each term declares whether it contributes to truncation or termination, and per-term counters are exposed for diagnostics.

Actuators

In Isaac Lab, *actuators* bridge desired joint commands and the simulated articulation. They define the control law that maps commands to joint efforts/targets and intrinsic joint properties like friction, armature and limits. Actuators come in two

families (*implicit* and *explicit*) and can be configured per joint or per joint group within the same articulation.

The **Explicit actuators** compute the control effort in the actuator code (discrete-time PD and variants) and then apply it to the solver. This mirrors many real controllers and gives fine-grained access to the commanded effort (useful for logging, custom saturation, delays, or additional control logic). However, the explicit loop is more sensitive to the integration step and can require careful stability tuning, especially under contact or with aggressive gains.

The **Implicit actuators** were the choosen one for this work. They delegate PD regulation to the PhysX joint drives. Stiffness and damping set the position tracking behavior. Because the PD law is integrated *within* the physics solver, constraint handling (including velocity/effort limits) and contact dynamics are resolved consistently at each substep. This often yields superior numerical stability at larger control time steps and more faithful behavior in contact-rich regimes that are central for this work.

In terms of configuration, the values of `stiffness` and `damping` were chosen to achieve a good trade-off between tracking accuracy and smoothness of motion, as will be discussed in more detail in the dedicated Section 3.3. The `velocity_limit` and `effort_limit` parameters, instead, were set according to the UR10e joint specifications reported in Appendix C. For practical reasons, the final velocity and effort limits used in simulation are summarized in Table 3.1.

Joint	$\dot{q}_{\max}$ **[rad/s]**	$\tau_{\max}$ **[N m]**
shoulder_pan	2.0944	330
shoulder_lift	2.0944	330
elbow	3.1416	150
wrist_1	3.1416	56
wrist_2	3.1416	56
wrist_3	3.1416	56
gripper_finger	2.0	200

Table 3.1: Velocity and effort limits used for the UR10e joints and the gripper actuator in the simulation setup.

3.1.3 Workstation

All experiments were carried out on a dedicated workstation used both for neural network training and for large-scale physics simulation. The machine is equipped with a multi-core CPU, and an NVIDIA GPU, providing enough compute and memory bandwidth to sustain thousands of parallel Isaac Lab environments and

to complete PPO training runs within practical time. This setup was crucial to support the training of the task with 4,096 concurrent environments. The complete technical specifications of the workstation are reported in Appendix A.

Figure 3.2: Workstation used for the training part

3.2 Deployment Pipeline and Tools

The deployment part of this work relies on a combination of software tools and frameworks to enable the transfer of learned policies from simulation to the real UR10e robotic arm. The overall pipeline is shown in Figure 3.3 and is organised into three layers:

1. **First layer:** the policy is tested on the same simulator in which it was trained (Isaac Lab/Isaac Sim), in order to verify that the learned behaviour is consistent and safe within the original training environment.

2. **Second layer:** the policy is tested on URSim, the official simulator provided by Universal Robots. This layer not only provides an additional safety check in a controller-accurate simulation, but also allows us to develop and validate the deployment code that will control the real robot (with the exception of the gripper integration, which is not available in URSim).

3. **Third layer:** the policy is finally deployed and tested on the real UR10e robot.

Both the first and the second layer fall into the Simulation-to-Simulation (Sim2Sim) category: the trained policy is first validated in the same environment in which it was learned, and then exercised on a different simulator to assess whether the resulting motion can be considered safe. The policy is promoted to the next layer of the pipeline only if the results at the current stage are deemed satisfactory.

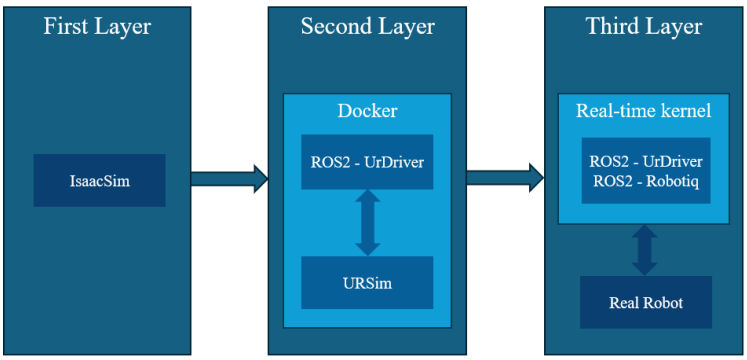


Figure 3.3: Deployment pipeline from Sim2Sim validation in Isaac Sim to URSim and final Sim2Real deployment

The main components of this pipeline are summarised in the following sections, while a more in-depth discussion of the Sim2Sim and Sim2Real validation procedures is provided in Chapter 5.

3.2.1 Universal Robots Simulator (URSim)

Universal Robots provides a software package, called URSim, that reproduces the control environment of a UR robot on a standard PC. URSim runs the same PolyScope interface and the same controller software that are available on the physical e-Series robots, but without requiring the actual manipulator to be connected. In practice, it exposes the same communication endpoints and accepts the same program structures that would be executed on the real controller. For this reason it is a convenient tool to verify that external applications, like ROS 2 nodes or custom drivers, can connect to the UR control stack and exchange data in the expected format before putting the real hardware at risk.

In this work URSim was deployed inside a Docker[1] container and connected to a second container hosting the ROS 2 stack through a dedicated virtual network. This setup mirrors the final deployment in which the ROS 2 nodes run on a separate machine and communicate with the robot over Ethernet. Once the URSim instance is running, the ROS 2 driver can establish an exchange session where, publish the robot state, and forward joint trajectories exactly as it would do with a physical UR10e. In other words, URSim allows us to validate the whole "southbound" part of the control pipeline (network reachability, driver configuration, controller

[1]Docker is a containerization platform that runs applications inside lightweight, isolated containers for reproducible deployment

selection, topic names) without requiring access to the laboratory robot.

It is important to note, however, that URSim emulates the *controller*, not the full dynamics of the robot: there is no rigid-body simulation, no realistic collision model, and no physical gripper attached. Motions that in URSim appear to run smoothly may still trigger protective stops on the real arm if they violate safety settings or if the real payload and friction differ from the nominal ones. Likewise, additional devices such as the Robotiq 2F-140 must be integrated separately on the ROS 2 side, because the simulator does not provide a simulated end-effector. Despite these limitations, URSim remains an essential intermediate step in the validation pipeline adopted in this thesis: it guarantees that the software components can talk to the official UR control stack, and that switching from the simulated controller to the physical one is reduced to a change of IP address and safety parameters.

3.2.2 Robot Operating System 2

The deployment pipeline developed in this work relies on Robot Operating System 2 (ROS 2), the second generation of the widely used open-source middleware for robotics. ROS 2 was introduced to overcome several structural limitations of ROS 1, such as the single-master communication model and the limited support for distributed and heterogeneous deployments [36]. In addition, ROS 2 has become the de facto standard for new developments in the ROS ecosystem, and most actively maintained drivers and tools now target ROS 2 first.

In this thesis, ROS 2 is used as a common abstraction layer between three classes of components: (i) the Universal Robots controller (simulated or real), accessed through the ROS 2 UR driver and exposing the robot state and trajectory interfaces; (ii) the control and integration nodes, which run the ROS 2 driver, the joint trajectory controller and the auxiliary interfaces (e.g., for the gripper); and (iii) the learning component, which publishes joint targets or short trajectories generated by the policy. The choice of ROS 2 is therefore mainly driven by practical deployment aspects: it simplifies communication across multiple containers and machines, and it provides a stable and well-supported interface to both URSim and the physical UR10e controller.

Alternative solutions based on direct communication with the UR controller via its Real-Time Data Exchange (RTDE) interface were also considered. A pure RTDE-based architecture would have been feasible and is commonly adopted in industrial applications. However, in the context of this thesis it would have required developing and maintaining a dedicated communication layer for streaming robot state, sending commands, handling errors and logging data, as well as separate integration code for additional devices such as the gripper. By contrast, the ROS 2 driver for Universal Robots and the ROS 2 adapter for the Robotiq gripper already expose these capabilities through standard topics and services, so that both the

arm and the end-effector can be treated as ROS entities and their data can be recorded or replayed with standard tooling [37, 38]. For these reasons, ROS 2 was adopted as the main integration layer, while RTDE is used indirectly through the UR ROS 2 driver.

3.2.3 UR10e robotic arm

The robotic arm used in this work is the Universal Robots UR10e, a 6-DOF collaborative manipulator belonging to the e-Series family. It offers a payload of 10 kg, a reach of approximately 1300 mm and a repeatability of ± 0.05 mm, which makes it suitable both for industrial manipulation and for research scenarios that require accurate positioning over a relatively large workspace [39]. Being a collaborative robot, the UR10e integrates force/torque sensing at the tool and a set of safety functions (speed and separation monitoring, reduced mode, configurable safety planes) that can limit its motion when operating close to humans or when external control inputs are not fully trusted.

From the perspective of this thesis, the UR10e is relevant for two main reasons. First, it is natively supported by the official UR ROS 2 driver, so that joint states, trajectories and IO can be accessed through standard ROS topics without writing a vendor-specific client. Second, the kinematic structure of the UR10e is well documented and widely used in robotics literature, which simplifies the derivation of Denavit-Hartenberg (DH) parameters, the definition of tool frames consistent with simulation, and the comparison between simulated and real joint configurations. The same USD model of the UR10e was imported in Isaac Sim, so that the articulated structure in simulation matches the physical arm, reducing discrepancies at deployment time. For further details on the UR10e, please refer to the Appendix C.

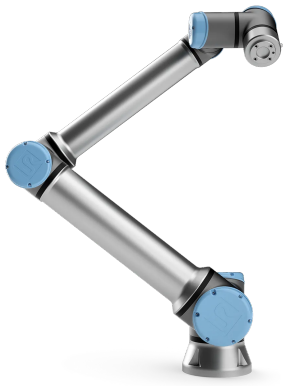


Figure 3.4: UR10e robotic arm

3.2.4 Robotiq 2F-140 gripper

The end-effector mounted on the UR10e is a Robotiq 2F-140, a two-finger adaptive gripper with a maximum stroke of 140 mm, intended for general-purpose grasping of objects with different sizes [40]. The gripper is position-controlled and is designed to be integrated with UR robots through URCaps, so that open/close actions can be triggered directly from the robot controller. In the experimental setup used in this thesis, the gripper is commanded from ROS 2 through the dedicated adapter discussed in Section 5.2. This keeps the arm and the end-effector on the same communication layer (ROS 2) and allows the learned policy to trigger open/close commands in synchrony with the generated joint trajectories.

The 2F-140 is particularly suitable for the reach-and-grasp and lift tasks considered in this thesis, since its large stroke makes it possible to grasp a large variaty of objects without changing fingers or tooling, while keeping the control interface simple (open/close or target position). In addition, its documentation provides precise dimensions and mounting interfaces, which are replicated in the USD asset used in Isaac Sim. Further details on the Robotiq 2F-140 are reported in Appendix D.

Figure 3.5: Robotiq 2F-140 gripper mounted on UR10e

3.3 Gain tuning

To allow the robot to effectively learn the desired manipulation tasks we splitted the training process into multiple stages, each focusing on a specific sub-task. This progressive approach allow to define custom and specific reward functions for each stage, facilitating the learning process.

As previously mentioned, our robotic arm consists of six actuated joints, also referred to as its degrees of freedom (DOF). Each of them are actuated through an electric motor which introduce nonlinear characteristics into the system, including

actuation delays, torque saturation, and velocity limits, which shape the robot's overall dynamic response.

In simulation, joint actuation is defined as *position-controlled*, for which the physics engine internally implements PD controller that computes the torques applied on the actuated joints.

In particular, each joint in Isaac Lab is associated with an `ImplicitActuatorCfg` object, which defines its drive behavior by specifying the stiffness and damping coefficients of the implicit PD controller. This controller computes the actuation torque as

$$\tau = K_p(q - q_{\text{target}}) + K_d(\dot{q} - \dot{q}_{\text{target}}), \tag{3.1}$$

where K_p represents the *stiffness* gain and K_d the *damping* gain and generally $\dot{q}_{\text{target}} = 0$.
The stiffness determines how aggressively the joint reacts to positional errors, while the damping smooths oscillations and adds numerical stability to the system. The implicit actuator model is provided by the underlying physics engine (PhysX), which adds an additional layer of numerical damping to ensure stable integration at high stiffness values.

A correct tuning of these gains is essential, since they directly influence the robot's simulated dynamics. If the joint drives are poorly tuned, the robot may behave unrealistically or even become unstable, making reinforcement-learning training ineffective.

3.3.1 Manual tuning procedure

We manually tuned the joint gains using Isaac Sim's Gain Tuner, applying sinusoidal position references and observing the resulting motion. The goal was to identify, for each joint, a pair of stiffness and damping coefficients that provided a fast and stable response consistent with the UR10e's physical dynamics.

We decide to create a position drive, in other terms given Eq. (3.1) we set $K_p > 0$ and K_d can be any value. The joint drives were configured in **position mode** with **type = force**. In this configuration, each drive interprets the command as a desired joint position and applies the corresponding torque according to Eq. (3.1). Using the *force* type means that the computed effort is directly applied as torque, resulting in a more realistic reproduction of the UR10e's physical actuation compared to the acceleration-normalized alternative. After selecting the appropriate configuration for each joint drive, the tuning procedure was carried out in two main stages. First, the stiffness and damping parameters were tuned individually for each joint, testing one degree of freedom at a time. This allowed precise observation of the local dynamic response and facilitated isolating the effect of each parameter on the corresponding actuator. Once satisfactory values were obtained for all joints, a global validation test was performed in which **all joints were actuated**

simultaneously, rather than in sequential order. This second stage ensured that the chosen parameters also provided stable and coherent behavior under coupled multi-joint motion, confirming the robustness of the tuning for full-arm operation as shown in figure3.6.

To ensure feasibility for the real robot, the sinusoidal references were chosen so that the needed joint velocities did not exceed the UR10e's maximum joint velocity limits:

- For *base* and *shoulder* joints with $\dot{q}_{\max} = 120°/\text{s}$: period $T = 19\,\text{s}$.
- For *lift*, *wrist 1/2/3* joints with $\dot{q}_{\max} = 180°/\text{s}$: period $T = 13\,\text{s}$.

More generally, for a sinusoidal reference $q(t) = A\sin(\omega t)$, a conservative feasibility condition that avoids velocity saturation is

$$T \geq \frac{2\pi A}{\dot{q}_{\max}},$$

where A in this case is 360°. This ensures that the maximum velocity $A\omega$ remains within the joint's capabilities.

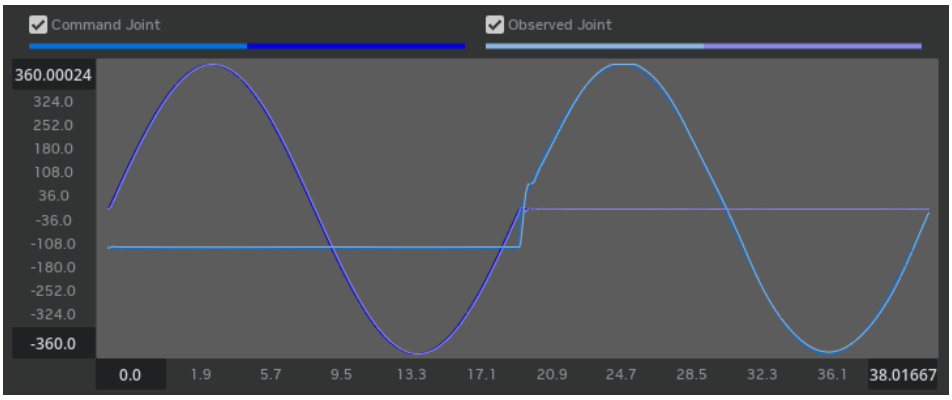
3.3.2 Final gain values and unit conventions

The table below distinguishes the gains set in the robot USD (expressed in degrees) from the gains used in the `ImplicitActuatorCfg` (expressed in radians). The conversion formula are available in the Appendix B

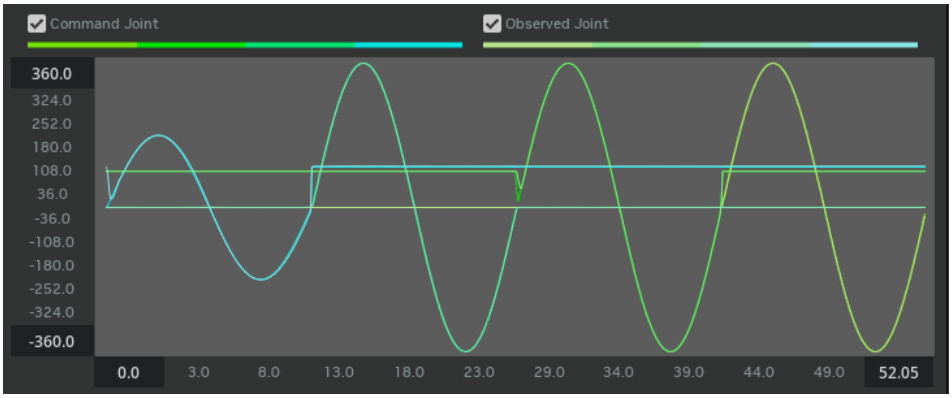
Joint	K_p^{USD}	K_d^{USD}	K_p^{At}	K_d^{At}
shoulder_pan	17.44	1.39	1000	80
shoulder_lift	15.70	1.13	900	65
elbow	13.96	0.78	800	45
wrist_1	12.21	0.61	700	35
wrist_2	11.34	0.52	650	30
wrist_3	10.47	0.44	600	25
gripper	5.23	0.17	300	10

Table 3.2: Joint-drive gains: values set in USD vs `ImplicitActuatorCfg`

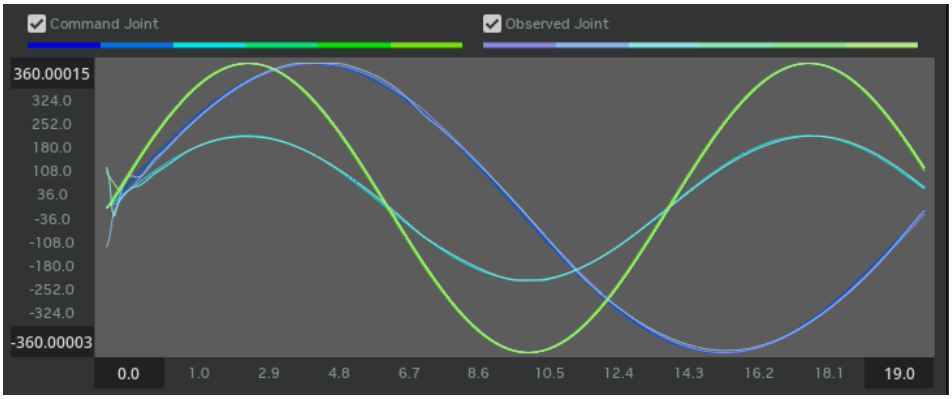
The resulting joint trajectories are shown in Fig. 3.6: for all joints the measured position closely follows the reference with limited overshoot and no visible oscillations, indicating that the chosen stiffness-damping pairs provide a good trade-off between tracking accuracy and smoothness.



(a) Shoulder joints, period 19 s



(b) Elbow and wrist joints, period 13 s



(c) All joints together, period 19 s

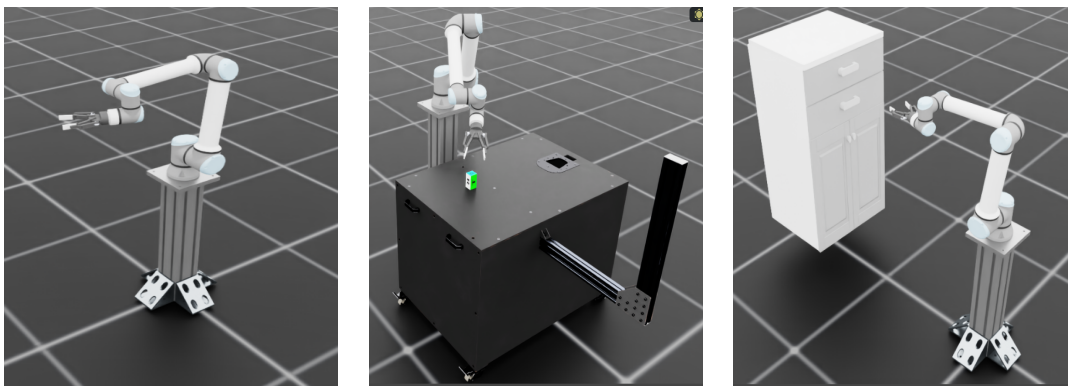
Figure 3.6: Sinusoidal response of the tuned UR10e

Chapter 4

Training in simulation

In this chapter we describe how the three manipulation policies (*Reach*, *Lift*, and *OpenDrawer*) are trained in NVIDIA Isaac Lab, starting from the environments provided in the Isaac Lab library [41] and then adapting them to our specific setup. Our goal is to highlight which elements of the training pipeline are shared across tasks and which ones must be specialised as the manipulation problem becomes more constrained. The *Reach* task is first used to define the core components of the training environment (scene, action and observation spaces, command structure, and reward design), which then serve as a template for the subsequent tasks.

Throughout the chapter we follow a common structure: we first present the *agent configuration*, i.e. the PPO-based learning setup shared by all experiments, and then we detail, in separate sections, the environment-specific choices for each task (scene, observations, rewards, terminations, and any curriculum terms).



(a) `Reach` Environment (b) `Lift` Environment (c) `OpenDrawer` Environment

Figure 4.1: Initial configuration of the three simulated environments used in this work

4.1 Agent configuration

All tasks are optimized with the same on-policy algorithm, Proximal Policy Optimization (PPO), using the implementation shipped with `rsl-rl`[42]. Training is run with a large number of parallel environments (4096) so that each policy update can exploit a sizeable batch of fresh on-policy data, which is particularly convenient in Isaac Lab where environment stepping is GPU-accelerated. Across all tasks we use the same discount factor and advantage estimation parameters:

$$\gamma = 0.99, \qquad \lambda = 0.95,$$

and we adopt the clipped surrogate objective with ratio clip $\epsilon = 0.2$. Actor and critic are both implemented as feed-forward multilayer perceptrons with ELU activations as shown in Figure 4.2. In the figure, $n_1 = \dim(o)$, where o denotes the observation (or state) vector, while n_2, n_3 and n_4 represent the number of neurons in each hidden layer of the MLP, whose values differ depending on the specific task(see Table 4.1), whereas the number of layers is kept fixed across all tasks.

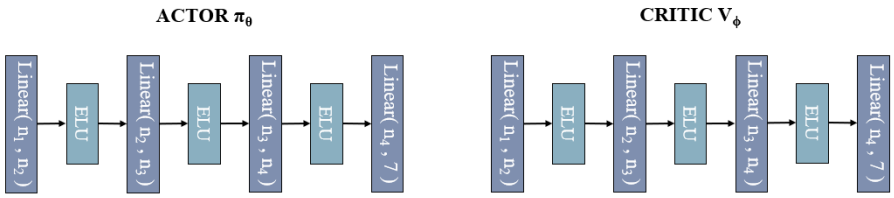


Figure 4.2: Actor and critic architecture used for all the experiments

The Table 4.1 reports, in the first half, the settings that are shared and the task-specific, in the second half.

In practice, the **reach** task can afford a higher learning rate and a smaller network because the observation vector only contains robot state and the generated target pose. The **lift** task increases the rollout horizon and the network capacity to encode object-related features and to let PPO see complete grasping episodes in each batch. The **open-drawer** task reuses the lift configuration, and augments only the environment-side definition adding observations and multi-stage rewards.

4.2 Environment configuration

In this section we describe the environment-side configuration for each task, focusing on the parts that do change from one task to another. What changes from task to

	Reach	**Lift**	**OpenDrawer**
Algorithm	PPO (rsl-rl)	PPO (rsl-rl)	PPO (rsl-rl)
Num. environments	4096	4096	4096
Discount γ	0.99	0.99	0.99
GAE λ	0.95	0.95	0.95
Clip parameter ϵ	0.2	0.2	0.2
Entropy coefficient	0.005	0.005	0.005
Max iterations	1500	1500	1500
Steps per environment	32	64	96
Actor-critic MLP	[64, 64, 32]	[256, 128, 64]	[256, 128, 64]
Learning rate	1×10^{-3}	1×10^{-4}	1×10^{-4}
Policy epochs / update	5	16	16
Mini-batches / update	4	64	64

Table 4.1: PPO agent settings common to all tasks and task-specific deviations.

task is therefore not *how* we command the robot, but *what* we ask it to track (the command), *what* we show to the policy (the observations), and *how* we pay it (the reward). The action interface in fact kept **the same across all tasks**, so that policies can be trained and deployed with a single, fixed control head.

Common action interface

All environments expose to the agent a fixed, low-dimensional action space composed of:

- a *joint-position* action on the six UR10e joints, numbered from the base (1) to the last wrist joint (6),

  $$\mathbf{a}_t^{\text{arm}} = (a_t^1, a_t^2, a_t^3, a_t^4, a_t^5, a_t^6),$$

  which is first scaled and then added to a task-specific default posture $\mathbf{q}_0^{\text{arm}} \in \mathbb{R}^6$. Given a fixed scale factor $s > 0$ (e.g., $s = 0.5$), the joint commands sent to the controller are:

  $$\mathbf{q}_t^{\text{arm}} \;=\; \mathbf{q}_0^{\text{arm}} + s\,\mathbf{a}_t^{\text{arm}}; \tag{4.1}$$

- a scalar *binary joint-position* action for the Robotiq finger joint,

  $$a_t^{\text{gr}},$$

which is mapped to a discrete open/close command. In particular, the commanded finger position q_t^{gr} is obtained as:

$$q_t^{\mathrm{gr}} = \begin{cases} 0.7 & \text{if } a_t^{\mathrm{gr}} < 0 \quad \text{(closed gripper)}, \\ 0.0 & \text{if } a_t^{\mathrm{gr}} \geq 0 \quad \text{(open gripper)}. \end{cases} \tag{4.2}$$

Keeping this interface fixed has the advantage that the same seven-dimensional action vector $\mathbf{a}_t = (\mathbf{a}_t^{\mathrm{arm}}, a_t^{\mathrm{gr}}) \in \mathbb{R}^7$ and the corresponding command vector $\mathbf{q}_t = (\mathbf{q}_t^{\mathrm{arm}}, q_t^{\mathrm{gr}}) \in \mathbb{R}^7$ are used consistently throughout training and deployment for all three tasks.

Reference frames and notation

Unless otherwise stated, all positions and orientations are expressed in the world frame. In particular, we denote by $\mathbf{p}_t \in \mathbb{R}^3$ the Cartesian position of a point at time t and by $\mathfrak{q}_t \in \mathbb{R}^4$ its orientation, represented as a unit quaternion in scalar-first form. Whenever a quantity is naturally defined in a different frame (for instance, a command expressed in the robot base frame), it is internally transformed to the world frame before computing distances or tracking errors. As a consequence, all rewards that involve Euclidean distances or pose errors are implicitly assumed to operate on quantities represented in a common reference frame.

4.2.1 Reach

The `Reach` environment is the minimal setup used to train a manipulation policy. At the beginning of each episode, a target end-effector pose is sampled in a small region of the robot workspace, and the policy must drive the arm so that the current Tool Center Point (TCP) pose matches the commanded one. In other words, the robot has to:

1. move the TCP to the desired Cartesian position;
2. align the TCP orientation with the target pose.

This task can be used to put the robot in a good initial configuration for the subsequent, more complex tasks (for example, training the robot to reach a specific pose near the object to be grasped can be used as a starting point for the `Lift` task).

Scene and robot instantiation. In the `Reach` task only the UR10e manipulator is instantiated as an active element in the scene, with no additional movable

objects. At the beginning of each episode, the robot joints are reset to the nominal configuration

$$\mathbf{q}_0 = (\mathbf{q}_0^{\text{arm}}, q_0^{\text{gr}}) = (0.0,\ -1.712,\ 1.712,\ 0.0,\ \tfrac{\pi}{2},\ 0.0,\ 0.0)\ rad,$$

with small random perturbations around this pose, so that the policy does not overfit to a single exact joint configuration. The robot is mounted on a fixed aluminium pedestal, which raises the base to a convenient working height and reproduces the mounting configuration used in the real setup. Finally, the TCP is provided with a dedicated reference frame, as illustrated in Figure 4.3. We denote the pose of this point at time t by

$$\mathbf{r}_t^{TCP} = (\mathbf{p}_t^{\text{TCP}}, \mathfrak{q}_t^{\text{TCP}})$$

where $\mathbf{p}_t^{\text{TCP}} \in \mathbb{R}^3$ and $\mathfrak{q}_t^{\text{TCP}} \in \mathbb{R}^4$.

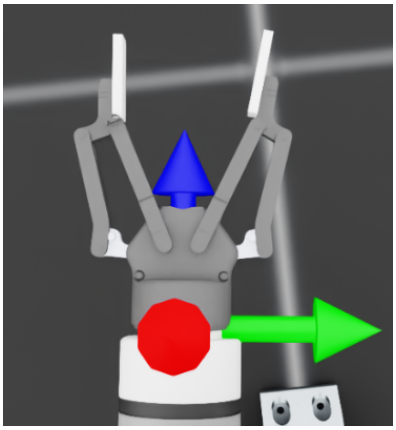


Figure 4.3: Reference frame associated with the Tool Center Point of the robot

Command. The command represents the target pose generated by the environment, i.e., the pose that the policy is required to track. Since each episode lasts 8 s and the command is resampled every 4 s, the policy must be able to reach *two* different targets within a single episode. This also means that, from the policy's point of view, the target is *non-stationary* within the episode: a controller that only works for a single fixed goal would not solve this environment.

The command at time t is a 7-dimensional pose

$$\mathbf{c}_t = (\mathbf{p}_t^{\text{cmd}}, \mathfrak{q}_t^{\text{cmd}}),$$

where $\mathbf{p}_t^{\text{cmd}} \in \mathbb{R}^3$ denotes the Cartesian target position expressed in metres, and

$\mathfrak{q}_t^{\text{cmd}} \in \mathbb{R}^4$ is the unit quaternion[1] encoding the desired end-effector orientation in the world frame.

At every resampling, the target position is drawn uniformly from

$$x \in [0.55, 0.85]\ m, \quad y \in [-0.2, 0.2]\ m, \quad z \in [0.35, 0.7]\ m,$$

while the orientation is sampled by fixing

$$\text{roll} = 0\ rad, \qquad \text{pitch} = \tfrac{\pi}{2}\ rad, \qquad \text{yaw} \in [-\pi, \pi]\ rad.$$

Observations. The policy observation is the concatenation of:

- $\mathbf{q}_t - \mathbf{q}_0$: arm and gripper joint positions expressed as offsets from the initial configuration;
- $\dot{\mathbf{q}}_t^{\text{arm}} - \dot{\mathbf{q}}_0^{\text{arm}}$: arm joint velocities expressed as offsets from the initial rest state, with $\dot{\mathbf{q}}_0^{\text{arm}} = \mathbf{0}$;
- $\mathbf{c}_t$: the current command pose;
- $\mathbf{a}_{t-1}$: the action applied at the previous step.

The overall, noise-free observation vector at time t is therefore defined as

$$\hat{\mathbf{o}}_t = \left((\mathbf{q}_t - \mathbf{q}_0)^\top,\ (\dot{\mathbf{q}}_t^{\text{arm}} - \dot{\mathbf{q}}_0^{\text{arm}})^\top,\ \mathbf{c}_t^\top,\ \mathbf{a}_{t-1}^\top\right)^\top \in \mathbb{R}^{27}.$$

To improve robustness, the actual observation fed to the policy is obtained by adding elementwise uniform noise to joint positions and velocities,

$$\mathbf{o}_t = \hat{\mathbf{o}}_t + \boldsymbol{\eta}_t,$$

where the components of $\boldsymbol{\eta}_t$ corresponding to joint positions and velocities are independently sampled from the uniform distribution $\mathcal{U}[-0.01, 0.01]$, and set to zero for all other entries. The gripper velocity is excluded from the observation because it is not directly available on the real robot.

Rewards. The reach task combines five terms; each term returns a non-negative quantity, and the final sign is given by its weight in the configuration:

$$r_t = w_{\text{pos}} r_t^{\text{pos}} + w_{\text{pos-fine}} r_t^{\text{pos-fine}} + w_{\text{ori}} r_t^{\text{ori}} + w_{\text{act}} r_t^{\text{act}} + w_{\Delta\text{act}} r_t^{\Delta\text{act}} \tag{4.3}$$

[1] In Isaac Lab quaternions are represented in scalar-first form, $\mathfrak{q}_t = (w_t, x_t, y_t, z_t)$.

(1) Coarse position tracking. This term penalises the Euclidean distance between the current TCP position and the commanded target position. The coarse position error is:

$$r_t^{\text{pos}} = \left\| \mathbf{p}_t^{\text{TCP}} - \mathbf{p}_t^{\text{cmd}} \right\|_2 \tag{4.4}$$

(2) Fine position tracking. The same distance is passed through a bounded kernel to provide a stronger reward signal when the TCP is already close to the target:

$$d_t = \left\| \mathbf{p}_t^{\text{TCP}} - \mathbf{p}_t^{\text{cmd}} \right\|_2 \tag{4.5}$$

$$r_t^{\text{pos-fine}} = 1 - \tanh\left(\frac{d_t}{\sigma_{\text{pos}}} \right) \tag{4.6}$$

with $\sigma_{\text{pos}} = 0.1$ m in the experiments. For small distances $d_t \ll \sigma_{\text{pos}}$ the term approaches 1, while it smoothly decays towards 0 as d_t increases. For more details about the $tanh(x)$ function, please refer to Appendix E.

(3) Orientation tracking. The orientation error is computed from the quaternion that maps the current TCP orientation to the commanded one. The quaternion error is:

$$\mathfrak{q}_t^{\text{err}} = \mathfrak{q}_t^{\text{cmd}} \otimes \left(\mathfrak{q}_t^{\text{TCP}} \right)^{-1} \tag{4.7}$$

and the corresponding angular error is obtained from its scalar part:

$$r_t^{\text{ori}} = 2 \arccos\Big(|w_t| \Big) \tag{4.8}$$

where w_t denotes the scalar component of the unit quaternion $\mathfrak{q}_t^{\text{err}} = (w_t, x_t, y_t, z_t)$.

(4) Action magnitude. This term penalises large actions by applying an L2-squared penalty to the full action vector, including both arm and gripper components:

$$r_t^{\text{act}} = \left\| \mathbf{a}_t \right\|_2^2 \tag{4.9}$$

(5) Action rate. This term penalises rapid changes in the action by applying an L2-squared penalty to the difference between consecutive actions:

$$r_t^{\Delta\text{act}} = \left\| \mathbf{a}_t - \mathbf{a}_{t-1} \right\|_2^2 \tag{4.10}$$

Each of the previously defined reward terms is then multiplied, at each step t, by its corresponding weight as shown in Equation (4.3). All weight values are reported in Table 4.2.

Term	Eq.	Weight	Type
Coarse position tracking	r_t^{pos}	-0.4	Penalty
Fine position tracking	$r_t^{\text{pos-fine}}$	0.6	Reward
Orientation tracking	r_t^{ori}	-0.4	Penalty
Action magnitude	r_t^{act}	-5×10^{-3}	Penalty
Action rate	$r_t^{\Delta\text{act}}$	-5×10^{-3}	Penalty

Table 4.2: Reward weights for the `Reach` task

Events. At every episode `reset` the environment applies the following randomisations:

- the robot joints are reset around the nominal configuration, by adding independent offsets uniformly sampled in the range $[-0.125, 0.125]$ to each joint and setting all joint velocities to zero;
- the actuator gains of the robot are randomly scaled, with stiffness uniformly sampled in $[0.9, 1.1]$ and damping in $[0.75, 1.5]$.

These perturbations help prevent the policy from overfitting to a single initial pose and a single set of actuator gains.

Terminations. The environment uses only a time-out condition, in other words, the episode is terminated after 8 s.

4.2.2 Lift

The `Lift` environment extends the previous `Reach` task by adding a table and a graspable object on it. The policy has to execute, in a short episode (2.5 s), a typical manipulation micro-sequence:

1. reach the object with the end-effector,
2. close the gripper while staying close to the object,
3. lift the object above the table,
4. move the object towards a commanded goal pose.

The scene is randomized at reset so that the learned policy does not overfit a single configuration.

Scene and robot instantiation. The object used in this environment is a rigid cube taken from the Isaac Sim asset library, spawned at

$$\mathbf{p}_0^{\text{obj}} = (0.5,\, 1.0,\, -0.095)\ m, \qquad \mathfrak{q}_0^{\text{obj}} = (0.7071068,\, 0,\, 0,\, -0.7071068)\ rad,$$

on top of a table. This pose is then slightly perturbed at reset time by a dedicated `Event` to increase variability. The table itself is instantiated as a static asset positioned at

$$\mathbf{p}_t^{\text{tab}} = (0.3,\, 1.0,\, -0.15)\ m, \qquad \mathfrak{q}_t^{\text{tab}} = (0.707,\, 0,\, 0,\, 0.707)\ rad,$$

with a scaled reduced height of a factor 0.9. It acts as a fixed support surface for the cube and as a collision object for the robot, without introducing any additional dynamics or control inputs.

At the beginning of each episode the UR10e joints are initialised around the nominal configuration

$$\mathbf{q}_0^{\text{arm}} = (1.0,\, -1.0,\, 1.4,\, 0.0,\, \tfrac{\pi}{2},\, 0.0)\ rad,$$

corresponding to the six arm joints, with small random perturbations applied by the `Event` to improve robustness.

Finally, the end-effector is provided with a dedicated reference frame located between the two fingertips of the gripper pads, as illustrated in Figure 4.4. We denote the pose of this point at time t by

$$\mathbf{r}_t^{\text{gr}} = \left(\mathbf{p}_t^{\text{gr}},\, \mathfrak{q}_t^{\text{gr}}\right),$$

where $\mathbf{p}_t^{\text{gr}} \in \mathbb{R}^3$ and $\mathfrak{q}_t^{\text{gr}} \in \mathbb{R}^4$. In the implementation, this frame is realised in Isaac Lab as a `FrameTransformer` attached to the TCP link, with a fixed translational offset of 0.2068 m along the local its z-axis.

Command. Each episode a single command is sampled. It represents the *desired object position* and, as the previous case, it is expressed in the robot frame. Only the position, while not the orientation, is randomized in the intervals:

$$x \in [0.4, 0.6]\ m, \quad y \in [0.7, 0.9]\ m, \quad z \in [0.0, 0.25]\ m,$$

and obviously, with

$$\text{roll} = 0\ rad, \qquad \text{pitch} = 0\ rad, \qquad \text{yaw} = 0\ rad.$$

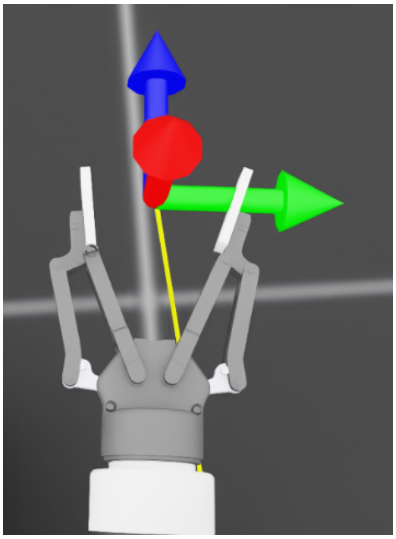


Figure 4.4: Reference frame associated with the end-effector of the robot

Observations. The observation vector reuses the same four components as in the `Reach` task (see Section 4.2.1), but, it also **includes the object position** $\mathbf{p}_t^{\text{obj}}$ expressed in the robot base frame. The overall, noise-free observation vector at time t is therefore defined as:

$$\hat{\mathbf{o}}_t = \Big((\mathbf{q}_t - \mathbf{q}_0)^\top,\ (\dot{\mathbf{q}}_t^{\text{arm}} - \dot{\mathbf{q}}_0^{\text{arm}})^\top,\ \mathbf{p}_t^{\text{obj}\top}, \mathbf{c}_t^\top,\ \mathbf{a}_{t-1}^\top \Big)^\top \in \mathbb{R}^{30}.$$

To improve robustness, the actual observation fed to the policy is obtained by adding elementwise uniform noise to selected components:

$$\mathbf{o}_t = \hat{\mathbf{o}}_t + \boldsymbol{\eta}_t,$$

where the components of $\boldsymbol{\eta}_t$ corresponding to joint positions and arm joint velocities are independently sampled from $\mathcal{U}[-0.01, 0.01]$, those corresponding to the object position $\mathbf{p}_t^{\text{obj}}$ are sampled from $\mathcal{U}[-0.05, 0.05]$, and all other entries are set to zero. Also in this case the gripper velocity is excluded from the observation because it is not directly available on the real robot.

Reward. The total reward is defined as:

$$\begin{aligned} r_t = {} & w_{\text{reach}} r_t^{\text{reach}} + w_{\text{grasp}} r_t^{\text{grasp}} + w_{\text{lift}} r_t^{\text{lift}} \\ & + w_{\text{goal}} r_t^{\text{goal}} + w_{\text{goal-fine}} r_t^{\text{goal-fine}} \\ & + w_{\Delta\text{act}} r_t^{\Delta\text{act}} + w_{\text{act}} r_t^{\text{act}} + w_{\text{joint}} r_t^{\text{joint}} \end{aligned} \tag{4.11}$$

Each term corresponds to a specific phase of the task (reach, grasp, lift, move-to-goal) or to a regularization component (action and joint penalties). All the weights are reported in the Table 4.3

(1) EE-object reaching. This term encourages bringing the end-effector frame, on top of the cube frame. Let $\mathbf{p}_t^{\text{gr}} \in \mathbb{R}^3$ denote the position of the end-effector frame and $\mathbf{p}_t^{\text{obj}} \in \mathbb{R}^3$ the position of the cube frame. The instantaneous distance between the two is

$$d_t^{\text{ee-obj}} = \left\| \mathbf{p}_t^{\text{gr}} - \mathbf{p}_t^{\text{obj}} \right\|_2, \tag{4.12}$$

and the corresponding reward term is defined as the bounded kernel

$$r_t^{\text{reach}} = 1 - \tanh\left(\frac{d_t^{\text{ee-obj}}}{\sigma_{\text{ee}}} \right), \qquad \sigma_{\text{ee}} = 0.05 \text{ m}. \tag{4.13}$$

As in the reach task, this kernel takes values close to 1 only when the end-effector is very close to the object (within a few centimetres), and smoothly decays towards 0 as the distance increases, thus providing a dense but bounded signal throughout the approach phase.

(2) Grasp proximity and closure. This term encourages the agent to close the gripper only when the end-effector is already close to the object. Let r_t^{reach} denote the proximity component already defined in Equation (4.13), let q_t^{gr} be the gripper joint position, and let $q_{\text{gr}}^{\max} = \pi/6 \; rad$ be the reference closed value used for normalization. We define the closure factor as:

$$r_t^{\text{close}} = \min\left(1,\ \max\left(0,\ \frac{q_t^{\text{gr}}}{q_{\text{gr}}^{\max}}\right)\right), \tag{4.14}$$

which increases from 0 (fully open gripper) to 1 (closed around the reference value). The overall grasping reward is then given by

$$r_t^{\text{grasp}} = r_t^{\text{reach}} \cdot r_t^{\text{close}}. \tag{4.15}$$

In this way the term becomes significant only when the end-effector is close to the object *and* the gripper is closing, discouraging premature closures far from the target.

(3) Object lifted. This term provides a sparse binary reward as soon as the cube is lifted above a given height. Let $\mathbf{p}_t^{\text{obj}} = (x_t^{\text{obj}}, y_t^{\text{obj}}, z_t^{\text{obj}})$ denote the object position, and let $h_{\min} = -0.05 \; m$ be the minimal height threshold. The lifting reward is defined as

$$r_t^{\text{lift}} = \begin{cases} 1, & \text{if } z_t^{\text{obj}} > h_{\min}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.16}$$

Once the object has been lifted above $h_{\min}$, the agent keeps receiving this unit reward at every step as long as the condition remains satisfied.

(4) Coarse object-to-goal tracking. Once the object has been lifted, the policy is encouraged to move it towards the commanded goal position. Let r_t^{lift} denote the binary lifting reward already defined in Equation (4.16), and let $\mathbf{p}_t^{\text{goal}} \in \mathbb{R}^3$ and $\mathbf{p}_t^{\text{obj}} \in \mathbb{R}^3$ be the goal and object positions, respectively. The distance between the object and the goal is

$$d_t^{\text{goal}} = \left\| \mathbf{p}_t^{\text{goal}} - \mathbf{p}_t^{\text{obj}} \right\|_2. \tag{4.17}$$

Let $\sigma_{\text{goal}} = 0.3$ m be the kernel scale. The reward term is then defined as:

$$r_t^{\text{goal}} = r_t^{\text{lift}} \left(1 - \tanh\left(\frac{d_t^{\text{goal}}}{\sigma_{\text{goal}}} \right) \right). \tag{4.18}$$

Since r_t^{lift} is zero when the object is still below the lifting threshold, this contribution becomes active only once the cube has been lifted, and it is therefore associated exclusively with the goal-reaching phase.

(5) Fine Object-to-goal tracking. A second object-to-goal term uses the same distance d_t^{goal} defined in Equation (4.18), but with a tighter scale to refine the final placement of the cube once it is close to the target. Let $\sigma_{\text{goal-fine}} = 0.05$ m be the fine kernel scale. The reward is defined as:

$$r_t^{\text{goal-fine}} = r_t^{\text{lift}} \left(1 - \tanh\left(\frac{d_t^{\text{goal}}}{\sigma_{\text{goal-fine}}} \right) \right), \tag{4.19}$$

so that it only becomes active when the object has already been lifted and provides a sharper shaping signal around the desired goal position.

(6) Regularization terms. To keep the commanded motion smooth and to avoid overly aggressive control, three quadratic penalty terms are added to the reward:

$$r_t^{\Delta\text{act}} = \left\| \mathbf{a}_t - \mathbf{a}_{t-1} \right\|_2^2, \tag{4.20}$$

$$r_t^{\text{act}} = \left\| \mathbf{a}_t \right\|_2^2, \tag{4.21}$$

$$r_t^{\text{joint}} = \left\| \dot{\mathbf{q}}_t^{\text{arm}} \right\|_2^2. \tag{4.22}$$

The first two terms, $r_t^{\Delta\text{act}}$ and r_t^{act}, are identical to those used in the `Reach` task (see Section 4.2.1) and penalise, respectively, rapid changes in the action and large

action magnitudes. The additional term r_t^{joint} penalises large arm joint velocities, encouraging smoother trajectories of the manipulator.

Each of the previously defined reward terms is then multiplied, at each step t, by its corresponding weight as shown in Equation (4.11). All weight values are reported in Table 4.3

Term	Eq.	Weight	Type
EE-object reaching	r_t^{reach}	2.0	Reward
Grasp proximity and closure	r_t^{grasp}	15.0	Reward
Object lifted	r_t^{lift}	10.0	Reward
Coarse object-to-goal tracking	r_t^{goal}	50.0	Reward
Fine object-to-goal tracking	$r_t^{\text{goal-fine}}$	70.0	Reward
Action rate	$r_t^{\Delta\text{act}}$	-5×10^{-3}	Penalty
Action magnitude	r_t^{act}	-5×10^{-3}	Penalty
Joint velocities	r_t^{joint}	-1×10^{-4}	Penalty

Table 4.3: Reward weights for the `Lift` task.

Events. At every episode `reset` the environment applies a fixed sequence of active events:

- the whole scene is restored to its default state;
- the object root pose is resampled on the table, with $x, y \in [-0.1, 0.1]$ and $z = 0$;
- the robot joints are reset around the nominal configuration, with joint positions scaled in $(0.75, 1.25)$ and zero joint velocity;
- the actuator gains of the robot are randomly scaled, with stiffness sampled in $[0.9, 1.1]$ and damping in $[0.75, 1.5]$.

In addition, at `prestartup` the object scale is randomized uniformly with $x, y \in [0.8, 1.2]$ and $z \in [1.0, 2.0]$. Furthermore at `startup` the object mass is randomized in $[0.200, 2.0]$ kg with inertia recomputation. These geometric and inertial perturbations, together with the reset-time randomizations, make the policy less sensitive to a single initial pose and to a single object inertia.

Curriculum. To avoid constraining exploration too early, the penalty weights are initially set to the values reported in Table 4.3 and then, after N_{des} environment steps, they are made stricter through two curriculum terms.

Let k_{des} denote the PPO iteration at which we desire to switch the weights, and let H be the number of steps per environment collected at each iteration (see Table 4.1). The corresponding number of environment steps N_{des} can be approximated as

$$N_{\text{des}} \approx H \times k_{\text{des}}. \tag{4.23}$$

In our experiments we chose $k_{\text{des}} \approx 470$, and the resulting curriculum schedule is summarised in Table 4.4. In this way the agent can first discover a successful grasp-and-lift sequence under relatively mild regularization, and only afterwards is it encouraged to produce smoother actions and joint trajectories.

Term	Symbol	Initial weight	Final weight	Steps N_{des}
Action rate penalty	$r_t^{\Delta\text{act}}$	-5×10^{-3}	-1×10^{-2}	30 000
Joint velocity penalty	r_t^{joint}	-1×10^{-4}	-1×10^{-3}	30 000

Table 4.4: Curriculum schedule for the `Lift` task.

Terminations. Two termination conditions are:

- **time-out**: the episode terminates after 2.5 s;
- **object dropping**: if the object root frame goes below -0.2 m along the z-axis, the episode ends early, since this corresponds to the cube falling off the table.

4.2.3 OpenDrawer

The `OpenDrawer` environment extends the previous `Reach` task by adding a cabinet with a drawer in front of the robot. In an 8 s episode the policy must execute a manipulation micro-sequence that involves:

1. reaching the drawer handle with the end-effector,
2. grasping the handle,
3. pulling the drawer along its opening direction.

As in the other tasks, the scene is randomized at reset so that the learned policy does not overfit a single cabinet and robot configuration.

Scene and robot instantiation. At the beginning of each episode the arm joints are reset to the same nominal configuration used for the `Reach` task,

$$\mathbf{q}_0^{\text{arm}} = (0.0,\, -1.712,\, 1.712,\, 0.0,\, \tfrac{\pi}{2},\, 0.0)\ rad.$$

The cabinet is instantiated as an articulated object from the Isaac Sim asset library, with root pose

$$\mathbf{p}_0^{\text{cab}} = (1.5,\, 0.0,\, -0.35)\ m, \qquad \mathfrak{q}_0^{\text{cab}} = (0.0,\, 0.0,\, 0.0,\, 1.0)\ rad,$$

and with all door and drawer joints initially closed. In the implementation four joints are modelled, but for the purpose of this task only the prismatic joint q_t^{dr} associated with the top drawer is considered, and it is used to measure the drawer opening. We also define a reference frame attached to the drawer handle, whose pose at time t is denoted by

$$\mathbf{r}_t^{\text{h}} = \left(\mathbf{p}_t^{\text{h}},\, \mathfrak{q}_t^{\text{h}}\right),$$

where $\mathbf{p}_t^{\text{h}} \in \mathbb{R}^3$ and $\mathfrak{q}_t^{\text{h}} \in \mathbb{R}^4$.

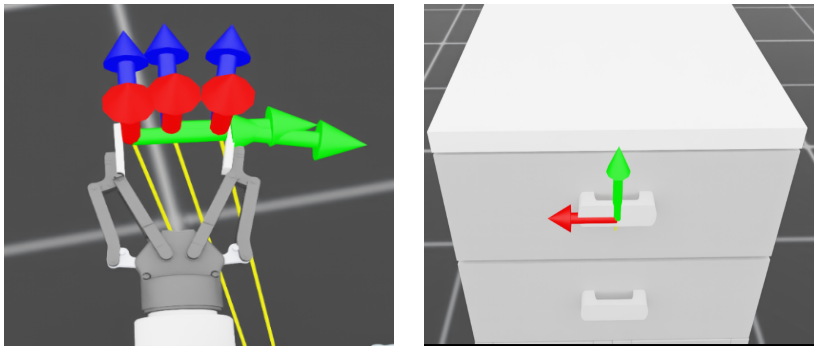
Similarly to the `Lift` task, the end-effector is provided with a dedicated reference frame located between the two fingertips of the gripper pads; in addition, two further frames are defined, one for each pad. We denote the poses of these three frames by

$$\mathbf{r}_t^{\text{pad-r}} = \left(\mathbf{p}_t^{\text{pad-r}},\, \mathfrak{q}_t^{\text{pad-r}}\right), \qquad \mathbf{r}_t^{\text{pad-l}} = \left(\mathbf{p}_t^{\text{pad-l}},\, \mathfrak{q}_t^{\text{pad-l}}\right), \qquad \mathbf{r}_t^{\text{gr}} = \left(\mathbf{p}_t^{\text{gr}},\, \mathfrak{q}_t^{\text{gr}}\right).$$

In the implementation these frames are realised in Isaac Lab as a `FrameTransformer` attached to the TCP link, with a fixed translational offset along its local z-axis for the gripper-centre frame, and additional target frames attached to the inner pads of the left and right fingers to monitor the contact configuration with the drawer handle. The Figure 4.5 show such reference frame

Observations. The observation vector is the concatenation of six terms:

- $\mathbf{q}_t - \mathbf{q}_0$: arm and gripper joint positions expressed as offsets from the initial configuration;
- $\dot{\mathbf{q}}_t^{\text{arm}} - \dot{\mathbf{q}}_0^{\text{arm}}$: arm joint velocities expressed as offsets from the initial rest state (with $\dot{\mathbf{q}}_0^{\text{arm}} = \mathbf{0}$);
- q_t^{dr}, $\dot{q}_t^{\text{dr}}$: the prismatic joint position and velocity of the top drawer, so that the agent knows how much the drawer has been opened and how fast it is moving;
- $\mathbf{d}_t^{\text{gr-h}}$: a custom term encoding the relative gripper-handle displacement, built from the two frame transformers;



(a) End-effector frames (b) Drawer-handle frame

Figure 4.5: Reference frames used for the gripper and the drawer handle in the manipulation tasks.

- $\mathbf{a}_{t-1}$: the last action, to help the policy smooth the command sequence.

Where the only component that can be useful to explain is the distnace between the `FrameTransformer` of the end-effector and the handle, defined as:

$$\mathbf{d}_t^{\text{gr-h}} = \mathbf{p}_t^{\text{h}} - \mathbf{p}_t^{\text{gr}} \in \mathbb{R}^3$$

The overall, noise-free observation vector at time t is defined as

$$\hat{\mathbf{o}}_t = \left((\mathbf{q}_t - \mathbf{q}_0)^\top,\ (\dot{\mathbf{q}}_t^{\text{arm}} - \dot{\mathbf{q}}_0^{\text{arm}})^\top,\ q_t^{\text{dr}},\ \dot{q}_t^{\text{dr}},\ (\mathbf{d}_t^{\text{gr-h}})^\top,\ \mathbf{a}_{t-1}^\top\right)^\top \in \mathbb{R}^{25},$$

To improve robustness, the actual observation fed to the policy is obtained by adding elementwise uniform noise to selected components:

$$\mathbf{o}_t = \hat{\mathbf{o}}_t + \boldsymbol{\eta}_t,$$

where the components of $\boldsymbol{\eta}_t$ corresponding to joint positions and velocities (for both the robot and the drawer joint) are uniformly and independently sampled from $\mathcal{U}[-0.01, 0.01]$, those corresponding to the relative gripper-handle term $\mathbf{o}_t^{\text{gr-h}}$ are uniformly sampled from $\mathcal{U}[-0.05, 0.05]$, and all other entries, are left noise-free.

Rewards. The total reward is defined as:

$$\begin{aligned} r_t = {} & w_{\text{reach}}\, r_t^{\text{reach}} + w_{\text{align}}\, r_t^{\text{align}} + w_{\text{wrap}}\, r_t^{\text{wrap}} \\ & + w_{\text{gr-app}}\, r_t^{\text{gr-app}} + w_{\text{grasp}}\, r_t^{\text{grasp}} + w_{\text{open}}\, r_t^{\text{open}} \\ & + w_{\text{stage}}\, r_t^{\text{stage}} + w_{\Delta\text{act}}\, r_t^{\Delta\text{act}} + w_{\text{act}}\, r_t^{\text{act}} + w_{\text{joint}}\, r_t^{\text{joint}}, \end{aligned} \tag{4.24}$$

with weights reported in Table 4.5.

(1) Approach EE–handle. This term rewards bringing the gripper close to the drawer handle. Let $\mathbf{d}_t^{\text{gr-h}} \in \mathbb{R}^3$ be the vector from the gripper frame to the handle frame (defined as in the observation term), and let:

$$d_t \;=\; \left\|\mathbf{d}_t^{\text{gr-h}}\right\|_2 \tag{4.25}$$

be its Euclidean norm. We first define the shaping function

$$\phi(d_t) \;=\; \left(\frac{1}{1+d_t^2}\right)^2, \tag{4.26}$$

and then express the reward as

$$r_t^{\text{approach}} = \begin{cases} 2\,\phi(d_t), & d_t \le \tau, \\ \phi(d_t), & d_t > \tau, \end{cases} \tag{4.27}$$

with $\tau = 0.05\ m$ the proximity threshold. In this way the agent is gently pulled toward the handle at larger distances, and receives an additional boost once it enters the small neighbourhood of radius τ around it.

(2) EE–handle alignment. Both the gripper and the handle frames provide three orthonormal axes; let e_x, e_y, e_z be the unit vectors of the gripper frame and h_x, h_y, h_z those of the handle frame. The reward averages the three axis-wise alignments and penalizes anti-alignment by flipping the sign:

$$r_t^{\text{align}} = \frac{1}{3} \sum_{a\in\{x,y,z\}} \operatorname{sign}\Big(\langle e_a, h_a\rangle\Big)\,\langle e_a, h_a\rangle^2. \tag{4.28}$$

When the corresponding axes are parallel ($\langle e_a, h_a\rangle \approx +1$) the contribution is strongly positive, whereas anti-parallel axes ($\langle e_a, h_a\rangle \approx -1$) are penalised. This guides the wrist to approach the handle with the correct orientation to pull the drawer.

(3) Gripper wrapped around the handle. The frame transformers expose the fingertip positions of the two pads. Let $z_t^{\text{pad-l}}, z_t^{\text{pad-r}}$ and z_t^{h} denote, respectively, the vertical components of the left pad, right pad and handle positions at time t. We first define a binary indicator that checks whether the handle lies between the two pads along the z-axis:

$$r_t^{\text{wrap}} = \begin{cases} 1, & z_t^{\text{pad-l}} > z_t^{\text{h}} > z_t^{\text{pad-r}}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.29}$$

so it is active only when the handle is geometrically wrapped by the two fingers.

(4) Gripper approach around the handle. Let r_t^{wrap} be the term previously defined in Equation (4.29) , a second shaping term encourages bringing the two pads close to the handle height while they already wrap it from above and below. Let

$$d_t^{\text{L}} = \left| z_t^{\text{pad-l}} - z_t^{\text{h}} \right|, \qquad d_t^{\text{R}} = \left| z_t^{\text{pad-r}} - z_t^{\text{h}} \right|$$

be the vertical distances of the left and right pads from the handle, and let $\delta_z = 0.04\ m$ be the vertical margin. The reward term in this case is defined as:

$$r_t^{\text{gr-app}} = r_t^{\text{wrap}} \Big[(\delta_z - d_t^{\text{L}}) + (\delta_z - d_t^{\text{R}}) \Big]. \tag{4.30}$$

When the handle is wrapped between the fingertips and both pads are close to its height, the term is positive and encourages a precise approach. If the handle is still between the pads but they are far from its height, the contribution becomes negative and acts as a penalty; when the configuration is not graspable the term is identically zero.

(5) Gripper closure near the handle. The grasping reward is activated only if the end-effector is close to the handle and the gripper joint is closing. Let d_t denote the gripper-handle distance as in Equation (4.25), and let $\tau_{\text{grasp}} = 0.05\ m$ be the distance threshold. Let q_t^{gr} be the gripper joint position and $q^{\text{close}} = \pi/6\ rad$ the reference closed value. The closure is normalised by q^{close} and clipped to [0,1]:

$$r_t^{\text{grasp}} = \begin{cases} \min\Big(1,\ \max\Big(0,\ \dfrac{q_t^{\text{g}}}{q^{\text{close}}}\Big)\Big), & d_t \le \tau_{\text{grasp}}, \\ 0, & d_t > \tau_{\text{grasp}}. \end{cases} \tag{4.31}$$

In practice, the term is zero when the end-effector is far from the handle, increases linearly with the gripper closure while the joint moves from fully open to q^{close}.

(6) Continuous drawer opening. Once the grasp around the handle is established, the agent is encouraged to pull the drawer out. Let q_t^{dr} denote the position of the top drawer prismatic joint and let r_t^{wrap} be the binary term defined in Equation (4.29), which is 1 when the handle is wrapped between the two fingers and 0 otherwise. The implemented reward is:

$$r_t^{\text{open}} = \Big(1 + r_t^{\text{wrap}}\Big)\, q_t^{\text{dr}}. \tag{4.32}$$

In other words, the contribution grows proportionally to the drawer opening, and it is doubled whenever the hand maintains a correct wrap around the handle.

(7) Multi-stage drawer opening. To make progress visible to the agent, a piecewise bonus is added at three drawer positions. Let q_t^{dr} denote the top drawer joint position and r_t^{wrap} the binary wrap term defined in Equation (4.29).The reward term is defined as the sum of three components:

$$r_t^{\mathrm{stage}} = r_t^{\mathrm{easy}} + r_t^{\mathrm{med}} + r_t^{\mathrm{hard}}, \tag{4.33}$$

with

$$r_t^{\mathrm{easy}} = \begin{cases} 0.5, & q_t^{\mathrm{dr}} > 0.01, \\ 0, & q_t^{\mathrm{dr}} \leq 0.01, \end{cases} \tag{4.34}$$

$$r_t^{\mathrm{med}} = \begin{cases} r_t^{\mathrm{wrap}}, & q_t^{\mathrm{dr}} > 0.2, \\ 0, & q_t^{\mathrm{dr}} \leq 0.2, \end{cases} \tag{4.35}$$

$$r_t^{\mathrm{hard}} = \begin{cases} r_t^{\mathrm{wrap}}, & q_t^{\mathrm{dr}} > 0.3, \\ 0, & q_t^{\mathrm{dr}} \leq 0.3. \end{cases} \tag{4.36}$$

A small opening is always rewarded by r_t^{easy}, while larger openings at $0.2\ m$ and $0.3\ m$ contribute only if the hand is in a plausible grasp configuration around the handle.

(8) Regularization terms. As in the previous tasks, smoothness is encouraged through quadratic penalties on the commands and on the joint velocities. A first term penalises changes in the action between consecutive steps, while a second term penalises the magnitude of the action itself; a third term penalises large joint velocities:

$$r_t^{\Delta\mathrm{act}} = \left\| \mathbf{a}_t - \mathbf{a}_{t-1} \right\|_2^2, \qquad r_t^{\mathrm{act}} = \left\| \mathbf{a}_t \right\|_2^2, \qquad r_t^{\mathrm{joint}} = \left\| \dot{\mathbf{q}}_t \right\|_2^2. \tag{4.37}$$

Their contributions to the total reward are scaled by negative weights.

Each of the previously defined reward terms is then multiplied, at each step t by its corresponding weight as shown in Equation (4.24). All the values are reported in Table 4.5:

Events. At every episode `reset` the environment applies a fixed sequence of active events:

- the whole scene is restored to its default state;
- the robot joints are reset around the nominal configuration, with joint positions scaled in $(0.75, 1.25)$ and zero joint velocity;

Term	Eq.	Weight
EE-handle approach	r_t^{approach}	1.0
EE-handle alignment	r_t^{align}	0.5
Gripper approach around handle	$r_t^{\text{gr-app}}$	5.0
Gripper wrapped around handle	r_t^{wrap}	0.5
Gripper closure	r_t^{grasp}	2.0
Continuous drawer opening	r_t^{open}	7.5
Multi-stage drawer opening	r_t^{stage}	1.0
Action rate penalty	$r_t^{\Delta\text{act}}$	-1×10^{-2}
Action magnitude penalty	r_t^{act}	-5×10^{-3}
Joint velocity penalty	r_t^{joint}	-1×10^{-3}

Table 4.5: Reward weights for the `OpenDrawer` task.

- the cabinet root pose is resampled, keeping x and y fixed and perturbing only the height with $z \in [0.1, 0.2]$ m.

In addition, at `startup` the contact material of the drawer handle is randomised: the static friction coefficient is sampled in $[1.0, 1.25]$, the dynamic friction in $[1.25, 1.5]$, and the restitution is kept at zero. These reset-time and material perturbations reduce sensitivity to a single cabinet placement and a single handle-finger friction value.

Terminations. Also in this case the only active termination term is the timeout at 8 s.

Chapter 5

Validation and transfer from simulation to real robot

This chapter describes how the policies trained in Isaac Lab are validated and progressively transferred from pure simulation to the physical UR10e robot. Rather than jumping directly from training to hardware, the deployment is organized as a sequence of increasingly realistic stages as presented in Section 3.2. Each layer is designed to detect different classes of issues before they can affect the real system.

5.1 Simulation-to-Simulation

A natural intermediate step between policy training and execution on the real UR10e consists in replaying the learned policy inside a simulator. This phase, often referred to as *Simulation-to-Simulation* (Sim2Sim), is useful to answer a very concrete question before involving real hardware: *does the trained policy actually solve the task, and does it do so with motion profiles that are acceptable for a real robot?*. As shown in Figure 3.3 this validation is articulated in two layers. First, we run the policy inside Isaac Sim, which shares the same physics backend used during training in Isaac Lab and therefore isolates purely *execution-related* problems (too fast motions, wrong grasp heights, gripper collisions). Second, we replay the same policy through a ROS 2 control stack connected to URSim. This second step is closer to deployment because it tests not only the policy, but also the whole communication and control pipeline (topics, controller configuration, timing).

5.1.1 Policy validation in Isaac Sim

After the PPO policy has converged in Isaac Lab, the quickest and least invasive check consists in loading exactly the same task in Isaac Sim and feeding the policy

outputs to the simulated UR10e. Since training is performed in headless mode, this is also the first opportunity to visually inspect that the learned behaviour qualitatively matches the intended task, as illustrated in Figure 5.1.

Since both components rely on the same Omniverse/PhysX[27] simulation stack and use the same USD-based articulation description, this test removes from the equation most of the classical sources of domain shift (different friction models, slightly different damping values, different link inertias). What remains to be assessed is therefore the *execution quality*: the manipulator must reach the target, follow the intended approach strategy, and—most importantly—it must do so without overly aggressive joint accelerations or undesired contacts between the gripper fingers and the table or the drawer front.

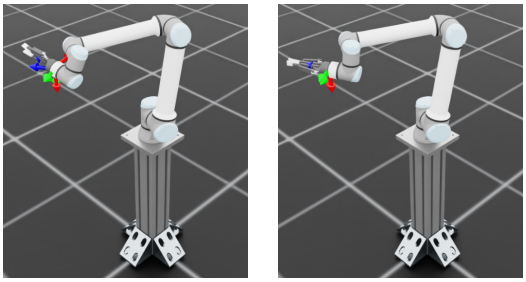
This point is not marginal. A policy that in training was rewarded mainly for completing the task, so, it might legitimately converge to solutions that exploit the simulator: fast wrist swings, sudden changes of direction, or sliding of the fingertips on the support surface may all be acceptable in Isaac Lab, but the very same motions would immediately trigger a protective stop on the real UR10e.

5.1.2 Policy validation in URSim

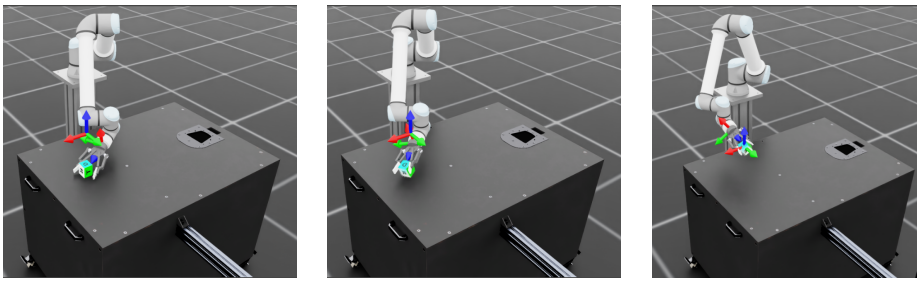
The second part of the Sim2Sim pipeline is more interesting from a deployment perspective, because it reproduces exactly the same communication architecture that will later be used with the physical robot. The setup is based on two Docker containers connected through a dedicated Docker network: one container runs URSim, which emulates the UR10e controller, while the other runs the ROS 2 stack. The custom network makes the IP addressing stable and lets the ROS 2 nodes talk to the simulated controller as if it were a real UR robot on the LAN.

On the ROS 2 side we adopt the official `universal_robots_ros2_driver` for Humble [37] to establish the connection with the URSim instance. This driver exposes the robot as a standard ROS 2 articulation and provides the usual control interfaces (joint state publisher, trajectory controllers, IO topics) by internally using the UR RTDE protocol. From the point of view of our policy node, this is the key property: sending a command to URSim or to the real UR10e becomes the *same* ROS 2 operation. Topic names, controller names, and message types do not change. This is exactly the benefit anticipated in the introduction: once the control pipeline work in simulation, the same code can be reused on the real robot with only minor adjustments to network parameters.

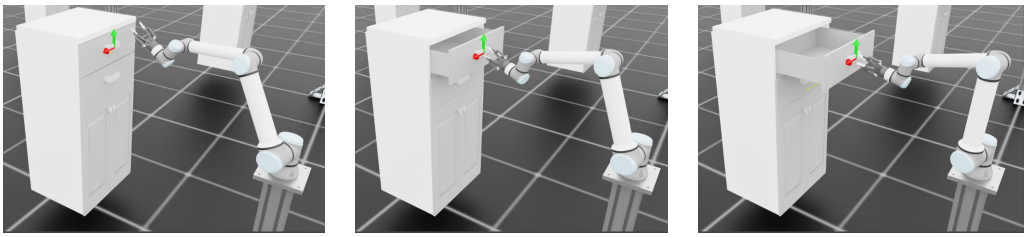
Controller choice. In the ROS 2 setup based on the UR driver we use the version of the trajectory controller that is aware of the robot's speed scaling, rather than a plain `joint_trajectory_controller` that only relies on wall-clock time. The standard controller executes joint-space trajectories exactly according to the



(a) Approach to target (b) Alignment at target



(c) Pre-grasp pose (d) Grasping phase (e) Lifted object



(f) Approach to handle (g) Handle grasp (h) Drawer fully opened

Figure 5.1: Qualitative rollouts of the trained policies in Isaac Sim for the three tasks: `Reach` (top row), `Lift` (middle row), and `OpenDrawer` (bottom row)

timestamps contained in the incoming message: if the trajectory says that the target must be reached in 2 s, the controller will advance its internal state over 2 s of real time [37]. This becomes problematic when the robot itself slows down the motion (for example because the speed slider is not at 100% or because a safety mode is active), since the controller has no direct knowledge of this slowdown and may consider the trajectory completed while the robot is still moving.

The UR driver, instead, exposes the current speed-scaling value coming from the UR controller and provides a trajectory controller that advances according to this *scaled* time [37]. If the robot runs at 50% of the nominal speed, the controller also progresses at 50%, keeping the ROS 2 side and the UR side synchronised and avoiding premature completion of the trajectory. It is important to note that this

mechanism does not reshape the trajectory to satisfy stricter velocity or acceleration limits; it only ensures that trajectory execution is tracked in the same time base as the robot.

This behaviour is particularly useful for the reach and lift tasks considered in this work. In the reach task, the arm must pass near configurations that could bring the robot close to self-collision. In the lift task, the end-effector moves very close to the table surface.So in both cases, slowing the motion down to 50% at controller level provides an additional safety margin against incidental collisions.

Code-level observation handling for the lift task. A small but important detail emerges when we try to replay the `lift` policy outside of the original Isaac Lab environment. The policy was trained assuming that the pose of the cube was part of the observation vector (typically a 7-dimensional quantity collecting position and orientation). During deployment, however, we do not have a dedicated camera in the loop, and the policy still expects that slot in the observation. To preserve compatibility we implement a simple observation switch at code level:

1. until the grasp event is detected, the policy receives the original "static" 7D object pose, i.e., the same structure it saw during training;

2. as soon as the grasp succeeds, we overwrite that part of the observation with the forward kinematics of the end-effector plus a fixed tool offset, under the assumption that the object is now rigidly attached to the gripper.

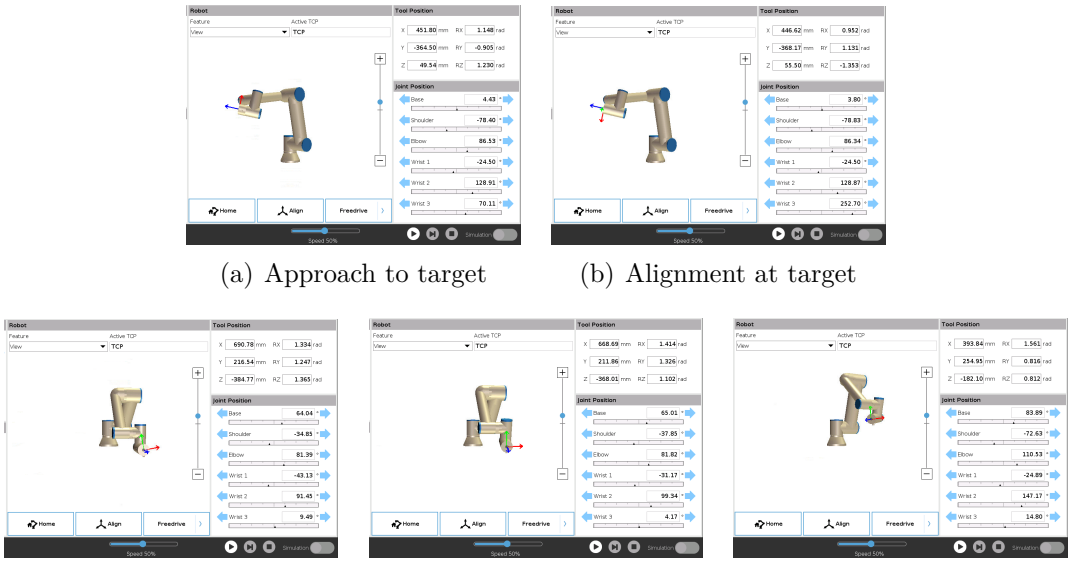
In practice, the grasp event is declared when the distance between the end-effector and the object falls below a small threshold and, at the same time, the gripper action commands a closing motion.

$$g_t = \begin{cases} 1, & \text{if } \left\| \mathbf{p}_t^{\text{ee}} - \mathbf{p}_t^{\text{obj}} \right\|_2 \leq \varepsilon_{\text{grasp}} \ \wedge \ a_t^{\text{gr}} < 0, \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

In other words, after grasping we stop treating the object as an independent body whose pose must be sensed, and we start *deriving* its pose from the robot kinematics. This trick is coherent with the real setup (once the object is in the fingers, it moves with the end-effector) and, at the same time, it keeps the observation shape exactly equal to what the policy was trained on.

5.2 Simulation-to-Reality

After validating the control pipeline against URSim, the final step consists in redirecting the same ROS 2 nodes to the real UR10e controller. At this stage the



(a) Approach to target (b) Alignment at target

(c) Pre-grasp pose (d) Grasping phase (e) Lifted object

Figure 5.2: Qualitative rollouts of the trained policies in URSim for two tasks: `Reach` (top row), `Lift` (bottom row)

overall architecture remains unchanged. The transition therefore mainly involves changing the network endpoint from the Docker network used for URSim to the IP address of the physical controller and loading the appropriate safety configuration on the robot.

Unlike the URSim stage, however, the real robot must also execute the grasping actions commanded by the policy. For this reason we have to remove the fake gripper and integrate the `robotiq_2f_urcap_adapter` package [38], which bridges the Robotiq 2F-140 gripper mounted on a UR controller with ROS 2 topics and services. This adapter connects to the URCap running on the controller and exposes the typical open/-close commands as ROS entities. In this way the same policy that in Isaac Lab learned to close the gripper at a specific stage of the task can now trigger a real end-effector action on the physical setup, without changing its internal logic and without introducing a separate ad-hoc communication channel just for the gripper.

To further mitigate timing variability and to make the commanded trajectories appear smoother on the manipulator, we installed a real-time Linux kernel (PREEMPT_RT) on the ROS 2 host running the UR ROS 2 driver, following the recommendations in the Universal Robots documentation [43]. This setup significantly reduced the jitter observed in the joint updates and made the overall motion execution noticeably more fluid.

Trajectory time allocation. The ROS 2 policy node does not stream full joint trajectories computed offline, but rather generates at each timer callback a short horizon command that is compatible with the `scaled_joint_trajectory_controller`. Let $\mathbf{q}_{t-1}^{\text{arm}} \in \mathbb{R}^6$ be the most recent joint state received from the robot and $\mathbf{q}_t^{\text{arm}} \in \mathbb{R}^6$ the target configuration produced by the policy (both computed following the Equation (4.1)). To avoid abrupt changes, the commanded joint position is obtained through an moving-average smoothing:

$$u_{t,i}^{\text{arm}} = (1-\alpha)\, q_{t-1,i}^{\text{arm}} + \alpha\, q_{t,i}^{\text{arm}}, \qquad i = 1,\ldots,6. \tag{5.2}$$

for each joint $i = 1,\ldots,6$. This filter dampens high-frequency oscillations that may appear in the policy output while still allowing the robot to converge to the desired pose in a few iterations.

Given a maximum admissible joint speed $\dot{q}^{\max}$ (in the implementation set to 0.5 rad/s) and a minimum trajectory duration $\Delta t_{\min}$, the node computes for every joint the time needed to move from the current to the commanded position:

$$\Delta t_i = \max\left(\frac{|q_i^{\text{cmd}} - q_i^{\text{curr}}|}{\dot{q}^{\max}},\, \Delta t_{\min}\right). \tag{5.3}$$

Since the controller expects a single `JointTrajectoryPoint` with a single `time_from_start`, the final duration associated to the trajectory point is chosen as the maximum over all joints,

$$\Delta t = \max_i \Delta t_i, \tag{5.4}$$

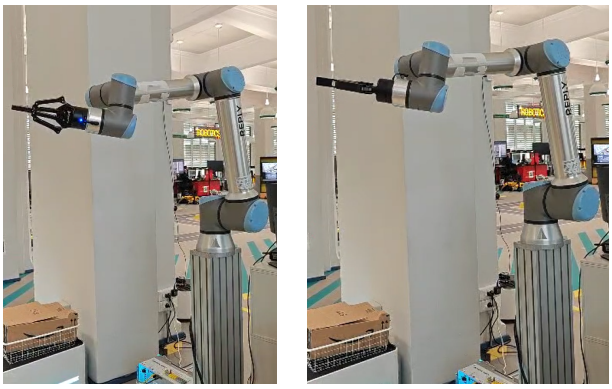
so that no joint is forced to exceed the velocity bound.

Gripper command handling. The end-effector is commanded with the same logic of the training expressed by Equation (5.5) but it's adapted to work with the robotiq driver where the close command is 0.14 m and the opening command is 0.0 m, in other words:

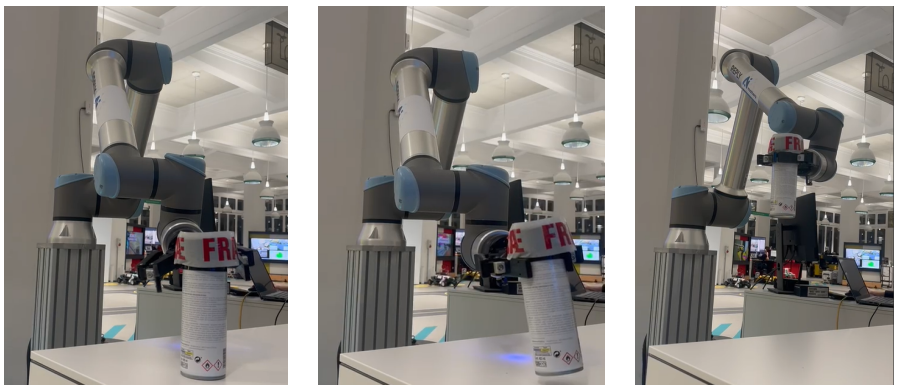
$$u_t^{\text{gr}} = \begin{cases} 0.14 & \text{if } a_t^{\text{gr}} < 0 \quad \text{(closed gripper)}, \\ 0.0 & \text{if } a_t^{\text{gr}} \geq 0 \quad \text{(open gripper)}. \end{cases} \tag{5.5}$$

Thanks to the `robotiq_2f_urcap_adapter` [38], the selected command u_t^{gr} is forwarded to the URCap running on the controller and executed on the physical Robotiq 2F-140 gripper.

Once these conditions are met, the Sim2Real execution becomes a straightforward continuation of the Sim2Sim pipeline.



(a) Approach to target (b) Alignment at target



(c) Pre-grasp pose (d) Grasping phase (e) Lifted object

Figure 5.3: Qualitative rollouts of the trained policies executed on the real UR10e for two tasks: `Reach` (top row) and `Lift` (bottom row)

Chapter 6

Experimental Results

This chapter presents the results obtained with the control policies trained in simulation and deployed on the real UR10e robot. The `Reach` and `Lift` policies are evaluated in Isaac Lab, in URSim, and on the physical UR10e, while the `OpenDrawer` task is evaluated only in Isaac Lab.

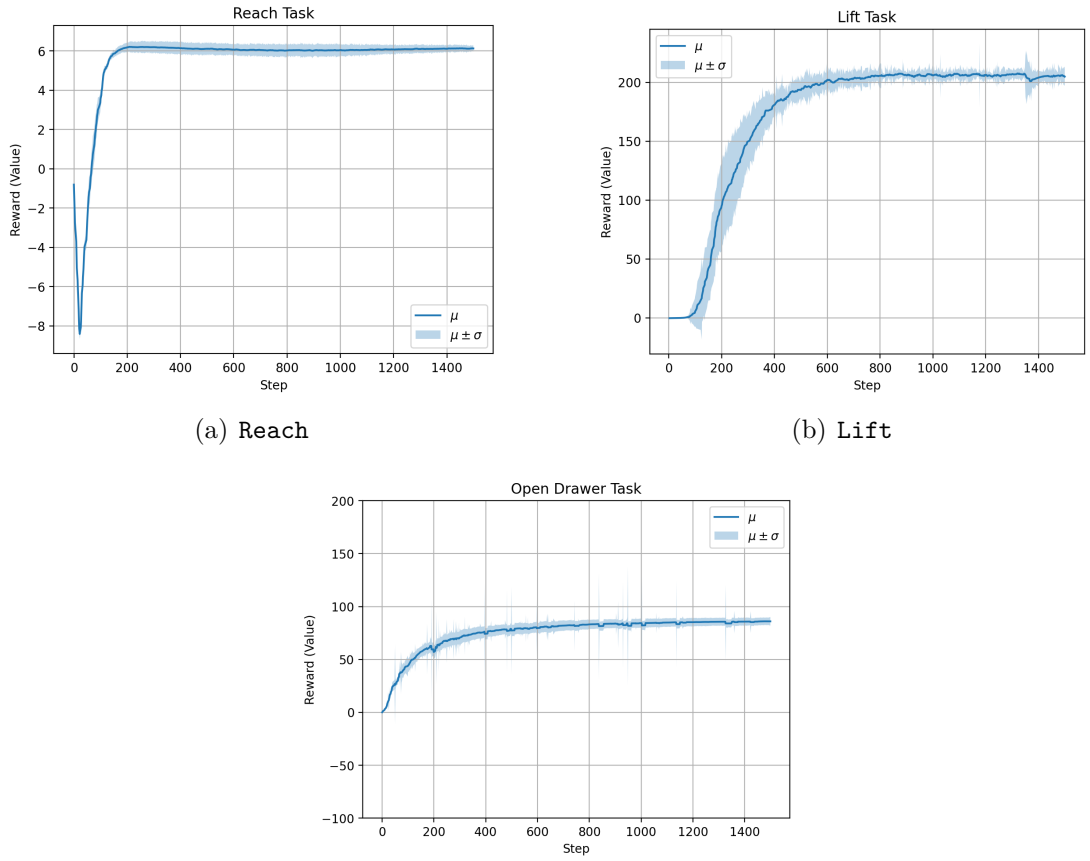
The evaluation is organised in two main parts:

1. We analyse the training performance, reporting learning curves averaged over multiple random seeds in order to characterise how quickly and how reliably each task is learned;

2. We move to the experimental evaluation, where we introduce the evaluation protocol and metrics used across all experiments, and present task-wise results in simulation, URSim, and on the real robot, concluding with a discussion of the observed sim-to-real gap and of the overall behaviour of the learned policies.

6.1 Training performance in simulation

Before analysing the deployment in URSim and on the physical UR10e, this section reports the training performance of the policies in the Isaac Lab simulation environment. For each task, the policy is trained with five different random seeds in order to assess the variability of the learning process and to avoid drawing conclusions from a single potentially lucky or unlucky run. We then plot the mean episode reward together with a shaded band corresponding to $\mu \pm \sigma$ across seeds, where μ and σ denote the empirical mean and standard deviation over seeds at each training step.



(a) `Reach`

(b) `Lift`

(c) `OpenDrawer`

Figure 6.1: Training performance for the three manipulation tasks. Each plot shows the mean episode reward over five random seeds (solid line) and the corresponding $\mu \pm \sigma$ band (shaded area).

6.2 Evaluation protocol and metrics

For each task and for each environment, the learned policies are evaluated over multiple episodes starting from initial conditions sampled from the same distributions used during training. Unless otherwise specified, the reported values are episode averages over all runs for the corresponding task and environment.

Task-level metrics. The main performance indicators used throughout this chapter are:

- **Success rate** SR [%]: fraction of episodes in which the task-specific success condition is triggered before the time limit. For the `Reach` task, success is

defined as the end-effector entering a sphere of radius ε_p around the target position while also satisfying an orientation constraint, and remaining within this tolerance for 50 consecutive control steps. For the `Lift` task we have 2 different kind of SR,the first one require that the object must be grasped, lifted above a height threshold h_{th}, and kept close to the target pose for 50 steps. The other is more soft, in fact it only checks that the object is stably grasped and lifted above h_{th}, independently of the final target pose. For the `OpenDrawer` task (simulated only), success requires the drawer joint to exceed a specified opening threshold while the gripper remains close to the handle always for 50 steps.

- RMSE [m]: root-mean-square error between the relevant body and its target over the last 50 control steps of the episode. For `Reach` this is the TCP-to-target error, whereas for `Lift` and `OpenDrawer` it measures the object-target or end-effector-handle distance depending on the task definition. An analogous RMSE is also computed for the drawer opening with respect to the desired opening value.
- **Episode length** T [steps]: number of control steps elapsed from the beginning of the episode until termination due to success.

These metrics are reported in the sections below and are used to quantify both the overall reliability of the learned behaviours and the degradation incurred when moving from simulation to the real UR10e.

6.3 Simulation Results

This section reports the performance of the three manipulation tasks evaluated in simulation. All metrics are reported as mean ± standard deviation computed over the five seeds.

6.3.1 Reach

For each episode of the `Reach` task we record a binary success flag, the episode length in control steps, and the TCP-target position and orientation RMSE over the last 50 control steps before termination, i.e., over the same time window used in the success condition. The success criterion requires both the position and orientation errors to remain below $0.25\,m$ and 17°, respectively, for 50 consecutive steps.

The main performance indicators are summarized in Table 6.1. Overall, these results indicate that the `Reach` policy learns a reliable controller that drives the end-effector close to the target pose with good positional and acceptable orientational

accuracy in simulation.

Metric	**Value (Simulation)**
Success rate [%]	97.7 ± 0.9
RMSE_{pos} $[m]$	0.0103 ± 0.0004
RMSE_{ori} [°]	12.0 ± 1.3
Episode length [steps]	65.3 ± 0.7

Table 6.1: Simulation performance of the `Reach` task, reported as mean ± standard deviation over 5 random seeds.

Figure 6.2 shows the per-seed success rates corresponding to Table 6.1. The bars are very close to each other, confirming that the variability across seeds is limited and that the aggregate metrics in Table 6.1 are representative of all runs rather than being dominated by a single lucky seed. This plot is intended as an indicative visual summary rather than a detailed statistical analysis: it allows one to verify at a glance that the training outcome for `Reach` is stable with respect to the random initialization.

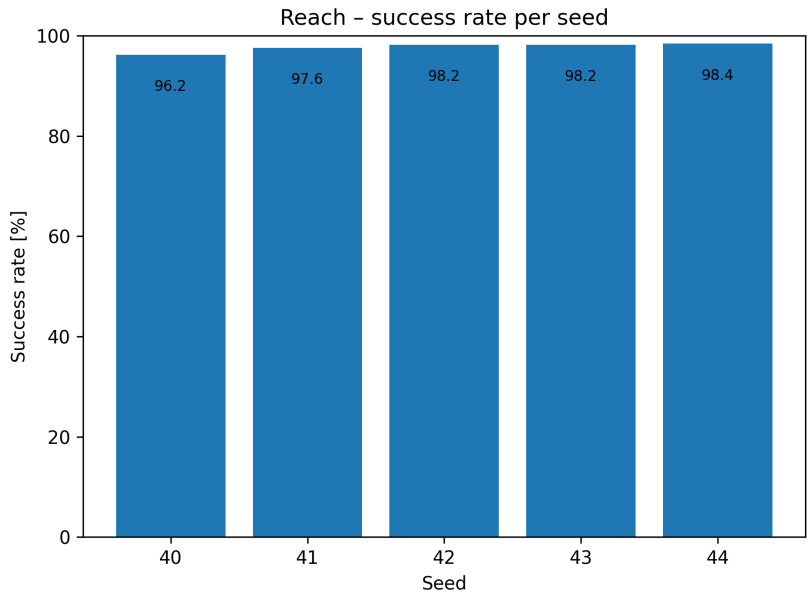


Figure 6.2: Per-seed success rate in simulation for the `Reach` task (seeds 40-44).

6.3.2 Lift

The `Lift` task extends `Reach` by requiring the robot to grasp and lift an object before bringing it towards a desired pose. In practice, the task naturally decomposes into two phases: a *lift* phase, in which the object must be securely picked up from the table, and a subsequent *placement* phase, in which the grasped object is moved and stabilised near the target pose for at least 50 steps.

Success metrics. For each rollout we therefore compute two binary success indicators:

- a *lift success* flag, which checks whether the object has been grasped and clearly lifted off the table. An episode is considered a lift success if the object is held by the gripper and its height exceeds $5\,cm$ above the table for at least 50 consecutive control steps, independently of the final target pose;
- a *placement success* flag, where an episode is counted as successful if the distance between the object and the target position remains below a given threshold for the last 50 control steps. This same window is used to define the object-target $\mathrm{RMSE}_{\mathrm{pos}}$.

The first metric focuses on the ability of the policy to establish a stable grasp and lift the object off the surface, while the second captures the full transport-and-placement performance.

Table 6.2 reports the aggregated performance in simulation over the five random seeds. The simulated `Lift` policy attains an average *lift* success rate of $84.6\% \pm 3.93\%$, confirming that in most simulated rollouts the object is successfully grasped and lifted from the table. The stricter *placement* success rate is $78.5\% \pm 5.2\%$. The mean object-target $\mathrm{RMSE}_{\mathrm{pos}}$ is $0.0194\,m$ with a standard deviation of $0.0036\,m$, corresponding to positioning errors on the order of a couple of centimetres at the end of the episode. The average episode length is 88.4 ± 4.2 steps. Compared to the simpler `Reach` task, the lower placement success rate and higher variability reflect the increased difficulty of reliably establishing a stable grasp, lifting the object and precisely placing it under the applied domain randomization.

The same type of bar visualization used for `Reach` is reported for `Lift`, which displays the *placement* success rate obtained by each of the five seeds. The bar chart is meant as an indicative tool to quickly assess the stability of the results, rather than as a substitute for the numerical summary in Table 6.2. For the subsequent deployment experiments in URSim and on the real robot, we selected the policy corresponding to seed 42. This seed provides performance that is close to the multi-seed averages, and in simulation it exhibited a smoother,non-aggressive interaction with the object and the table, which made it a natural candidate for transfer to the physical setup.

Metric	Value (Simulation)
Lift success rate [%]	84.6 ± 3.93
Lift episode length [steps]	69.2 ± 5.9
Placement success rate [%]	78.5 ± 5.2
RMSE_{pos} $[m]$	0.0194 ± 0.0036
Placement episode length [steps]	88.4 ± 4.2

Table 6.2: Simulation performance of the `Lift` task, reported as mean ± standard deviation over 5 random seeds. The table includes both the lift success rate and the stricter placement success rate.

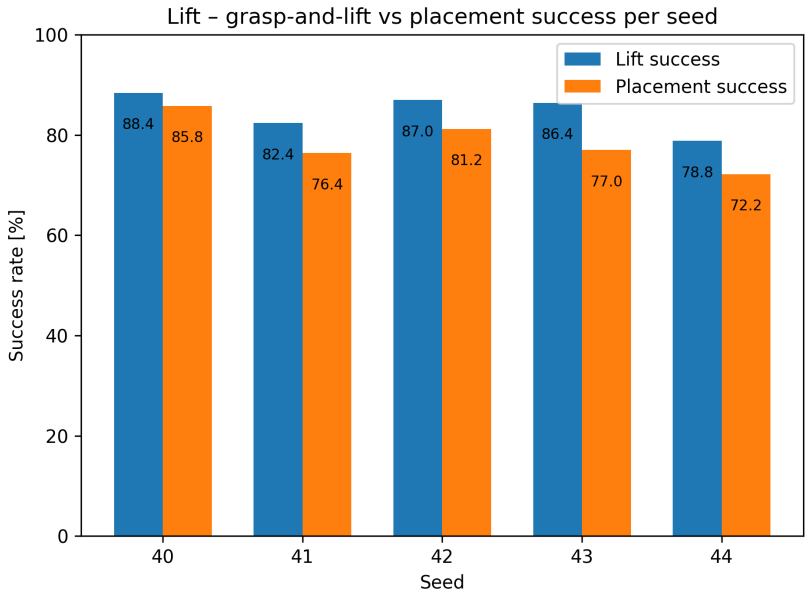


Figure 6.3: Per-seed lift and placement success rates in simulation for the `Lift` task (seeds 40-44).

Figure 6.3 shows, for each seed, both the lift and the stricter placement success rates. The former is consistently higher, confirming that most simulated rollouts succeed in grasping and lifting the object, while some of them only fail in the final precise placement phase.

6.3.3 OpenDrawer

The `OpenDrawer` task is the most complex scenario considered in this work. The policy must first approach and align the gripper with the drawer handle, establish

contact, and then execute a coordinated pulling motion to open the drawer. In this case we evaluate two RMSE measures over the last 50 control steps of each episode: the TCP-handle distance $\text{RMSE}_{\text{handle}}$ and the error on the drawer opening with respect to the fully open configuration $\text{RMSE}_{\text{maxOpen}}$, in addition to the binary success flag and the episode length.

As shown in Table 6.3, the `OpenDrawer` policy achieves an average success rate of $74.5\% \pm 11.5\%$ across seeds. The mean TCP-handle RMSE ($\text{RMSE}_{\text{handle}}$) in the last-50 window is $0.0364\,m$ with a standard deviation of $0.0011\,m$, indicating that the end-effector typically remains within a few centimetres of the handle during the final phase. The mean drawer opening RMSE($\text{RMSE}_{\text{maxOpen}}$) is $0.0090\,m \pm 0.0001\,m$ in the chosen opening coordinate, meaning that successful episodes reach a configuration close to the fully open drawer. The average episode length is 307.4 ± 42.9 steps, which is significantly larger than for `Reach` and `Lift` and reflects the increased complexity of the task.

Metric	**Value (Simulation)**
Success rate [%]	74.5 ± 11.5
$\text{RMSE}_{\text{handle}}$ $[m]$	0.0364 ± 0.0011
$\text{RMSE}_{\text{maxOpen}}$ $[m]$	0.0090 ± 0.0001
Episode length [steps]	307.4 ± 42.9

Table 6.3: Simulation performance of the `OpenDrawer` task, reported as mean ± standard deviation over 5 random seeds.

Figure 6.4 reports the per-seed success rates for the `OpenDrawer` task. In contrast to `Reach` and `Lift`, here the bars show a noticeably larger spread: some seeds achieve success rates close to 90%, while others are significantly lower. This behaviour is consistent with the higher standard deviation reported in Table 6.3, and reflects the increased sensitivity of this contact-rich, multi-stage task to the particular random initialization and training trajectory. As before, the bar chart should be interpreted as a compact visual aid that highlights the variability across seeds at a glance.

6.4 Real-robot experiments

After validating the policies in Isaac Lab, we deploy the `Reach` and `Lift` controllers on the physical UR10e robot equipped with the Robotiq 2F-140 gripper. The deployment pipeline reuses the same ROS 2 infrastructure and action conventions described in Section 5.2 (see Chapter 5),so that the policy outputs can be replayed on the real arm with minimal modifications to the code. In this section we present

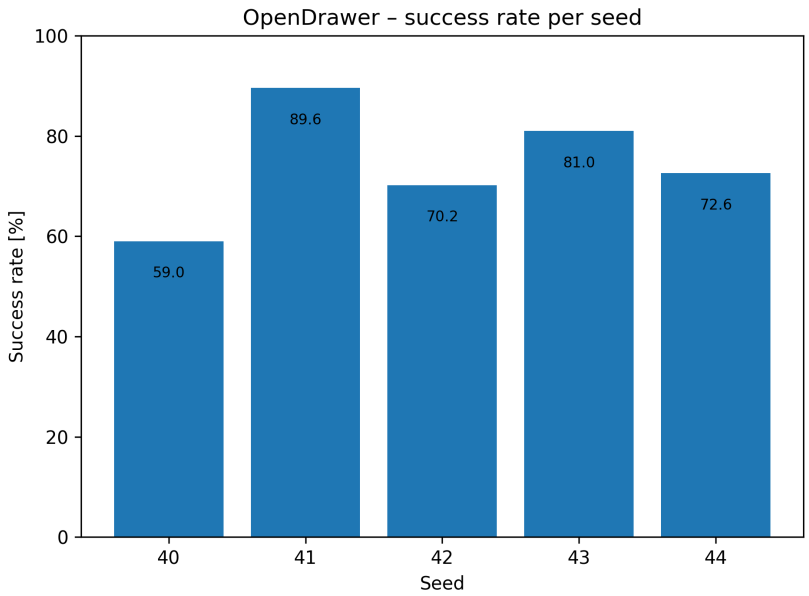


Figure 6.4: Per-seed success rate in simulation for the `OpenDrawer` task (seeds 40-44).

the task-wise results for `Reach` and `Lift` on the real system.

6.4.1 Reach

On the real UR10e, the `Reach` policy trained with seed 42 achieves a strict success rate of 100% over the ten evaluation episodes reported in Table 6.4. In all trials the end-effector is able to reach and stabilise near the commanded target pose. The mean TCP-target position RMSE over the last 50 control steps is $0.0174\,m$ with a standard deviation of $0.0026\,m$, which corresponds to position errors on the order of two centimetres and shows limited dispersion across episodes. The mean orientation RMSE in the same window is $16.0° \pm 0.7°$, indicating that the final TCP orientation remains close to the desired one, albeit with slightly larger errors than those observed in simulation. The average episode length is 2116 control steps with a standard deviation of 190 steps.

Qualitatively, the behaviour observed on the real robot is consistent with the simulation results: in all evaluated episodes the end-effector converges smoothly towards the target and remains close to the desired pose for the remainder of the rollout. From a quantitative standpoint, the real-robot experiments exhibit a 100% success rate over ten trials, which is compatible with the approximately 98% success obtained in simulation given the much smaller sample size. However, the

Metric	Value (Real robot)
Success rate [%]	100
RMSE_{pos} $[m]$	0.0174 ± 0.0026
RMSE_{ori} [°]	16.035 ± 0.7476
Episode length [steps]	2116.2 ± 190.4321

Table 6.4: Performance of the `Reach` policy (seed 42) on the real UR10e.

final TCP-target RMSE values are systematically higher on the real robot (about $1.7\,cm$ in position and 16° in orientation) than in simulation (about $1.0\,cm$ and 12°), reflecting the additional disturbances and model mismatches present in the real setup. This gap will be discussed more systematically in Section 6.5 when comparing the simulation and real-robot metrics side by side.

6.4.2 Lift

For the `Lift` task, the policy trained with seed 42 is deployed on the real UR10e in a scenario where the robot must grasp a spray can on the table and lift it towards a fixed target pose in free space. Ten episodes are executed under identical initial conditions. As in simulation, the task is naturally decomposed into two phases: a *lift* phase, where the object must be securely picked up from the table, and a *placement* phase, where the grasped object is moved and stabilised near the target pose.

To mirror the simulation analysis, we evaluate two separate success metrics on the real robot:

- a *lift success* flag, which is set to one if the object is held by the gripper and its height exceeds $0.05\,m$ above the table for at least 50 consecutive control steps, regardless of the final target pose;
- a *placement success* flag, which reuses the strict definition adopted in simulation: an episode is counted as successful if the distance between the object and the target pose remains below a given threshold for the last 50 control steps.

In addition, we compute the object-target RMSE_{pos} and the episode length in control steps.

The resulting real-robot performance is summarised in Table 6.5.

The values in Table 6.5 show that, on the real robot, the policy is frequently able to grasp and lift the spray can off the table: the lift success rate is high, confirming that the first phase of the task transfers reasonably well from simulation

Metric	Value (Real robot)
Lift success rate [%]	90.0
Lift episode length [steps]	3557.2 ± 2388.73
Placement success rate [%]	0.0
RMSE_{pos} $[m]$	0.1335 ± 0.0503
Placement episode length [steps]	8290 ± 2245

Table 6.5: Performance of the `Lift` policy (seed 42) on the real UR10e. The table reports both the lift success rate and the stricter placement success rate, together with the final object-target RMSE and episode length.

to hardware. In contrast, the strict placement success rate drops to 0%. This does not mean that the policy fails to manipulate the object; rather, it indicates that none of the real-robot episodes satisfies the demanding requirement of keeping the object within a tight tolerance around the target pose for a full 50-step window.

When the initial grasp is shallow or slightly misaligned the policy reacts by using the table as a support: the object is gently pushed against the surface while the gripper closes further, so that the grasp gradually evolves from a nearly parallel pinch to a more wrapped configuration around the can. This emergent behaviour was not explicitly programmed, yet it contributes to the high lift success rate observed on the real robot.

This strategy also explains the statistics reported in Table 6.5 for the *lift-only* metric. While the lift success rate reaches 90%, the corresponding lift episode length has a very large variance (3557.2 ± 2388.7 steps). In episodes where the initial contact already yields a sufficiently deep and well-aligned grasp, the can is lifted shortly after touchdown and the episode terminates relatively quickly. In other trials, however, the policy spends a substantial number of control steps performing small in-plane motions and using the table as a support to reconfigure the grasp before committing to the vertical lift. As a result, the distribution of lift episode lengths develops a long right tail: the policy eventually succeeds in lifting the can in most runs, but it sometimes delays the lift phase in order to obtain a more secure, wrapped grasp. This behaviour is beneficial for robustness, yet it also reveals a trade-off between grasp stability and execution efficiency that is not present to the same extent in the idealised simulation setting.

6.5 Sim-to-real comparison

In this section we compare the performance obtained in simulation and on the real UR10e. In both cases we focus on the policy trained with random seed 42,

which was selected for deployment as discussed above. The goal is not to provide an exhaustive statistical study, but rather to highlight how the main performance indicators change when moving from Isaac Lab to the physical robot.

6.5.1 Reach

The `Reach` task is the simplest of the three skills considered in this work, and it is also the one for which the behaviour in simulation and on the real robot is most closely aligned. Table 6.1 reports the aggregate simulation performance averaged over five random seeds, while Table 6.4 summarises the results obtained on the physical UR10e. For a more direct comparison, Table 6.6 juxtaposes the main metrics for the single deployed policy in simulation and on the real robot.

Metric	**Simulation**	**Real robot**
Success rate [%]	98.2	100.0
RMSE_{pos} $[m]$	0.0107 ± 0.003	0.0174 ± 0.003
RMSE_{ori} [°]	11.547 ± 5.99	16.035 ± 0.7476
Episode length [steps]	307.4 ± 42.9	2116.2 ± 190.4321

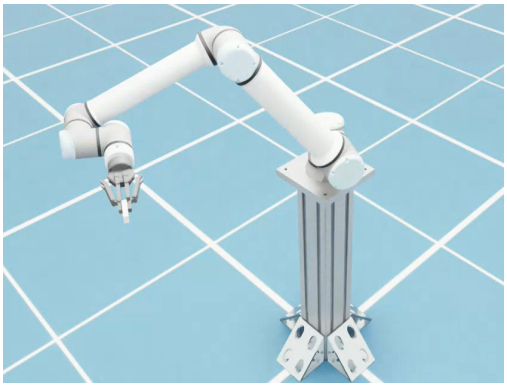
Table 6.6: Sim-to-real comparison for the `Reach` task, reported for the policy trained with seed 42.

Overall, the success rates in simulation and on the real robot are comparable: the deployed policy achieves 98.2% success in Isaac Lab and 100% success over ten trials on the real UR10e. Given the much smaller number of real-robot episodes, these figures should be interpreted as compatible rather than as evidence of genuinely better performance on the hardware.

In terms of geometric accuracy, the position RMSE over the last 50 control steps increases from approximately 1.1 cm in simulation to about 1.7 cm on the real robot, and also the orientation RMSE increases from roughly 11.5° to 16°. These deviations are consistent with the presence of additional sources of error in the real setup. Importantly, the errors remain small in absolute terms, and the qualitative behaviour of the policy–a smooth approach to the target pose and a stable final configuration–is preserved when moving from simulation to the physical UR10e.

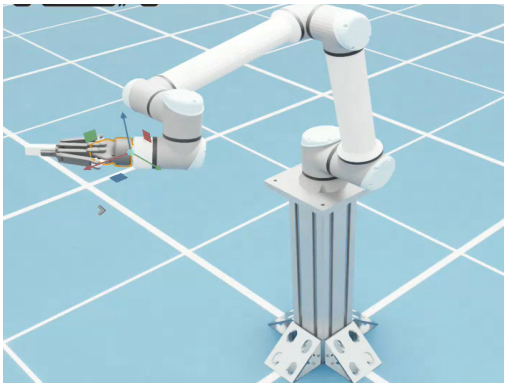
From a control perspective, these results indicate that for the `Reach` task the simulation-to-real transfer is largely successful: the policy trained in Isaac Lab produces reliable and accurate motions on the hardware without any additional fine-tuning. At the same time, the small but systematic increase in the final RMSE highlights that even for relatively simple set-point regulation tasks, residual model mismatch is still observable when comparing simulated and real executions.

Figure 6.5 provides a qualitative confirmation of this observation: for five representative targets, the final TCP poses in simulation (left) and on the real UR10e (right) are visually almost indistinguishable, which is consistent with the centimetre-level position RMSE reported in Table 6.6.



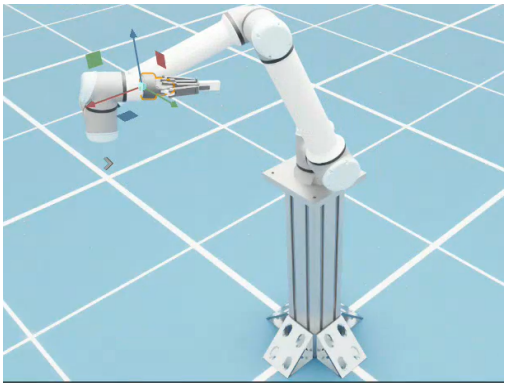
(a) Simulation - target 1

(b) Real robot - target 1

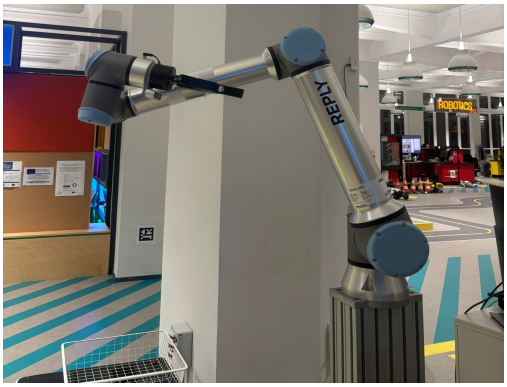


(c) Simulation - target 2

(d) Real robot - target 2

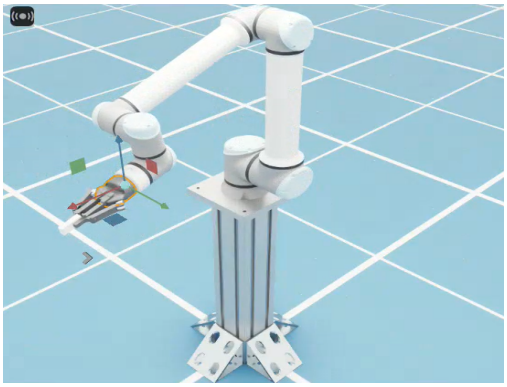


(e) Simulation - target 3



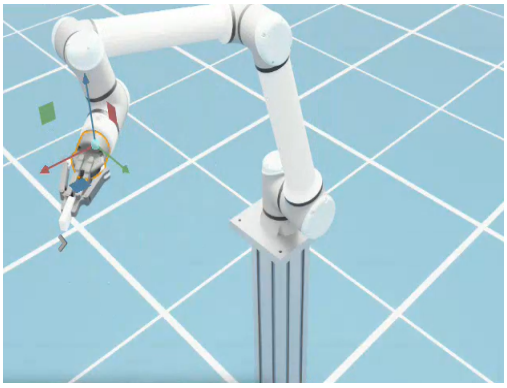
(f) Real robot - target 3

Figure 6.5: Visual comparison between simulation and real robot for targets 1, 2 and 3 in the `Reach` task. Each row shows the final TCP pose for the same target in simulation (left) and on the real UR10e (right), illustrating the close agreement between the two domains.

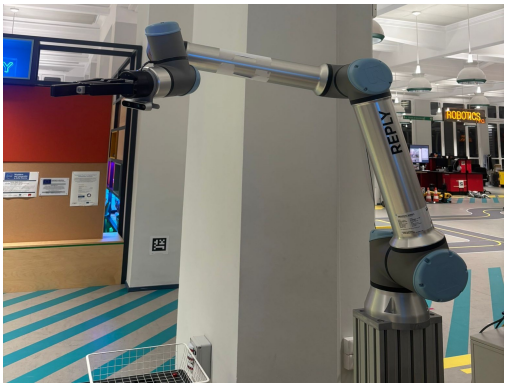


(a) Simulation - target 4

(b) Real robot - target 4



(c) Simulation - target 5



(d) Real robot - target 5

Figure 6.6: Visual comparison between simulation and real robot for targets 4 and 5 in the `Reach` task. Each row shows the final TCP pose for the same target in simulation (left) and on the real UR10e (right).

6.5.2 Lift task

Table 6.7 highlights a clear gap between simulated and real-robot performance for the `Lift` task, especially when using the strict placement-based success definition. In addition to the global success rates and RMSE_{pos}, we further analyse the real-robot behaviour through two auxiliary distance-based metrics described at the end of this section.

In Isaac Lab, the policy trained with seed 42 achieves a lift success rate of 87% and a placement success rate of 81.2%, with a final object-target RMSE_{pos} of $0.0236 \pm 0.0047\,m$. This corresponds to positioning errors of about 2-3 cm and confirms that most simulated episodes not only complete the lift phase but also stabilise the object close to the desired pose for a prolonged time window.

On the real UR10e, by contrast, the same policy attains a slightly higher lift success rate of 90%, confirming that the basic ability to grasp and lift the spray can transfers reasonably well to hardware. However, none of the ten real-robot episodes satisfies the more demanding placement criterion, leading to a placement success rate of 0%. The corresponding object-target $\mathrm{RMSE}_{\mathrm{pos}}$ is $0.1335 \pm 0.0503\,m$: this error is significantly larger than in simulation (roughly a factor of five), but it is still moderate in absolute terms, and many rollouts end with the object within a distance on the order of the object size from the target.

On the real system, the control sequence is executed under encoder noise, slight kinematic inaccuracies and unmodelled flexibilities, so that the object trajectory drifts more than in simulation and rarely remains inside the prescribed spatial window long enough to be counted as a strict success.

Overall, the `Lift` experiments confirm that the proposed pipeline is able to carry over the structure of the behaviour learned in Isaac Lab to the physical UR10e, especially for the lift phase, but also reveal that precise object placement remains strongly affected by modelling inaccuracies in contact interactions and by calibration errors. Addressing these limitations for instance by improving friction and contact modelling, better matching the real object properties in the training environment, or explicitly randomising target poses and calibration offsets is a promising direction for narrowing the sim-to-real gap in future work.

Metric	**Simulation**	**Real robot**
Lift success rate [%]	87.0	90.0
Lift episode length [steps]	65.2 ± 5.4	3557.2 ± 2388.73
Placement success rate [%]	81.2	0.0
$\mathrm{RMSE}_{\mathrm{pos}}$ $[m]$	0.0236 ± 0.0047	0.1335 ± 0.0503
Placement episode length [steps]	85.1 ± 19.4	8290 ± 2245

Table 6.7: Sim-to-real comparison for the `Lift` task, reported for the policy trained with seed 42. The table reports both lift and strict placement success rates; RMSE and episode lengths are averaged over all episodes.

Beyond the global metrics in Table 6.7, we introduce two auxiliary distance-based quantities that are computed only for the ten real-robot episodes in order to better characterise the failure modes observed on hardware. Specifically, for each trial we measure the minimum end effector-object distance $d_{\min}^{\mathrm{EE,obj}}$ and the minimum object-target distance $d_{\min}^{\mathrm{obj,target}}$ over the episode. Their mean and standard deviation across the real-robot rollouts are reported in Table 6.8, providing a compact summary of the best grasping and placement accuracy that the policy manages to achieve in the physical setup.

Metric	**Real robot**
$d_{\text{min}}^{\text{EE,obj}}$ [m]	0.0812 ± 0.0192
$d_{\text{min}}^{\text{obj,target}}$ [m]	0.1686 ± 0.0537

Table 6.8: Auxiliary distance-based metrics for the `Lift` task on the real UR10e. For each episode, we record the minimum end effector-object distance $d_{\text{min}}^{\text{EE,obj}}$ and the minimum object-target distance $d_{\text{min}}^{\text{obj,target}}$; the table reports their mean and standard deviation across trials.

Chapter 7

Conclusions

This thesis investigated the use of deep reinforcement learning for robotic manipulation on an industrial collaborative arm. In particular, it studied how policies trained purely in simulation can be transferred to a real UR10e robot equipped with a Robotiq 2F-140 gripper. Building on the NVIDIA Isaac ecosystem, ROS 2, and the URSim virtual controller. The work implemented an end-to-end pipeline that goes from task definition and training in Isaac Lab/Sim, through simulation-based validation, to deployment on physical hardware.

The project was organised around three manipulation tasks of increasing complexity–Reach, Lift, and OpenDrawer–and around a common experimental protocol based on multiple random seeds and shared performance metrics. The goal was to show that zero-shot sim-to-real transfer is possible in this setting, and to identify where performance degrades when moving from an idealised simulator to a real robot executing contact-rich motions.

7.1 Summary of contributions

The **first** contribution of this thesis is the design and implementation of a complete training-to-deployment workflow for UR-series manipulators. On the simulation side, the manipulation tasks were implemented in Isaac Lab as modular manager-based environments. On the control side, the same joint-space commands produced by the policy are used consistently across Isaac Lab, URSim, and the physical UR10e via a shared ROS 2 interface and compatible controller configurations. This design makes it possible to (i) train and debug policies where full task state and reward are available, (ii) test the deployment code against a virtual UR controller without risk to hardware, and (iii) replay the resulting trajectories on the real robot with only minor adjustments for gripper integration and real-time execution.

A **second** contribution is the systematic use and adaptation of three classes of

manipulation tasks with progressively richer structure. The Reach task isolates precise, closed-loop positioning of the TCP at a target pose; the Lift task extends this to grasping, lifting, and placing a movable object; the OpenDrawer task introduces drawer-handle interactions and constrained motion of an articulated object. These tasks build upon the manipulation environments and reward structures provided by NVIDIA Isaac Lab [41], which are instantiated, tuned, and documented in this thesis for the specific UR10e + Robotiq setup, the chosen domain-randomization ranges, and the sim-to-real deployment pipeline. In this sense, the work offers a reusable template for future manipulation skills on the same platform.

Third, the thesis provides a quantitative characterisation of training dynamics and deployment performance across tasks and domains. In simulation, the Reach policy attains high success rates and centimetre-level accuracy with limited variability across seeds, indicating that the task is reliably solved within the adopted training regime. The Lift policy also reaches high success rates for both lift and placement in simulation, but this level of performance does not transfer to the real robot, where only the lift phase is consistently successful. OpenDrawer, which combines reaching, grasping, and drawer opening, is learned to a reasonable performance level but shows sensitivity to the choice of random seed and a broader spread in success rates.

On the deployment side, the thesis shows that policies trained in Isaac Lab can be replayed in URSim and on the real UR10e without any retraining or explicit domain adaptation. For the Reach task, a representative policy (seed 42) transfers almost directly: real-robot executions exhibit smooth, repeatable motions and final TCP poses that closely match their simulated counterparts, with only a modest increase in position error. For the Lift task, the same policy reliably approaches and grasps a spray can and initiates the lift phase on the real robot, reproducing the qualitative structure of the simulated behaviour.

7.2 Discussion and lessons learned

The experimental results highlight both the potential and the current limitations of deep RL for robotic manipulation in an industrially relevant setting.

On the positive side, the thesis confirms that a simulator-centric, dense-reward workflow, combined with a standard on-policy algorithm such as PPO, can produce policies that transfer meaningfully to a real collaborative manipulator. For the Reach task, the sim-to-real gap is small: a policy trained entirely in Isaac Lab, with no access to real-world data, can drive the UR10e to the desired set-points with centimetre-level accuracy and smooth trajectories, using the same ROS 2 interface and controller stack employed in simulation. This suggests that, when the task is dominated by geometric accuracy and moderate contact forces, careful modelling,

gain tuning, and domain randomization are sufficient to obtain robust closed-loop behaviours.

The Lift experiments extend this picture to a more challenging contact-rich task. The *structure* of the learned behaviour carries over to hardware: the policy approaches the object, closes the gripper, and, in most trials, succeeds in lifting the spray can off the table. Real-robot rollouts also reveal emergent strategies, such as using the table as a support surface to reconfigure a shallow or misaligned grasp into a more secure one. This behaviour was not programmed explicitly but arises from the reward, the policy's stochastic exploration, and the closed-loop nature of the controller, illustrating how RL can discover contact-exploiting behaviours that would be difficult to encode by hand.

At the same time, the Lift results show that precise object placement remains strongly affected by modelling inaccuracies. Even when a simulated policy appears robust according to standard metrics, performance on the real robot deteriorates once the task depends critically on the details of contact interactions, friction coefficients, inertial properties, and exact table and target heights. In the present experiments, these discrepancies appear as larger final RMSE between the object and its target and as failures to satisfy strict success definitions that require the object to remain within a tight tolerance window for an extended time. This is not a failure of the RL algorithm as such, but a reminder that sim-to-real transfer is limited by how well the training environment matches the physical system.

Finally, the work underlines the value of intermediate validation stages. By inserting URSim as a controller-level that mirror the real one on UR10e, the thesis decouples task learning from deployment engineering. Policies can be trained and debugged where full state and reward information are available, while the deployment code, ROS 2 nodes, and controller parameters can be exercised extensively in a risk-free environment before any real-robot experiments.

7.3 Future work

The work presented in this thesis can be extended along a few concrete directions that are closely aligned with the current implementation.

A **first** natural step is to integrate an external camera into the pipeline. All experiments in this thesis rely on state information provided directly by the simulator or by the UR controller. Adding a calibrated RGB-D or monocular camera, together with a pose estimation module, would make it possible to close the loop on the object pose and to operate in less structured scenarios.

A **second** direction is to coordinate multiple policies to solve more complex, multi-stage tasks. In this thesis, Reach, Lift, and OpenDrawer are trained and executed as separate skills. Future work could combine them into a larger task

pipeline by using hierarchical reinforcement learning, where a high-level policy selects which low-level skill to execute, or by orchestrating existing skills through a ROS 2 action server. This would enable, for example, full pick-and-place or drawer-opening sequences composed of several learned behaviours, while keeping each individual policy relatively simple.

A **third** line of work concerns how the policies are trained. The current approach relies entirely on hand-crafted dense rewards, which are effective but time-consuming to design and tune. Imitation Learning techniques, such as Behaviour Cloning from human demonstrations, could be used to initialise the policy and speed up convergence, especially in contact-rich phases like grasping. In parallel, Inverse Reinforcement Learning or related reward-learning methods could help reduce the manual effort of specifying rewards, by inferring them from expert rollouts instead of engineering them term by term.

Overall, these extensions–better sensing, coordinated skills, and learning from demonstrations and inferred rewards–are direct continuations of the present work and have the potential to make the proposed pipeline more robust, more scalable, and closer to real industrial applications.

Appendix A

Hardware and Software Setup

This appendix reports the hardware and software configuration used to carry out the thesis project, in order to support the reproducibility of the results. The information is split between the host machine and the container environment (where Isaac Sim, Isaac Lab are installed and executed).

Host Machine

- **CPU**: AMD Ryzen 9 9950X (16 cores / 32 threads; min frequency 600 MHz, max 5756 MHz; boost enabled).
- **RAM**: 62 GiB (about 55 GiB available at the time of measurement).
- **GPU**: NVIDIA RTX A6000 (49 140 MiB VRAM).
- **Driver/CUDA**: NVIDIA driver `570.172.08`, CUDA runtime `12.8`.
- **Storage**: 1.8 TB NVMe SSD (root, ext4); 3.6 TB HDD (mounted at `/mnt/dati`, ext4).
- **OS**: Ubuntu 24.04.3 LTS (`noble`); kernel `6.14.0-29-generic`.
- **Containerisation**: Docker Engine `28.4.0`; NVIDIA Container Toolkit `1.18.0-rc.3`

Container Environment

All libraries and frameworks used for simulation, training and deployment are installed and executed inside a Docker container, with GPU pass-through enabled.

Languages and package managers

- Python `3.11.13`
- Pip `25.2`
- Conda `25.5.1` (solver `libmamba`)

Deep learning and numerical libraries

- PyTorch `2.7.0+cu128` (CUDA `12.8`, cuDNN `9.7.1`)
- NumPy `1.26.0`
- SciPy `1.15.3`
- scikit-learn `1.7.1`

Simulation and robotics

- Isaac Sim `5.0.0.0` (main modules: `isaacsim-app`, `isaacsim-core`, `isaacsim-rl`, `isaacsim-ros2`, etc.)
- Isaac Lab `2.2.0`
- ROS 2 Humble (packages `ros-humble-desktop`, `ros-humble-moveit`, UR and Robotiq drivers)

Reinforcement learning frameworks

- RSL-RL `2.3.3`
- Stable-Baselines3 `2.7.0`
- RL-Games `1.6.1`
- skrl `1.4.3`
- Gym `0.23.1`, Gymnasium `1.2.0`

Supporting tools

- TensorBoard `2.20.0` (training monitoring)
- Git `2.34.1` (version control)
- Visual Studio Code with the `Remote - Containers` extension
- Omniverse Kit SDK `107.3.1.206797`

Appendix B

Angle/velocity unit conversions

When values are entered in degrees on the USD side but radians are used in Python, use the following:

$$K_{p,^\circ} = K_{p,\mathrm{rad}} \frac{\pi}{180}, \qquad K_{d,^\circ/\mathrm{s}} = K_{d,\mathrm{rad/s}} \frac{\pi}{180}, \qquad \dot{q}_{^\circ/\mathrm{s}} = \dot{q}_{\mathrm{rad/s}} \frac{180}{\pi}.$$

Conversely:

$$K_{p,\mathrm{rad}} = K_{p,^\circ} \frac{180}{\pi}, \qquad K_{d,\mathrm{rad/s}} = K_{d,^\circ/\mathrm{s}} \frac{180}{\pi}, \qquad \dot{q}_{\mathrm{rad/s}} = \dot{q}_{^\circ/\mathrm{s}} \frac{\pi}{180}.$$

Appendix C

UR10e Technical Details

UR10e technical details

Performance

Power consumption	Approx. 350 W using a typical program
Safety System	All 17 advanced adjustable safety functions incl. elbow monitoring certified to Cat.3, PL d. Remote Control according to ISO 10218
Certifications by TUV Nord	EN ISO 13849-1, Cat.3, PL d, and full EN ISO 10218-1
F/T Sensor - Force, x-y-z	
Range	100 N
Resolution	2.0 N
Accuracy	5.5 N
F/T Sensor - Torque, x-y-z	
Range	10 Nm
Resolution	0.02 Nm
Accuracy	0.60 Nm

Specification

Payload	10 kg / 22 lbs
Reach	1300 mm / 51.2 in
Degrees of freedom	6 rotating joints DOF
Programming	Polyscope graphical user interface on 12 inch touchscreen with mounting

Movement

Pose Repeatability	+/- 0.05 mm, with payload, per ISO 9283	
Axis movement robot arm	Working range	Maximum speed
Base	± 360°	± 120°/s
Shoulder	± 360°	± 120°/s
Elbow	± 360°	± 180°/s
Wrist 1	± 360°	± 180°/s
Wrist 2	± 360°	± 180°/s
Wrist 3	± 360°	± 180°/s
Typical TCP speed		1 m/s / 39.4 in/s

Features

IP classification	IP54	
ISO Class Cleanroom	5	
Noise	Less than 65 dB(A)	
Robot mounting	Any Orientation	
I/O ports	Digital in	2
	Digital out	2
	Analog in	2
	Tool communication	RS-485
I/O power supply in tool	12V/24V 600mA continuous, 2A peak	
Ambient temperature range	0-50°C*	
Humidity	90%RH (non-condensing)	

Physical

Footprint	Ø 190 mm
Materials	Aluminium, Plastic, Steel
Tool (end-effector) connector type	M8 | M8 8-pin
Cable length robot arm	6 m / 236 in
Weight including cable	33.5 kg / 73.9 lbs

* The robot can work in a temperature range of 0-50°C. At high continuous joint speeds the maximum allowed ambient temperature is reduced.

Control box

Features

IP classification	IP44
ISO Class Cleanroom	6
Ambient temperature range	0-50°
I/O ports	Digital in 16 Digital out 16 Analog in 2 Analog out 2 500 Hz control, 4 separated high speed quadrature digital inputs
I/O power supply	24V 2A
Communication	Control frequency: 500 Hz ModbusTCP: 500 Hz signal frequency ProfiNet and EthernetIP: 500 Hz signal frequency USB ports: 1 USB 2.0, 1 USB 3.0
Power source	100-240VAC, 47-440Hz
Humidity	90%RH (non-condensing)

Physical

Control box size (WxHxD)	475 mm x 423 mm x 268 mm 18.7 in x 16.7 in x 10.6 in
Weight	Max 13.6 kg / 30.0 lbs
Materials	Steel

Teach pendant

Features

IP classification	IP54
Humidity	90%RH (non-condensing)
Display resolution	1280 x 800 pixels

Physical

Materials	Plastic
Weight including 1 m of TP cable	1.6 kg / 3.5 lbs
Cable length	4.5 m / 177.17 in

UNIVERSAL ROBOTS

universal-robots.com

Figure C.1: UR10e technical specifications [39].

Appendix D

Robotiq 2F-140 Gripper Technical Details

This appendix reports a mechanical drawing of the Robotiq 2F-140 gripper mounted on the UR10e end-effector.

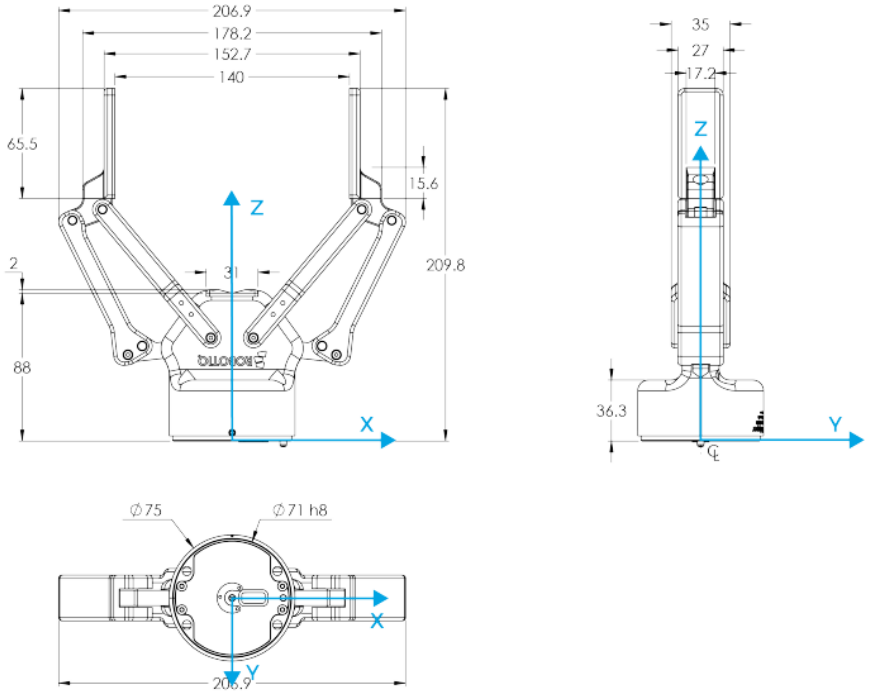


Figure D.1: Robotiq 2F-140 technical specifications [40].

Appendix E

Nonlinear activation functions

This appendix reports the non-linear activation functions used in the policy and value networks, together with the nonlinearity employed in some of the reward shaping terms.

Exponential Linear Unit (ELU)

The hidden layers of both actor and critic use the Exponential Linear Unit (ELU) activation, defined as

$$\mathrm{ELU}_\alpha(x) = \begin{cases} x, & x > 0, \\ \alpha\big(e^x - 1\big), & x \le 0, \end{cases} \tag{E.1}$$

For positive inputs, ELU behaves like the identity function, while for negative inputs it smoothly saturates towards the finite asymptote $-\alpha$. In all experiments, the same ELU with $\alpha = 1.0$ is used in every hidden layer of both the actor and the critic networks, and this choice is kept fixed across all tasks. Compared to ReLU, ELU reduces the risk of "dead" neurons and provides negative outputs with non-zero mean, which can help stabilise optimisation in the PPO-based training setup[33].

Figure E.1 illustrates the shape of the ELU activation over a representative input range.

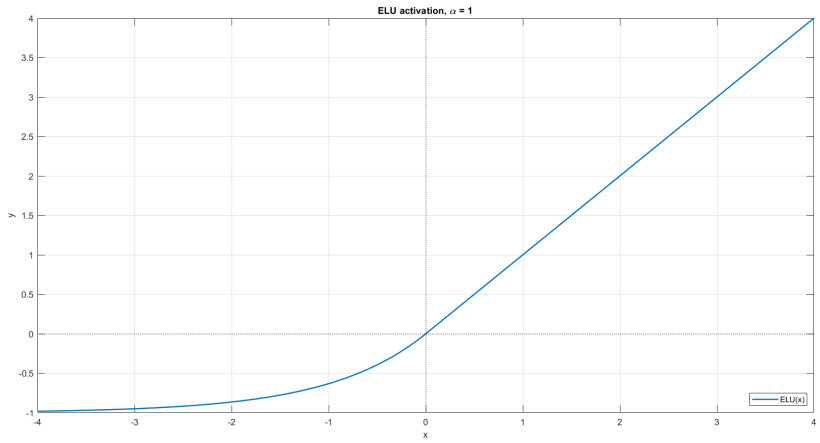


Figure E.1: Exponential Linear Unit (ELU) activation function

Hyperbolic tangent

The hyperbolic tangent function is defined as

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{E.2}$$

and maps all the inputs to the open interval $(-1, 1)$. It assumes values $\tanh(x) \approx -1$ for large negative inputs and $\tanh(x) \approx 1$ for large positive inputs.

In this work, the hyperbolic tangent is used as a bounded kernel in some reward shaping terms (for example, in the fine position tracking reward described in Section 4.2.1), where expressions of the form $1 - \tanh(d_t/\sigma)$ are employed to turn a distance $d_t \geq 0$ into a reward that is close to 1 when the error is small and smoothly decays towards 0 as the error increases.

Figure E.2 illustrates how input scaling affects the shape of the tanh nonlinearity. The three panels show $y = \tanh(x)$, $y = \tanh(x/2)$ and $y = \tanh(2x)$, respectively. Dividing the argument by 2 stretches the function horizontally, so saturation is reached more slowly and a wider range of inputs is mapped to intermediate values. Conversely, multiplying the argument by 2 compresses the function horizontally, making the transition around the origin steeper and causing the function to saturate more quickly. These plots provide an intuitive picture of the different scalings with which the hyperbolic tangent is used in the reward functions, and help the reader understand how changing the argument affects the resulting reward landscape.

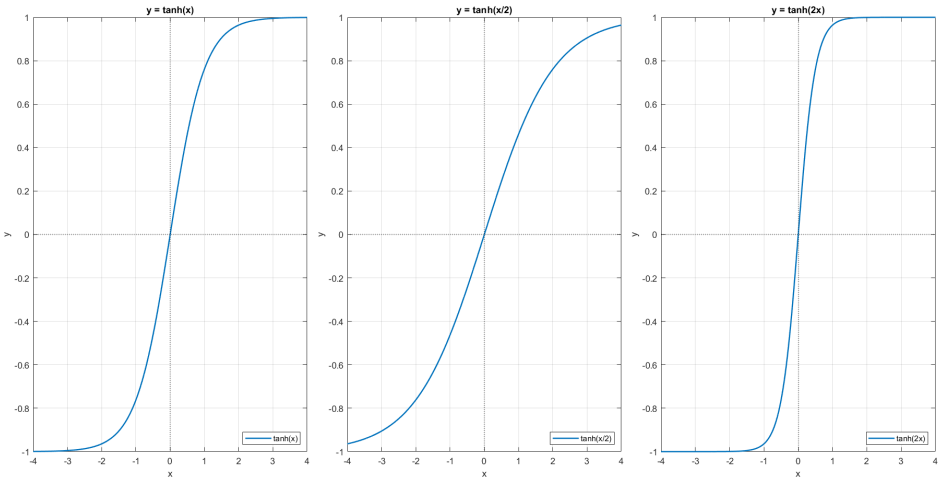


Figure E.2: Hyperbolic tangent activation function for three different input scalings

Bibliography

[1] Mehran Ghafarian Tamizi, Marjan Yaghoubi, Homayoun Najjaran, Behnam M. Tehrani, Samer BuHamdan, and Aladdin Alwisy. «A review of recent trend in motion planning of industrial robots». In: *International Journal of Intelligent Robotics and Applications* 7.1 (2023), pp. 556–574. DOI: 10.1007/s41315-023-00274-2 (cit. on p. 1).

[2] David Coleman, Ioan Şucan, Sachin Chitta, and Nikolaus Correll. «Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study». In: *arXiv preprint arXiv:1404.3785* (2014). URL: https://arxiv.org/abs/1404.3785 (cit. on p. 1).

[3] Steven M. LaValle. *Planning Algorithms*. Cambridge, UK: Cambridge University Press, 2006. URL: http://planning.cs.uiuc.edu/ (cit. on p. 1).

[4] Lydia E. Kavraki, Petr Švestka, Jean-Claude Latombe, and Mark H. Overmars. «Probabilistic roadmaps for path planning in high-dimensional configuration spaces». In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439 (cit. on p. 1).

[5] James J. Kuffner and Steven M. LaValle. «RRT-Connect: An Efficient Approach to Single-Query Path Planning». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2000, pp. 995–1001. DOI: 10.1109/ROBOT.2000.844730 (cit. on p. 1).

[6] Devendra Rawat, Mukul Kumar Gupta, and Abhinav Sharma. «Intelligent control of robotic manipulators: a comprehensive review». In: *Spatial Information Research* 31 (2023), pp. 345–357. DOI: 10.1007/s41324-022-00500-2 (cit. on p. 1).

[7] Y. Liu, H. J. Yap, U. Khairuddin, et al. «Motion planning of robotic manipulators in dynamic environments: a comprehensive review». In: *Journal of Sensors* (2024), p. 5969512. DOI: 10.1155/2024/5969512 (cit. on p. 1).

[8] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. «Integrated Task and Motion Planning: A Survey». In: *Annual Review of Control, Robotics, and Autonomous Systems* 4 (2021), pp. 265–293. DOI: 10.1146/annurev-control-091420-084139 (cit. on p. 1).

[9] NVIDIA Corporation. *NVIDIA Isaac Sim: A Scalable Robotics Simulation Application*. Online; accessed 2025. 2024. URL: https://developer.nvidia.com/isaac-sim (cit. on pp. 1, 20).

[10] NVIDIA Corporation. *Isaac Lab: A GPU-Accelerated Robot Learning Framework*. Online; accessed 2025. 2024. URL: https://developer.nvidia.com/isaac-lab (cit. on pp. 1, 20).

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347 (cit. on pp. 1, 11, 14, 16).

[12] Bingjie Tang, Michael A. Lin, Iretiayo Akinola, Ankur Handa, Gaurav S. Sukhatme, Fabio Ramos, Dieter Fox, and Yashraj Narang. «IndustReal: Transferring Contact-Rich Assembly Tasks from Simulation to Reality». In: *arXiv preprint arXiv:2305.17110* (2023). arXiv: 2305.17110 [cs.RO] (cit. on p. 2).

[13] Haozhi Qi, Brent Yi, Sudharshan Suresh, Mike Lambeta, Yi Ma, Roberto Calandra, and Jitendra Malik. «General In-Hand Object Rotation with Vision and Touch». In: *arXiv preprint arXiv:2309.09979* (2023). arXiv: 2309.09979 [cs.RO] (cit. on p. 2).

[14] Tao Chen, Megha Tippur, Siyang Wu, Vikash Kumar, Edward Adelson, and Pulkit Agrawal. «Visual Dexterity: In-Hand Reorientation of Novel and Complex Object Shapes». In: *arXiv preprint arXiv:2211.11744* (2023). arXiv: 2211.11744 [cs.RO] (cit. on p. 2).

[15] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. «End-to-end training of deep visuomotor policies». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1488–1495. DOI: 10.1109/ICRA.2016.7487342 (cit. on p. 2).

[16] Dmitry Kalashnikov et al. «QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation». In: *Proceedings of the Conference on Robot Learning (CoRL)*. 2018, pp. 651–673. URL: https://arxiv.org/abs/1806.10293 (cit. on p. 2).

[17] Chen Tang, Ben Abbatematteo, Jiaheng Hu, Rohan Chandra, Roberto Martín-Martín, and Peter Stone. «Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Survey on real-world DRL for robotics. 2025. URL: https://arxiv.org/abs/2410.19414 (cit. on pp. 2, 4, 11, 15, 16).

[18] Jens Kober, J. Andrew Bagnell, and Jan Peters. «Reinforcement Learning in Robotics: A Survey». In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721 (cit. on pp. 2, 11).

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. Cambridge, MA: MIT Press, 2018. ISBN: 978-0262039246 (cit. on p. 9).

[20] Md. Al-Masrur Khan, Md Rashed Jaowad Khan, Abul Tooshil, Niloy Sikder, M. A. Parvez Mahmud, Abbas Z. Kouzani, and Abdullah Al Nahid. «A Systematic Review on Reinforcement Learning-Based Robotics Within the Last Decade». In: *IEEE Access* 8 (2020), pp. 176598–176623. DOI: 10.1109/ACCESS.2020.3027152. URL: https://doi.org/10.1109/ACCESS.2020.3027152 (cit. on p. 10).

[21] Hyun-Kyo Lim, Ju-Bong Kim, Joo-Seong Heo, and Youn-Hee Han. «Federated Reinforcement Learning for Training Control Policies on Multiple IoT Devices». In: *Sensors* 20.5 (2020), p. 1359. ISSN: 1424-8220. DOI: 10.3390/s20051359. URL: https://doi.org/10.3390/s20051359 (cit. on p. 12).

[22] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. «High-Dimensional Continuous Control Using Generalized Advantage Estimation». In: (2016). URL: https://arxiv.org/abs/1506.02438 (cit. on p. 13).

[23] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. «Trust Region Policy Optimization». In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. 2015, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html (cit. on p. 13).

[24] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer (cit. on p. 18).

[25] Universal Robots. *DH parameters for calculations of kinematics and dynamics*. Universal Robots. URL: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/ (visited on 10/30/2025) (cit. on pp. 18, 19).

[26] Isaac Lab Project Developers. *Reference Architecture — Isaac Lab Documentation*. Accessed: 2025-11-21. 2025. URL: https://isaac-sim.github.io/IsaacLab/main/source/refs/reference_architecture/index.html (cit. on p. 21).

[27] NVIDIA Corporation. *NVIDIA PhysX SDK*. Release date: 2019-03-08. 2019. URL: https://developer.nvidia.com/physx-sdk (visited on 11/18/2025) (cit. on pp. 20, 57).

[28] NVIDIA Corporation. *OpenUSD — Universal Scene Description by NVIDIA*. Accessed: October 2025. 2025. URL: https://www.nvidia.com/it-it/omniverse/usd/ (cit. on p. 21).

[29] NVIDIA Corporation. *Isaac Sim 5.0 — Assembler Tool Tutorial*. Accessed: October 2025. 2025. URL: https://docs.isaacsim.omniverse.nvidia.com/5.0.0/robot_setup_tutorials/tutorial_import_assemble_manipulator.html (cit. on p. 22).

[30] Clemens Schwarke, Mayank Mittal, Nikita Rudin, David Hoeller, and Marco Hutter. *RSL-RL: A Learning Library for Robotics Research*. 2025. arXiv: 2509.10771 [cs.RO]. URL: https://arxiv.org/abs/2509.10771 (cit. on p. 23).

[31] Mayank Mittal, Kelly Guo, Gavriel State, Spencer Huang, et al. *Isaac Lab: A GPU-Accelerated Simulation Framework for Multi-Modal Robot Learning*. NVIDIA Research Whitepaper. Whitepaper PDF available at NVIDIA Research; accessed October 31, 2025. 2025. URL: https://research.nvidia.com/publication/2025-09_isaac-lab-gpu-accelerated-simulation-framework-multi-modal-robot-learning (cit. on p. 23).

[32] Viktor Makoviychuk et al. «Isaac Gym: High Performance GPU Based Physics Simulation for Robot Learning». In: *arXiv preprint arXiv:2108.10470* (2021). URL: https://arxiv.org/abs/2108.10470 (cit. on p. 23).

[33] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. «Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)». In: *arXiv preprint arXiv:1511.07289* (2016). URL: https://arxiv.org/abs/1511.07289 (cit. on pp. 24, 89).

[34] NVIDIA Isaac Lab Developers. *isaaclab.managers — Isaac Lab Documentation*. API reference for manager classes in Isaac Lab. 2025. URL: https://isaac-sim.github.io/IsaacLab/main/source/api/lab/isaaclab.managers.html (visited on 11/19/2025) (cit. on p. 25).

[35] NVIDIA Isaac Lab Team. *Isaac Lab: MDP/Manager-Based Environments (API Documentation)*. Accessed Oct. 31, 2025. 2025. URL: https://isaac-sim.github.io/IsaacLab/main/source/api/lab/isaaclab.envs.mdp.html (cit. on p. 25).

[36] Steve Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, Architecture, and Uses in the Wild». In: *arXiv preprint arXiv:2211.07752* (2022). URL: https://arxiv.org/abs/2211.07752 (cit. on p. 30).

[37] Universal Robots, FZI Forschungszentrum Informatik, and Contributors. *Universal Robots ROS 2 Driver*. ROS 2 driver for UR e-Series and CB-Series robots. 2025. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver (visited on 11/09/2025) (cit. on pp. 31, 57, 58).

[38] FZI Forschungszentrum Informatik and Contributors. *Robotiq 2F URCap Adapter*. ROS 2 interface to control Robotiq 2F grippers mounted on UR controllers. 2025. URL: https://github.com/fzi-forschungszentrum-informatik/robotiq_2f_urcap_adapter (visited on 11/09/2025) (cit. on pp. 31, 60, 61).

[39] *UR10e User Manual*. Universal Robots A/S. 2024. URL: https://www.universal-robots.com/products/ur10e (cit. on pp. 31, 87).

[40] *2F-140 Adaptive Gripper – Instruction Manual*. Robotiq Inc. 2023. URL: https://robotiq.com/products/adaptive-grippers#Two-Finger-Gripper (cit. on pp. 32, 88).

[41] Isaac Lab Developers. *Available Environments. Isaac Lab Documentation*. Accessed on 17 November 2025. 2025. URL: https://isaac-sim.github.io/IsaacLab/main/source/overview/environments.html (visited on 11/17/2025) (cit. on pp. 36, 80).

[42] Robotic Systems Lab, ETH Zurich. *RSL-RL: A Learning Library for Robotics Research*. Accessed from the official GitHub repository. 2025. URL: https://github.com/leggedrobotics/rsl_rl (visited on 11/19/2025) (cit. on p. 37).

[43] *Real-time configuration for UR ROS2 driver*. https://docs.universal-robots.com/Universal_Robots_ROS2_Documentation/doc/ur_client_library/doc/real_time.html. Accessed: 2025-11-10 (cit. on p. 60).