Alma Mater Studiorum - Università di Bologna

SCHOOL OF ENGINEERING
DEPARTMENT of
ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING
"Guglielmo Marconi"
DEI
Master Degree in Automation Engineering

Sim2real Transfer for Reinforcement Learning in Robotic Arm Control: a Closed-Loop Optimization approach for Parameter Estimation

Master Degree Thesis
in
Autonomous and Mobile Robotics

Supervisor:
Prof. Gianluca Palli

Co-Supervisors:
Alex Pasquali
Riccardo Zanella

Candidate:
Davide Bargellini

Academic Year
2022/2023
III Session

Abstract

This thesis delves into sim-to-real transfer challenges for torque-controlled robotic arms. Employing reinforcement learning, the research addresses the simulation-reality gap to improve control policies for practical applications. The study investigates static and dynamic friction compensation, as well as the optimization of simulation parameters and domain randomization. The results offer valuable insights into the efficacy of the proposed methodology and its potential implications for real-world robotic systems. Additionally, the thesis introduces an approach to refining simulation parameters by comparing trajectories generated in simulation with those obtained from real-world experiments. This contributes to a more comprehensive understanding of the sim-to-real transfer problem and advances the state-of-the-art in robotic control.

Contents

List of Figures

Chapter 1

Introduction

Robotic systems play a central role in various applications, ranging from industrial automation to healthcare. The capability of these systems to perform effectively in real-world environments is crucial for their widespread adoption. However, transferring control policies learned in simulation to real-world scenarios, known as sim-to-real transfer, remains a challenging problem, particularly for torque-controlled robotic arms.

This thesis addresses the complexities associated with sim-to-real transfer, focusing on the Panda robot, a torque-controllable robotic arm. The motivation for this work originates from the need to bridge the gap between simulation environments, such as MuJoCo, and the real-world counterpart, emphasizing the challenges that come from this process.

1.1 Introduction to Sim-to-Real Transfer

Sim-to-real transfer is a critical frontier in the realm of robotic systems, standing at the crossroads of simulation, machine learning, and real-world deployment. The fundamental challenge lies in the seamless integration of insights gained in a simulated environment into the complexities and uncertainties of the physical world. In the context of torque-controlled robotic arms, this challenge is magnified by the need for precise and adaptable control strategies to navigate real-world scenarios effectively.

Motivated by the potential advantages of sim-to-real transfer, the thesis project seeks to bridge the gap between simulation and reality, in particular for torque-controlled robotic arms with many unknown parameters. The advantages of training in simulation and then transferring the trained model in reality are many. Firstly, the ability to train and optimize control policies in a simulated environment significantly reduces the cost and time associated with real-world experiments. This not only accelerates the development cycle (depending on the hardware, the simulation time could be hundreds of times faster than reality) but also enables the exploration of a wide range of scenarios that may be impractical or dangerous in reality. Secondly, the deployment of robotic systems in diverse and dynamic environments necessitates adaptability. Sim-to-real transfer allows robotic models to adapt to unforeseen variations, ensuring that learned behaviours generalize effectively. Such adaptability is particularly vital for industrial applications, where robots must operate in different environments, for example, a robotic arm should work properly with different weights, workspace limits and varying tasks. Whether handling light or heavy payloads, moving in confined spaces or expansive work areas and performing a range of tasks with precision, the ability of robotic arms to seamlessly adjust to diverse conditions ensures efficiency and reliability in industrial settings.

Our primary goal is to devise methodologies and strategies that enable torque-controlled robotic arms, in our case the Franka Emika Panda robot, to seamlessly transition from simulated training environments to real-world tasks. In our study, we focused on the task of reaching random points in the workspace using the Panda robotic arm. The end effector of the Panda arm was tasked with reaching specified points in the three-dimensional space, requiring the learning algorithms to acquire robust control policies for diverse scenarios. To achieve this, we have used, as a base, MuJoCo, a physics engine that serves as our primary training environment, and Gazebo, the simulation platform that offers a closer-to-reality control interface for simulating the behaviour of robotic systems.

In the subsequent sections, we provide a detailed overview of the Panda robot, the MuJoCo training environment, Gazebo simulation platform and

reinforcement learning. We explore the difficulties encountered when transitioning from simulation to Gazebo, highlighting the problems of adjusting control policies to accommodate varying dynamics. As we explore sim-to-real transfer, we refer to existing literature, introduce new adaptations, and share results that add to the ongoing conversation about the effectiveness of sim-to-real transfer for torque-controlled robotic arms.

1.2 Motivation and Goals

The motivation behind this research comes from the growing demand for robotic systems that can seamlessly transition from simulation to real-world environments. Moreover, our focus lies in addressing the challenge posed by unknown parameters associated with robotic arms, such as the Panda robot's inertia, static friction and dynamic friction compensation. Notably, the dynamic friction compensation remains concealed and inaccessible through the libfranka API and the inertia parameters are not revealed by Franka Robotics, further limiting access to crucial details of the robotic system's dynamics. The primary goals of this project encompass understanding and mitigating the challenges associated with sim-to-real transfer for torque-controlled robotic arms. Specific objectives include:

- Investigating the limitations of current simulation environments in capturing the complexities of real-world physics and dynamics.
- Developing effective strategies for training torque-controlled robotic arms in simulation for reaching specific points, with a focus on reinforcement learning methodologies.
- Exploring the adaptation of simulation-trained models to real-world scenarios, considering factors such as friction, gravity compensation, and unknown parameters.
- Assessing the performance of the adapted models through rigorous experimentation and comparison between simulated and real-world trajectories.

By achieving these goals, we aim to advance the state-of-the-art in sim-to-real transfer for torque-controlled robotic arms.

1.3 Overview of Robotics Components

In this section, we will show the fundamental components that form the backbone of our robotics exploration. We begin by introducing the Panda robot, a versatile and widely used robotic platform, highlighting its key features and functionalities. Subsequently, we explore the simulation environments, MuJoCo and Gazebo, which play crucial roles in our research and development efforts. These environments provide a virtual playground for testing and refining robotic algorithms before deploying them on physical counterparts. Through a comprehensive overview of these robotics components, we aim to lay the groundwork for the subsequent chapters that delve into specific aspects of our research and experiments.

1.3.1 Panda Robot

The Panda robot, developed by Franka Emika, serves as the primary robotic platform for our investigation.

Figure 1.1: Franka Emika Panda [1].

The Franka Emika Panda robot is a 7-axis robot arm designed for research, it enables direct control and is developed for multiple tasks including manipulation and grasping.

The robot is also equipped with force/torque sensors at its joints, enabling it to sense external forces and torques.

1.3.2 ROS

ROS [2] (Robot Operating System) is an open-source operating system for robotic applications. The idea of ROS is to develop a high-level control that can communicate with the lower level of the robot. This high-level controller is an abstraction, meaning that the program written for a robotic platform can be reused eventually with other robots even from different producers if their interface with ROS is provided. The key idea of ROS is the fact that it is a distributed framework of processes that run concurrently, each process, called "node", is meant to work independently of each other and represents a different module. The nodes are capable of sharing information with each other through a communication system based on the concept of "topic" and "messages".

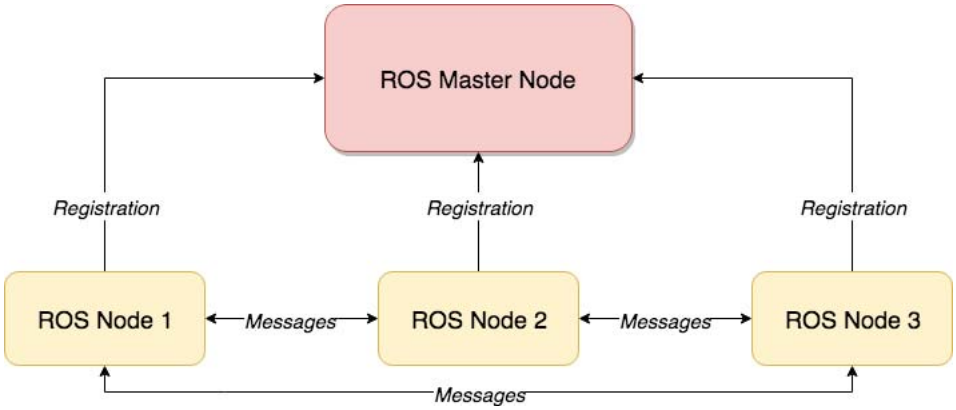


Figure 1.2: ROS communication scheme, nodes are registered to a central master node. During their execution, nodes communicate with each other with messages, services and actions.

Nodes

In ROS each program executed is denoted as a Node, it can be seen as a single entity running concurrently in the system. Each node can communicate with the other nodes by using a server-client communication architecture, in which each node can work as a client but also as a server. To communicate, a node must be connected with a master node, whose purpose is to enable communication between all the nodes registered to it. Nodes can communicate between themselves through the direct invocation of another node's services or actions or through publish/subscribe. A ROS−-based robot control system is usually composed of many nodes. Each node should be designed to have a small and specific task. Nodes should perform their tasks and exchange the results with other nodes. This net of elaboration of data will form a complex graph−like structure capable of solving demanding problems. Node−based architecture provides ROS with many benefits, where the biggest benefits are fault tolerance (as each node is an isolated part of the system) and reduced code complexity compared to monolithic systems, which are not decoupled.

Topics

ROS topics are the "mailboxes" used by nodes to communicate. This communication is based on a publish/subscribe mechanism. Each ROS topic has a unique name associated so that ROS nodes can publish or subscribe to it. Any node is enabled to publish or subscribe to any ROS topic as long it knows the name of the topic. Furthermore, there is no limitation on the number of topics to which a node can be subscribed or publish. As this communication is not direct, nodes are not aware of who are they getting the data from or who are they sending the data to.

Messages

ROS messages are exchanged between ROS nodes using the publish/subscribe mechanism. One ROS node would publish the ROS message to a certain ROS topic, while the other ROS node would subscribe to that ROS topic

and obtain the ROS message sent. ROS Messages are described as .txt files inside a specific folder called msgs under the ROS package folder structure. Each ROS message is described with a data structure which contains only primitive types like integers, floats or booleans, it also can include arrays of the primitive types listed earlier. A ROS message can also include other ROS messages. In this way, it is possible to create complex and longer messages. Additionally, ROS messages can contain the other ROS message or an array of ROS messages as a data type. ROS messages can also be exchanged in a direct communication between nodes. This mechanism is called "Services".

Figure 1.3: Visual concept of the message communication scheme in ROS.

Services

Topics can be seen as named "mailboxes", where a node can publish or read messages. In this context, there are no direct connections between the nodes or knowledge of the topics connected which are reading the published messages. ROS services instead are used when there is a wish for nodes to be able to communicate directly with each other. By using the ROS services, the publish/subscribe mechanism is avoided and nodes can send requests and replies to each other directly using the defined srv. Like the messages, also the services have their target folder named "srv" in which custom services can be defined, each srv is a .txt file containing its two parts, the request sent from the client node, and the response from the Server node. Since the ROS services are a form of direct communication, they are increasing the performance of the system, however, they are decreasing the system decoupling.

1.3.3 Simulation environment

Two main simulation environments are central to our study: MuJoCo and Gazebo. MuJoCo, developed by Google DeepMind, offers a straightforward plug-and-play physics engine. Gazebo, on the other hand, is a versatile open-source robotics simulator that provides a realistic 3D environment for testing and developing robotic systems. Simulation environments play a crucial role in RL-based training for robotic systems. They provide a safe and scalable platform for the agent to explore different scenarios and learn optimal policies.

MuJoCo

MuJoCo [3] (Multi-Joint dynamics with Contact) is an open-source physics engine, providing a simulation environment well-suited for reinforcement learning tasks. Developed by Google DeepMind, MuJoCo offers a physics engine designed to capture the complexities of robotic systems, making it a popular choice for training and evaluating reinforcement learning agents.

MuJoCo excels in simulating the dynamics of multi-joint systems, making it appropriate for torque-controlled robotic arms. The environment supports the modelling of various physical properties, including friction, inertial properties, and joint dynamics. This flexibility enables the creation of diverse and realistic scenarios for training reinforcement learning agents. In MuJoCo, the robot is represented through an XML model, specifying joint properties, links, and physical parameters. The control of the robot is achieved by specifying torques at each joint. The training process in MuJoCo involves the iterative optimization of policies through reinforcement learning algorithms. Our approach utilizes a grid search, exploring a range of hyperparameters to enhance the training effectiveness. More detailed information about the grid search process and specific hyperparameter configurations will be provided in the subsequent chapter.

Gazebo

Gazebo serves as an integral component of our research infrastructure, providing a realistic robotic simulation environment within the Robot Operating System (ROS) framework. Gazebo enables the emulation of diverse robotic platforms and scenarios, facilitating the development and testing of control algorithms before deployment on physical hardware.

Gazebo is tightly integrated with ROS, allowing seamless communication between the simulated environment and the robotic control system. The integration enables ROS nodes to interact with the simulated robot in Gazebo, replicating the real-world interaction between software components and robotic hardware. This integration is fundamental to our sim-to-real transfer approach, as it allows us to utilize ROS functionalities for control and communication.

The Gazebo simulation engine incorporates a robust physics engine that simulates realistic interactions between the robot and its environment. This includes modelling friction, dynamics, and collisions, providing a high-fidelity representation of the robot's behaviour. The realism achieved in Gazebo is crucial for training reinforcement learning agents in scenarios that closely resemble real-world conditions.

In Gazebo, the robot model is typically represented using the Unified Robot Description Format (URDF). This is very similar to MuJoCo, being URDF an XML-based format that defines the robot's kinematic and dynamic properties, including joint specifications, link connections, and physical parameters. The URDF model serves as the blueprint for the robot in the simulated environment, ensuring consistency between simulation and reality.

Gazebo offers versatile tools for simulation control and visualization. Users can control the simulation time, pause and resume scenarios, and inspect the robot's state during runtime. Visualization tools provide insights into the robot's movements, sensor readings, and environmental interactions. These features are crucial in debugging and understanding the behaviour of the simulated system.

1.3.4 Challenges in Sim-to-Real Transfer

The process of transferring knowledge gained in simulation to real-world robotic systems is filled with challenges.

One of the fundamental obstacles lies in the intrinsic differences between simulation and reality. Physical properties, such as friction coefficients, inertial parameters, and joint dynamics, may vary between the simulated environment and the real-world robotic system. These discrepancies can lead to suboptimal or even failed transfer attempts, necessitating methods for adaptation. Robust adaptation strategies, such as domain randomization, are essential for handling these uncertainties.

In real-world robotic systems, compensatory mechanisms might be in place to address specific issues. However, these compensations may not be accurately reflected in the simulation. Additionally, incorrect compensations implemented in the simulation can lead to a mismatch between expected and actual behaviours in the real-world scenario.

Challenges specific to simulation environments, such as discrepancies in simulation and control frequencies, also contribute to the complexity of sim-to-real transfer. Strategies employed during simulation training might not seamlessly translate to real-world scenarios, requiring careful consideration and adaptation.

One of the main contributors to the field of sim-to-real transfer in deep reinforcement learning for robotics is the survey conducted by Zhao, Queralta, and Westerlund [4]. This paper provides a comprehensive review of the different methods being utilized to close the sim-to-real gap and accomplish more efficient policy transfer. It covers the fundamental background behind sim-to-real transfer in deep reinforcement learning and overviews the main methods being utilized at the moment, including domain randomization, domain adaptation, imitation learning, meta-learning, and knowledge distillation.

Meanwhile, the work by Peng et al. [5] significantly contributes to the field by expanding the concept of domain randomization. Their research delves into the development of adaptable control policies for robotic systems,

demonstrating its effectiveness in real-world scenarios. Through experiments, the authors showcase that policies trained exclusively in simulation maintain a similar level of performance when deployed on a real robot. This work, presented at the IEEE International Conference on Robotics and Automation in 2018, provides valuable insights into the sim-to-real transfer of robotic control with dynamics randomization.

Navigating through these challenges is the core of our research, as we aim to develop effective and robust sim-to-real transfer methodologies for torque-controlled robotic arms, with a particular focus on domain randomization to enhance adaptability to real-world variations.

1.3.5 Reinforcement Learning in Robotic Systems

Reinforcement learning (RL) has emerged as a potent approach for training robotic systems to perform complex tasks. In this subsection, we delve into the fundamental concepts of reinforcement learning, its applications in robotic control, and its relevance to sim-to-real transfer.

At its core, reinforcement learning involves an agent interacting with an environment to learn optimal actions that maximize a cumulative reward signal. The agent explores the environment, takes actions, receives feedback in the form of rewards, and adjusts its behaviour over time (see Fig. 1.4). This trial-and-error learning process enables the agent to discover effective strategies for solving tasks.

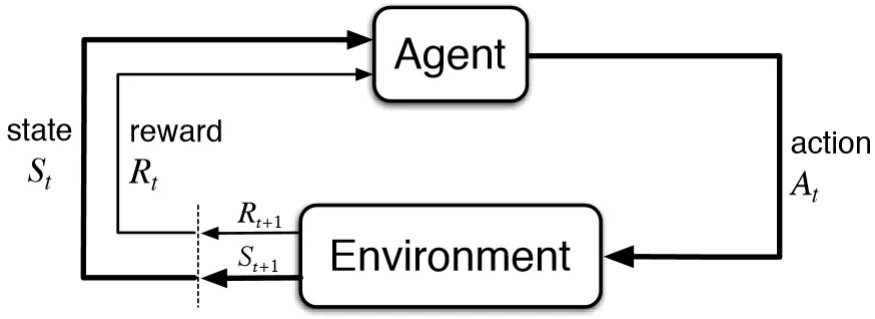


Figure 1.4: Reinforcement Learning simple representation.

In the context of robotic systems, reinforcement learning has been successfully applied to various tasks, ranging from simple manipulator control to complex locomotion and dexterous manipulation. RL algorithms allow robots to adapt and learn behaviours in dynamic and uncertain environments, making them versatile for a wide range of applications.

The foundational work by Kaelbling, Littman, and Moore provides a profound analysis of reinforcement learning and its applications [6]. In their extensive survey, they delve into the fundamental concepts of learning and algorithms, emphasizing crucial aspects such as exploration-exploitation trade-offs and the necessity for sample efficiency when learning from interactions with the environment.

In the realm of robotic arms, Lindner, Milecki, and Wyrwał have conducted a comprehensive study on the application of Reinforcement Learning (RL) algorithms for achieving precise positioning [7]. The article evaluates the performance of four RL algorithms—DDPG, TD3, SAC, and HER—in six combinations, considering critical factors such as positioning accuracy, motion trajectory, and the number of steps required to attain the goal. Their findings offer valuable insights into the effective use of RL for enhancing the precision of robotic arm movements. The results indicate that RL algorithms can be successfully applied for learning the positioning control of a robot arm, showcasing the potential of RL in robot programming and control. The study also highlights the need for further research to identify the strengths and weaknesses of each algorithm for specific tasks. Exploiting reinforcement learning through a systematic grid search methodology, we explore a multitude of hyperparameters and RL algorithms to understand their impact on training effectiveness and generalization.

To further enhance the probability of success of our sim-to-real transfer we have adopted the technique of *domain randomization*, a concept that introduces variability into the training environment. By randomizing parameters such as inertia, friction and goal position during simulation, we aim to create a diverse training dataset that better prepares our model for the real world. However, the effectiveness of domain randomization is dependent on judicious parameter selection, requiring expert knowledge to balance ro-

bustness without obstructing the learning process. In particular, the paper [8] discusses the limitations of existing reality gap mitigation methods like domain randomization, emphasizing the need for prior knowledge.

Meanwhile, Chebotar et al. [9] presents a data-driven strategy using real-world data to enhance simulation randomization. The focus is on learning simulation parameter distributions conducive to successful policy transfer, minimizing the necessity for exact real-world replication. The article delves into closed-loop policy rollouts, comparing them with trajectory-based parameter learning and underlining the potential of physics simulations for parameterized models. Experimental validations demonstrate successful policy transfers for complex robotic tasks, emphasizing the importance of accurate parameter estimation in achieving real-world applicability.

Chapter 2

Reinforcement Learning in Simulation Environments

As we explore the training of torque-controlled robotic arms with reinforcement learning (RL), it's crucial to thoroughly examine the simulation environments we're working with. This chapter takes a close look at the MuJoCo (Multi-Joint dynamics with Contact) simulator and Gazebo, which serve as the training platforms for our Panda robotic arm. We explore the details of RL methodologies in both environments, explaining the training procedures, environmental features, and the challenges encountered.

Using RL in simulation environments allows for iterative refinement of robotic control policies. MuJoCo and Gazebo, each with distinct strengths, provide suitable settings for training agents to master complex motor skills and adapt to dynamic scenarios. Despite successful simulations, the transition to real-world applicability poses a significant challenge. This chapter discusses the preparatory steps taken within MuJoCo and Gazebo to facilitate the subsequent transfer to the real-world counterpart. We will discuss design choices, training configurations, and insights from the RL process. We aim to offer a thorough understanding of the simulated foundation before addressing the challenges and adaptations needed to bridge the gap between simulation and reality.

2.1 MuJoCo Training Environment

Within MuJoCo, our Panda robotic arm becomes an agent, navigating its environment through a defined action space and perceiving the consequences through an observation space. To seamlessly interface with the MuJoCo simulator, we utilize Gymnasium [10], an open-source toolkit designed to develop and compare reinforcement learning algorithms. Gymnasium provides a unified interface for interacting with different simulation environments, streamlining the integration of MuJoCo into our RL pipeline.

The task was to make the Panda arm reach random points within its operational space. This simulated real-world scenarios where the robotic arm needs to deal with different positions. By picking random target coordinates, the study covered a wide range of possible end-effector locations. This approach mirrors the adaptability needed in real-world applications, showcasing the practicality and strength of the developed methodologies.

2.1.1 Action Space and Observation Space

The action space represents the set of actions of the Panda robot, the one we can take at each time step. While the virtual realm of MuJoCo provides a flexible arena for training the Panda robot, it is essential to ground our simulations in the physical reality of the real robot. The Panda robot, as a physical entity, is bound by certain limitations and constraints, especially concerning its joint movements and torques.

The joint limits, both in terms of position, velocity, acceleration and jerk, delineate the range within which each joint of the Panda robot can articulate. These limits, depicted in Table 2.1, safeguard against undesirable or even damaging configurations in the real-world scenario.

Name	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7
$q_{\max}$	2.8973	1.7628	2.8973	-0.0698	2.8973	3.7525	2.8973
$q_{\min}$	-2.8973	-1.7628	-2.8973	-3.0718	-2.8973	-0.0175	-2.8973
$\dot{q}_{\max}$	2.1750	2.1750	2.1750	2.1750	2.6100	2.6100	2.6100
$\ddot{q}_{\max}$	15	7.5	10	12.5	15	20	20
$\dddot{q}_{\max}$	7500	3750	5000	6250	7500	10000	10000
$\tau_{j_{\max}}$	87	87	87	87	12	12	12
$\dot{\tau}_{j_{\max}}$	1000	1000	1000	1000	1000	1000	1000

Table 2.1: Panda Robot Joint Limits and Constraints.
[11]

To align the simulation environment with these real-world constraints, the action space should be hard-constrained below the joints limits. The action space is defined as a continuous space representing the torques that could be applied on each joint of the Panda robot. The lower and upper limits are:

Joint	**Lower Limit**	**Upper Limit**
Joint 1	-2	2
Joint 2	-3	3
Joint 3	-2.5	2.5
Joint 4	-4	4
Joint 5	-2	2
Joint 6	-1	1
Joint 7	-1	1

Table 2.2: Joint Limits for the Panda Robot.

This formulation ensures that the simulated actions generated during training are not only feasible within the virtual environment but also adhere to the physical limits of the real Panda robot.

On the other side of the interaction, the observation space encapsulates the information available to the agent. This encompasses details about the current state of the robotic arm, such as joint angles, velocities and distances

between objects. Our observation space is also a gymnasium.spaces.box, the space dimension is 24 and each value is bounded between -1 and 1.

2.1.2 Training Process

The training process in MuJoCo involves iterative interactions between the Panda robot and its simulated environment. At each time step, the agent receives observations from the environment, processes this information, selects actions from its action space, and executes them. The dynamics of the simulated environment determine the next state, and the process repeats as can be seen in Alg. 1.

Algorithm 1 Reinforcement Learning Loop in MuJoCo with Periodic Gradient Descent.

Initialize neural network model for policy π_θ with parameters θ
Initialize replay buffer D
Initialize counter $t \leftarrow 0$
Set update frequency N
for episode in range(num_episodes) **do**
    Reset environment with `reset()` function
    for timestep in range(max_timesteps) **do**
        Observe current state s
        Select action a using policy π_θ with exploration strategy
        Execute action a and observe reward r and next state s'
        Store transition (s, a, r, s') in replay buffer D
        Increment counter: $t \leftarrow t + 1$
        if $t \mod N == 0$ **then**
            Sample a mini-batch from D and perform gradient descent on π_θ using optimization algorithm with learning rate α
        end if
        Move to the next state: $s \leftarrow s'$
    end for
end for

The training is guided by reinforcement learning methodologies, where the agent learns to associate its actions with rewards or penalties based on the consequences of those actions. Through repeated cycles of exploration and exploitation, the agent refines its policy, optimizing its behaviour towards achieving defined objectives.

2.1.3 Simulation Functions

Central to the training process are the environment functions `step()`, `compute_reward()`, `get_obs()` and `reset()`.

step()

The `step()` function advances the simulation by one time step, considering the actions taken by the agent and updating the state accordingly. It also returns the observations, the reward value for the current state and the boolean done.

```
function step(self, action):
    // Apply the specified joint torques to the simulated robot
    self.sim.execute(self.action)

    // Obtain observations for the current state
    observations = self.get_obs()

    // Calculate the reward based on the achieved state
    reward = self.compute_reward()

    // Check for episode termination conditions
    done = check_termination()

    return observations, reward, done
```

get_obs()

Within the MuJoCo training environment, the `get_obs()` function plays a central role in shaping the observational landscape available to the Panda robotic arm during training extracting key information from the simulated environment providing the agent essential data to make decisions.

```
def get_obs(self):
    q = self.sim.get_state()["joint_pos"][:7]
    dq = self.sim.get_state()["joint_vel"][:7]
    eef = self.sim.get_state()["site_pos"][1]
    target_pos = self.sim.get_state()["site_pos"][0]
    self.obs_dict = {"sinq": np.sin(q), "cosq": np.cos(q),
                "tanhdq": np.tanh(dq),
                "tanhdist": np.tanh(eef[:3] - target_pos[:3])}
    return self.obs_dict
```

Breaking down the components of `get_obs()`, the function captures the state of the simulation using `self.sim.get_state()`, extracting joint positions (`q`) and velocities (`dq`) from the simulation state. The function constructs an observation dictionary (`obs_dict`) encapsulating various features, including sine (`sinq`) and cosine (`cosq`) of joint angles, hyperbolic tangent (`tanhdq`) of joint velocities, and hyperbolic tangent (`tanhdist`) of the distance between the end-effector and the target position. These values are already normalized to the $[-1, 1]$ interval, collectively forming the observational input provided to the RL agent.

compute_reward()

`compute_reward()` is the base of the Panda robot's decisions and learning process, it's defined as:

```
def compute_reward(self):
    dist = (norm(self.obs_array[21:])) / self.init_dist_goal
    joint_acc = self.sim.get_joints_acc()[:7] / 100
```

```
joint_acc = sum(np.tanh(abs(joint_acc)))
dq = sum(np.tanh(abs(self.sim_state["joint_vel"][:7])))
r = -2 * dist - 0.03 * joint_acc / (0.15 + dist)
    - 0.05 * joint_acc / abs(1.15 - min(1, dist))
    - 0.03 * dq
return r
```

Let's analyze the terms contributing to the reward:

- `dist`: Normalized distance between the Panda robot end effector and its goal, emphasizing the importance of reaching the target.
- `joint_acc`: Sum of hyperbolic tangent `tanh`-transformed absolute joint accelerations, promoting smooth and controlled movements.
- `dq`: Sum of `tanh`-transformed absolute joint velocities, encouraging gradual changes in joint positions.

The reward function incorporates a weighted combination of these terms, with specific coefficients indicating their relative importance. Notably:

- The term `-2 * dist` encourages the robot to minimize the distance to the goal.
- `- 0.03 * joint_acc / (0.15 + dist)` penalizes excessive initial joint accelerations, with a diminishing penalty as the robot approaches the goal.
- `- 0.05 * joint_acc / abs(1.15 - min(1, dist))` imposes a penalty for joint accelerations when the robot is close to or has reached the goal.
- `- 0.03 * dq` penalizes rapid changes in joint velocities.

This reward function balances achieving the task at hand, maintaining smooth movements, and minimizing abrupt changes in joint behaviour. The

acceleration penalty reflects the consideration that rapid acceleration in a real-world scenario might lead to errors due to structure limits (see Table 2.1).

It's important to note that going forward, the term "reward function" will still be used, but the reward should be interpreted as a penalty. Given that the cumulative reward is always negative, with a value closer to 0 indicating better performance. An advantage of employing the penalty function is its constrained nature. Unlike traditional reward functions, where the optimal performance is unbounded and can potentially approach infinity, the penalty function operates on a reversed scale. In this context, the best achievable reward is constrained to 0, with values close to it indicating superior performance. Tuning these coefficients can significantly impact the learning dynamics, representing a crucial aspect of refining the agent's behaviour during the training process.

reset()

`reset()` initializes or resets the simulation, providing a starting point for each episode in the training process. The position of the target gets calculated randomly inside the workspace and the robot joints are placed back to their initial positions. The initial distance between the end effector and the target also gets calculated to be used by the `compute_reward()` function

2.2 ROS and Gazebo Setup

In the context of sim-to-real transfer, the interaction between the simulation environment and the real robot is a critical aspect. This section outlines the configuration and setups involving ROS (Robot Operating System) and Gazebo for evaluation.

To bridge the gap between simulation and reality, a key component is the implementation of a custom `JointTorqueController`. This controller facilitates communication with the robot and allows for torque control. The `JointTorqueController` subscribes to `/joint_states`, sets joint torques, and manages the pause/unpause functionality in the Gazebo simulation.

2.2.1 JointTorqueController

The `JointTorqueController` is a ROS node responsible for handling torque commands in Gazebo. It utilizes various ROS packages such as `sensor_msgs`, `std_msgs`, `gazebo_msgs`, and more. The node is initialized to handle joint torques, joint states, and link states.

Joint Torque Commands

To exert torques on the robot joints, the `JointTorqueController` utilizes the `effort_joint_torque_controller` in Gazebo. The torques are published to respective joint controllers with the topic `/panda_joint+i+_controller/command` (where `i` is a string containing the joints number) using the `Float64` message type.

Joint State Subscription

One of the fundamental aspects of the `JointTorqueController` is its ability to acquire real-time feedback from the simulated robot. This is achieved through the subscription to the `/joint_states` topic, a central communication channel providing information about the robot's joint configurations.

ROS facilitates inter-process communication through topics, and in the case of robotic systems, the `/joint_states` topic plays a central role. This topic is a standardized way of transmitting information about the state of a robot's joints, including position, velocity, and effort.

The messages published on the `/joint_states` topic typically adhere to the `sensor_msgs/JointState` message type.

By subscribing to `/joint_states`, the `JointTorqueController` gains access to real-time data regarding the configuration of each joint in the simulated robot. This includes the instantaneous positions (`position`), velocities (`velocity`), and efforts (`effort`) exerted on each joint.

The `JointState` messages are timestamped, allowing the controller to synchronize the received joint state information with its internal clock. This synchronization is crucial for accurately calculating velocities and accelera-

tions, aiding in the dynamic control of the robot.

Upon receiving `JointState` messages, the `JointTorqueController` extracts relevant information for further processing. This includes obtaining joint positions (`joints_pos`), joint velocities (`joints_vel`), and gripper positions (`gripper_pos`), enabling a comprehensive understanding of the robot's state.

To enhance control strategies, the controller computes joint accelerations (`accelerations`). Since `JointState` doesn't provide the joint accelerations, it is achieved by differentiating the received joint velocities over time, providing insights into the dynamics of the robotic system.

In summary, the `JointTorqueController` utilizes the `/joint_states` topic to establish a dynamic feedback loop with the simulated robot. This real-time information, encompassing joint positions, velocities, and efforts, is fundamental for implementing advanced control algorithms and adapting the robot's behaviour to the evolving simulation environment.

Pause and Unpause Functionality

The `JointTorqueController` provides services for pausing and unpausing Gazebo physics. This functionality is essential for resetting the simulation, adjusting the environment, and managing the simulation's temporal aspects.

Transformation Frames

The node uses the `tf2_ros` package to manage transformations between different frames in the Gazebo simulation and reality. It employs a buffer to store and retrieve transformations, aiding in obtaining the positions of specific links.

Switching Controllers

During the simulation, there is a need to switch between controllers when resetting the joints' position. The `JointTorqueController` interfaces with the `controller_manager` package to seamlessly transition between torque and position controllers.

Sphere Pose and Resetting World

The controller is responsible for setting the pose of a sphere in the Gazebo environment and resetting the entire simulation world. This feature introduces variability and randomness, crucial for testing the robot's capabilities.

In summary, the `JointTorqueController` serves as a crucial interface between the simulated environment and Gazebo. Its functionalities enable torque control, real-time feedback, and dynamic adjustments, contributing significantly to the sim-to-real transfer process.

2.3 Grid Search Methodology

The success of reinforcement learning (RL) algorithms is often contingent on the careful tuning of hyperparameters. The following section outlines the parameters included in the grid search, providing a detailed rationale for their selection:

- **Action Noise:** The action noise added to the agent's actions during exploration. The options considered include `None`, `Normal`, and `OrnsteinUhlenbeck`. The inclusion of different action noises aims to understand the impact of exploration strategies on the learning process.
- **Learning Rate:** The learning rate of the RL algorithm, a critical parameter that determines the size of the update made to the model's parameters during the training process. It influences how quickly or slowly a model learns from new experiences. Mathematically, the learning rate (α) is applied to the gradient of the loss function with respect to the model parameters ($\nabla\theta$) in order to update the model parameters (θ):

  $$\theta_{t+1} = \theta_t - \alpha \cdot \nabla\theta_t$$

  Here:

  - θ_t represents the model parameters at time step t.

- $\nabla\theta_t$ is the gradient of the loss function with respect to the model parameters at time step t.
- α is the learning rate.

- **Gamma:** In reinforcement learning, the discount factor gamma (γ) is a parameter that controls the importance of future rewards in the agent's decision-making process. Mathematically, the discount factor is used in the calculation of the discounted cumulative reward, also known as the discounted return.

  The discounted return (G_t) at time step t is defined as the sum of future rewards, each multiplied by the discount factor raised to the power of the time step difference:

$$G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots$$

  Here:

  - G_t is the discounted return at time step t,
  - R_t is the reward obtained at time step t, and
  - γ is the discount factor (a value between 0 and 1).

  The discount factor serves several purposes:

  1. **Temporal Focus:** A discount factor less than 1 gives more emphasis to immediate rewards, making the agent more focused on short-term gains. This is important in scenarios where immediate actions have a more significant impact.
  2. **Convergence:** The discount factor ensures that the sum converges, preventing an infinite sum of rewards. Without discounting, the sum of rewards in an infinite horizon might not converge.
  3. **Preference for Shorter Paths:** A smaller discount factor can lead the agent to prefer shorter paths to rewards since future rewards are less influential.

4. **Handling Uncertainty:** It incorporates a degree of uncertainty about the future, making the agent more robust in dynamic environments.

- **Buffer Size:** The size of the replay buffer used to store and sample experiences for training. A buffer size of 1,000,000 was selected to evaluate the impact of memory capacity on learning.
- **Batch Size:** The number of experiences sampled from the replay buffer in each training iteration. Values of 2,048 were explored to investigate the impact of batch size on learning stability.
- **Learning Starts:** The number of steps the agent takes in the environment before starting the learning process. This parameter is set to 5,000 to allow the agent to accumulate experiences before initiating learning.
- **Train Frequency:** The frequency at which the agent performs a learning update. A tuple of (500, "step") indicates an update every 500 steps.
- **Gradient Steps:** The number of optimization steps performed on each learning update. Two values, 50 and 25, were tested to assess the sensitivity of training to the number of gradient steps.
- **Sigma:** The standard deviation of the action noise when using `Normal` and `OrnsteinUhlenbeck` action noise. Values of 0.2, 0.1, and 0.05 were examined to understand the impact of noise magnitude on exploration.
- **Tau:** The soft update coefficient for target networks. Also called "polyak update" or "target network update" is a way of slowly updating the parameters of the target network towards the parameters of the main network. Mathematically, it is represented as:

$$\theta_{\text{target}} \leftarrow \tau \cdot \theta_{\text{online}} + (1 - \tau) \cdot \theta_{\text{target}}$$

Here:

- θ_{target} represents the parameters of the target network,
- θ_{online} represents the parameters of the online (main) network,
- τ is the Polyak update parameter.

This update is performed periodically during training. The slow update helps stabilize the training process by providing a moving average of the online network's parameters to the target network. This can make the learning more robust and prevent the target network from oscillating or diverging during training.

- **Seed:** The random seed is used to initialize the environment and algorithms.
- **Policy Architecture:** The neural network architecture of the policy. A feedforward architecture with three layers of 512 units each is employed and the number of critics is set to 2. This exploration aims to evaluate the influence of policy complexity on learning performance.

Additionally to hyperparameters, many different algorithms have been tested. In particular from Stable Baselines3 [12]: Deep Deterministic Policy Gradient (DDPG [13]), Soft Actor-Critic (SAC [14]) and Twin Delayed DDPG (TD3 [15]) while from Stable Baselines3 Contrib [16] we used Truncated Quantile Critics (TQC [17]).

2.3.1 Grid Search Results and Analysis

The grid search was conducted over a diverse set of hyperparameters to comprehensively explore their impact on the training effectiveness of the algorithms in the `PandaTorquesReachSite` environment. The following key findings and insights were derived from the extensive experimentation:

Action Noise Impact

The exploration of diverse action noise types, including `None`, `Normal`, and `OrnsteinUhlenbeck`, yielded varied improvements across different algorithms.

Notably, the role of action noise proved essential for DDPG, an algorithm reliant on a deterministic policy. The addition of action noise was essential in introducing the necessary exploration-exploitation balance.

For algorithms like SAC and TD3, while not strictly required due to their stochastic policies, the incorporation of action noise demonstrated enhanced convergence and improved training stability. However, the effect of action noise on the TQC algorithm would lead, in certain scenarios, to difficulties in reaching local minima.

An interesting overview impact of action noise in DDPG, TD3, and SAC can be found in the work of Hollenstein et al. [18]. This research provides valuable insights into the subtle effects of action noise on deterministic policies and contributes to a deeper understanding of its role in reinforcement learning algorithms.

In the context of DDPG, the impact of different action noises, specifically `Normal` and `OrnsteinUhlenbeck`, appears to exhibit minimal distinctions, as illustrated in Figure 2.1. The visual representation of the training trajectories under various action noise types suggests comparable performance between the two. This observation implies that, within the DDPG framework for the `PandaTorquesReachSite` environment, the choice between normal and Ornstein-Uhlenbeck action noises may not significantly influence training outcomes.

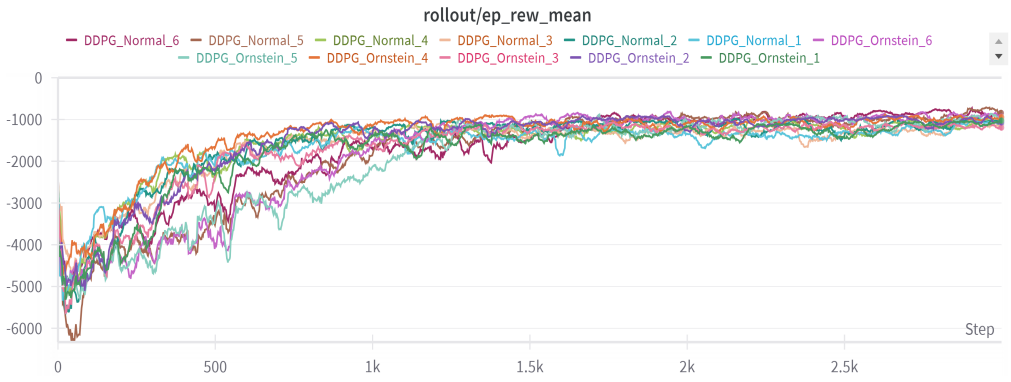


Figure 2.1: Comparison of Normal and OrnsteinUhlenbeck action noises in DDPG.

Impact of Gamma in SAC

In the SAC algorithm applied to the `PandaTorquesReachSite` environment, the choice of the discount factor (γ) significantly influences training outcomes. A comparative analysis of evaluation rewards with different γ values, specifically 0.95 and 0.99, reveals substantial differences in performance, as depicted in Figure 2.2. Notably, a γ value of 0.95 results in inferior training compared to the more conventional value of 0.99.

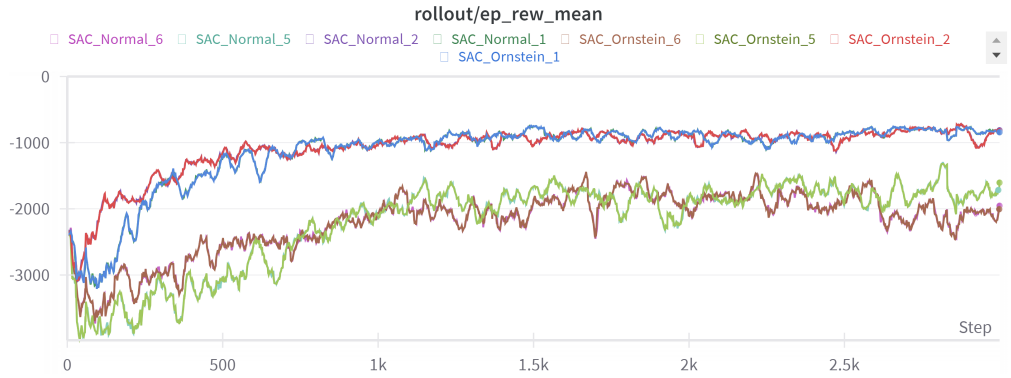


Figure 2.2: Impact of γ on training trajectories in SAC.

Results of TD3 algorithm

The TD3 (Twin Delayed DDPG) algorithm, often regarded as an enhancement over DDPG [19], exhibited suboptimal results in the `PandaTorquesReachSite` environment. Surprisingly, its performance was not superior to the original DDPG, contrary to the expected trend. Similarly to the SAC (Soft Actor-Critic) algorithm, TD3 delivered unfavourable results when γ was set lower than 0.99.

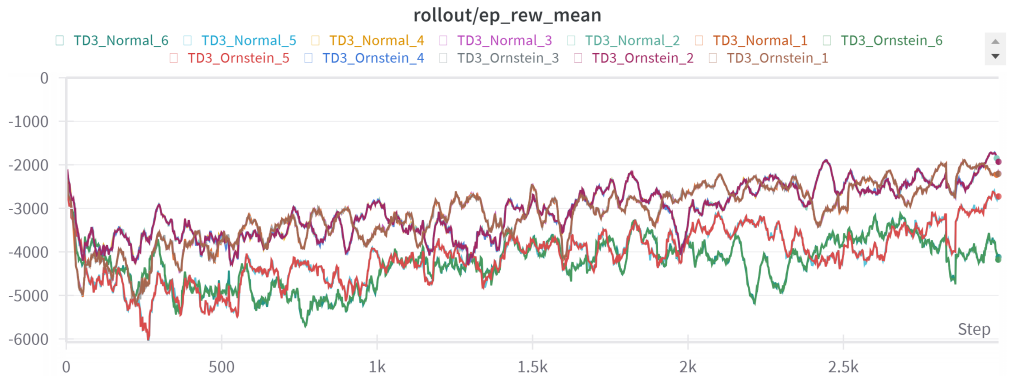


Figure 2.3: Results of TD3 algorithm.

The plot clearly shows that TD3 had a hard time learning and improving its policy in this environment. The observed decrease in performance suggests a need for a thorough examination of the adaptability and robustness of TD3 in this specific setting.

Learning Rate Sensitivity

The learning rate parameter, a crucial factor in determining the step size during optimization, showed notable sensitivity. Contrary to expectations, higher learning rates (e.g., 0.01) often resulted in a low convergence rate for all algorithms tested.

To illustrate this sensitivity, we conducted experiments with different learning rates and generated two key plots for the TQC algorithm. The first comparison, depicted in Figure 2.4, contrasts the outcomes of learning rates 0.0001 and 0.001, revealing minimal differences in results. However, the second comparison, shown in Figure 2.5, demonstrates that models trained with a learning rate of 0.01 consistently underperformed when compared to their counterparts.

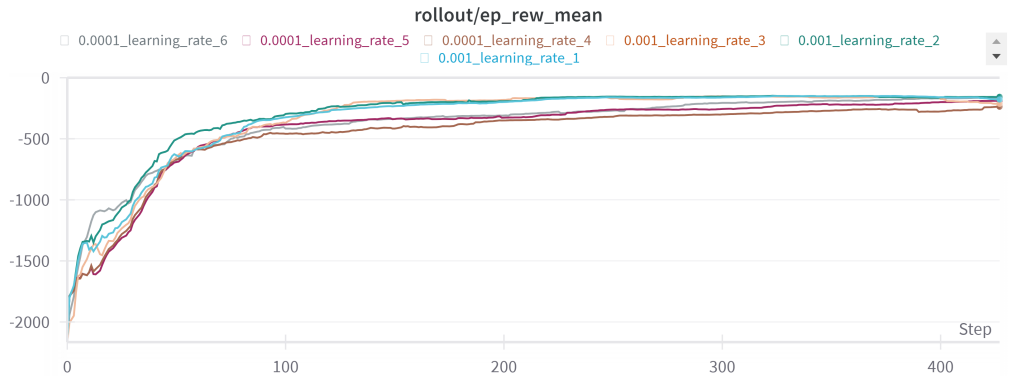


Figure 2.4: Comparison of learning rates 0.0001 and 0.001.

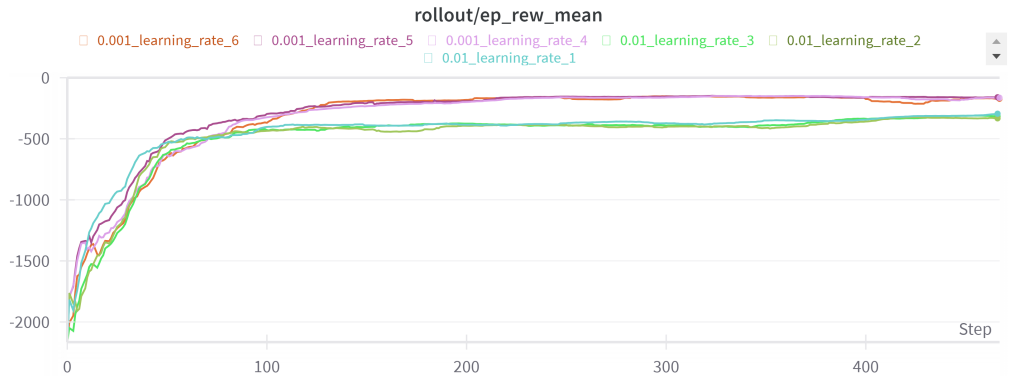


Figure 2.5: Comparison of learning rates 0.001 and 0.01.

Tau and Target Network Update

Exploring different values of tau for target network updates showed the importance of this parameter. Values equal to or exceeding 0.9 often led to suboptimal learning outcomes. This observation emphasized the need to moderate the update speed of target networks for improved stability and performance.

Gradient Steps and Training Stability

The number of gradient steps performed during each learning update was found to impact training stability. Higher values, such as 100, sometimes resulted in erratic behaviour and reduced overall effectiveness.

2.3.2 Optimal Hyperparameters for Different Algorithms

Several hyperparameters exhibited complex interactions, necessitating an understanding of their collective impact. Notably, combinations of lower learning rates, moderate tau values, and reduced gradient steps often contributed to more stable and effective learning. The following sets of hyperparameters were tested and the ones marked in **bold** represent the optimal choices for each algorithm in the `PandaTorquesReachSite` environment:

DDPG

- **Learning Rate:** {0.005, **0.002**, 0.001}
- **Gamma:** {0.99, 0.97, **0.95**}
- **Sigma:** {0.2, **0.1**}
- **Tau:** {0.005, **0.01**, 0.02}

SAC

- **Learning Rate:** {0.005, **0.002**, 0.001}
- **Gamma:** {**0.99**, 0.95}
- **Sigma:** {**0.2**, 0.1}
- **Tau:** {0.005, **0.01**, 0.02}

TD3

- **Learning Rate:** {0.005, **0.002**, 0.001}
- **Gamma:** {**0.99**, 0.95}
- **Sigma:** {**0.2**, 0.1}
- **Tau:** {0.005, **0.01**, 0.02}

TQC

- **Learning Rate:** {0.005, 0.002, **0.001**}
- **Gamma:** {0.99, **0.95**}
- **Tau:** {**0.9**, 0.8}

These optimal hyperparameter configurations reflect the specific parameter choices that demonstrated superior performance for each algorithm in achieving effective learning and control.

Performance of TQC Algorithm

The TQC algorithm consistently outperformed alternative algorithms (DDPG, SAC, and TD3) in our environment. This superiority, also demonstrated by Kuznetsov et al. [20], is more evident when confronted directly with the other algorithms as shown in Figure 2.6.

Moreover, differently from SAC and TD3, the TQC presented insensitivity to lower γ values. Remarkably, setting γ to 0.95 did not lead to the anticipated degradation in performance; the policy showed only marginal improvement when γ was set to 0.9. In general, the variation of parameters did not exert a significant influence on the training outcomes, despite the diverse set of parameters explored, the impact on the overall training effectiveness remained relatively consistent (see Figure 2.7).

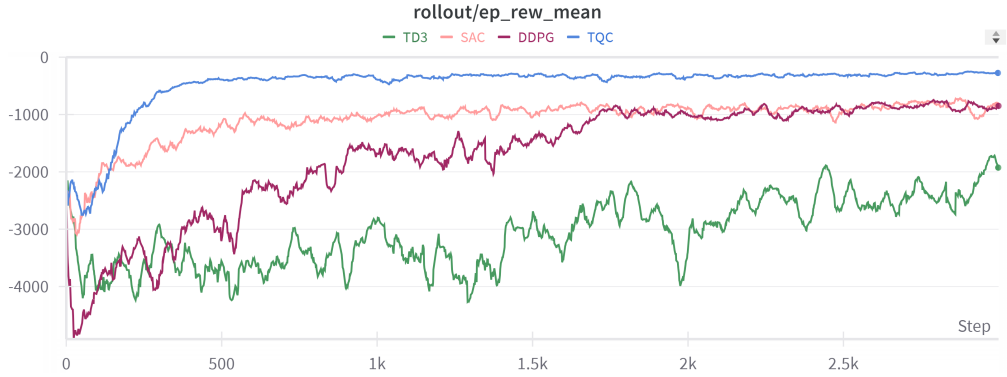


Figure 2.6: Comparison between algorithms.

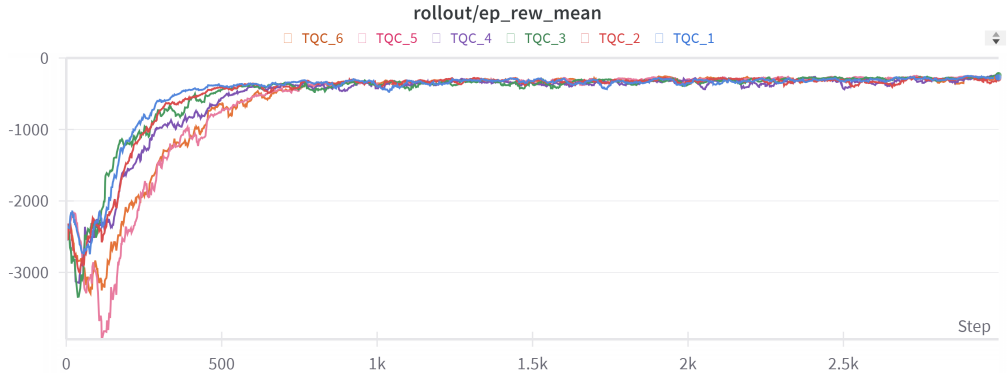


Figure 2.7: Results of TQC algorithm.

2.4 Challenges and Adaptations

During the transfer from MuJoCo to Gazebo several issues emerged, in particular, the disparities in the handling of damping and friction and the topic publishing rate posed significant problems. The reason behind this approach lies in the understanding that if the simulation functions effectively in Gazebo, it is more likely to exhibit similar behaviour in the real world. Testing in Gazebo acts as a preemptive and safer measure, helping identify and resolve issues before transitioning to the physical system.

2.4.1 Discrepancies in Friction Models

In MuJoCo, damping is applied as a force linear in velocity, included in the passive forces. However, the usage of damping in Gazebo differs, as it represents an opposing force to joint velocity, acting to decelerate the joint motion. Friction, on the other hand, is computed differently in both simulators. In Gazebo, it involves Coulomb friction coefficients, including `mu` and `mu2`, corresponding to the friction coefficients in two directions. This contrasts with MuJoCo, where friction and damping are conceptually similar but computed uniquely and it doesn't allow to set the static friction separately.

The absence of a mechanism to set proper static friction in MuJoCo complicated the challenge, as Gazebo employs a combination of damping and friction coefficients with distinctive formulations. Adjusting the simulation to address these differences became necessary to ensure accurate and consistent outcomes.

2.4.2 Friction Solution

To bridge the gap in dynamic friction between MuJoCo and Gazebo, a customized solution was implemented, focusing on the Franka Emika Panda Robot. The primary parameters influencing the reality gap were identified as inertias, static frictions, and dynamic frictions. A comprehensive approach, outlined in the paper [21], introduced a dynamic friction solution specifically addressing link-side friction. The dynamic friction compensation formula for joint j, as derived from the methodology presented in the referenced paper, is expressed as follows:

$$\tau_{f,j} = \frac{\varphi_{1,j}}{1 + e^{-\varphi_{2,j}(\dot{q}_j + \varphi_{3,j})}} - \frac{\varphi_{1,j}}{1 + e^{-\varphi_{2,j}\varphi_{3,j}}}, \quad j \in [1, \ldots, 7] \tag{2.1}$$

Here, $\tau_{f,j}$ represents the dynamic friction compensation for joint j, $\varphi_{1,j}$, $\varphi_{2,j}$, and $\varphi_{3,j}$ are parameters specific to each joint.

	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7
φ_1	0.5462	0.8722	0.6407	1.2794	0.8390	0.3030	0.5649
φ_2	5.1181	9.0657	10.1360	5.5903	8.3469	17.1330	10.3360
φ_3	0.0395	0.0259	−0.0461	0.0362	0.0262	−0.0210	0.0036

Table 2.3: Estimated Friction Parameters.

It's essential to highlight that this dynamic friction solution specifically addresses link-side friction, leaving motor-side friction not explicitly considered. Internally, the `libfranka` library incorporates a hidden friction observer that compensates for a significant portion of motor-side friction, recognizing its dominance over link-side friction.

By setting the friction parameters to 0 in both simulators and utilizing the dynamic friction compensation formula, the resulting friction for each joint can be subtracted from the policy's action. This approach aims to unify the friction modelling between MuJoCo and Gazebo, contributing to a more consistent simulation for the Franka Emika Panda Robot.

2.4.3 Frequency Mismatch in Joint State Controller

An aspect affecting the fidelity of the simulation was the disparity between the frequencies of the joint state controller [22] in Gazebo and the counterpart function providing joint states in MuJoCo. The joint state controller, responsible for obtaining joint positions, exhibited frequency-related issues. When its frequency surpassed 250 Hz, the controller produced identical position values for multiple consecutive timesteps even if the frequency of the joint state controller was set to 1000 Hz.

The frequency mismatch posed challenges in accurately calculating accelerations, as the joint state controller provided the same values of positions and velocities for consecutive timesteps.

Given that the controller outputs only positions and velocities, the acceleration must be derived using the formula:

$$a_i = \frac{v_i - v_{i-1}}{t_i - t_{i-1}}$$

where a_i is the acceleration of the i-th joint, v_i is the velocity at time t_i, v_{i-1} is the velocity at the previous time t_{i-1}, and $t_i - t_{i-1}$ is the time difference between the current and previous instants. Having both the velocity values equal for 2 timesteps would lead to an incorrect acceleration calculation.

Maintaining an optimal frequency is crucial for torque-controlled robots, as they require precise actions for each timestep within a simulation iteration. High-frequency control ensures fine-grained influence over movements, adapting to dynamic changes and disturbances.

Additionally, torque-controlled robots heavily rely on receiving timely and accurate information from the simulation environment. Parameters such as position, velocity, acceleration, and the distance from the target play central roles in formulating effective torque commands. High-frequency communication between the simulation and the controller is essential for maintaining synchronization and enabling the robot to exhibit the desired behaviour with responsiveness and precision as the actions the policy gives are strictly related to the observations received.

To address this frequency mismatch, adaptations were implemented in the joint state controller to ensure reliable position data acquisition. Limiting the controller frequency to values compatible with the simulation dynamics played a crucial role in preserving the integrity of acceleration calculations.

The challenges outlined above underscore the intricate process of transitioning robotic simulations between different environments. The adaptations made were essential in reducing the differences between MuJoCo and Gazebo, enhancing the accuracy and stability of the simulated robotic system.

Chapter 3

Sim-to-Real Parameters Adaptation

In this section, we will show the process of adapting simulation parameters to real-world conditions for the Panda robot. The focus encompasses static friction compensation, gravity compensation, and the optimization of dynamic friction and inertia parameters. Each of these elements plays a crucial role in bridging the gap between simulated and real-world robotic environments.

3.1 Gravity and Friction Issues

The real Panda robot exhibited an issue where, upon activating the torque controller, the arm would slowly fall due to inadequate gravity compensation. Moreover, we observed different kinetic friction based on the direction of applied torque leading to different velocity trajectories for positive and negative torque values. The second major problem involved the unknown static friction parameters. Until this point, our efforts had been primarily focused on simulating the effects of dynamic friction in the simulation. The equation we used is only dependent on the velocity and is equal to 0 when the joint is not moving, we need another way to simulate the static friction in a manner that both MuJoCo and Gazebo would show similarities.

A systematic procedure was created to enhance both gravity compensation

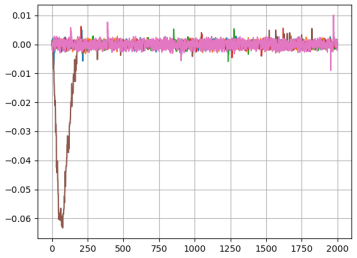
and different kinetic friction in a single step. The idea was to apply constant torque (both positive and negative) to each joint and observe the resulting velocity trajectory. To find the static friction parameters a similar approach was used but instead of a static torque, we used a constantly increasing torque in a way to obtain the value needed to break the static friction.

3.1.1 Gravity Compensation

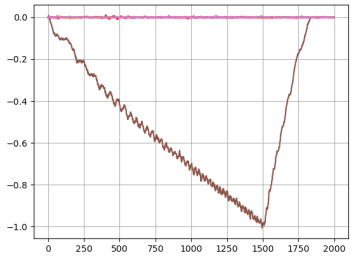
To observe the effects of erroneous gravity compensation and different kinetic friction constant torques were applied to individual joints, this revealed discrepancies for various configurations. In particular, joints 2 and 4, which are the ones most responsible for gravity compensations, showed very different trajectories for positive and negative torque. The joints 1, 3 and 6, while not usually in charge of gravity compensation, also revealed discrepancies in the trajectories because of different kinetic friction. Joints 5 and 7 did not show any relevant problems. Specifically, torque values were applied as follows:

- Joint 6: -0.6 N/m, -0.45 N/m, 0.45 N/m, 0.6 N/m
- Joint 4: -1.5 N/m, -1 N/m, 1 N/m, 1.5 N/m
- Joint 3: -1.5 N/m, -1 N/m, 1 N/m, 1.5 N/m
- Joint 2: -1.5 N/m, -1 N/m, 1 N/m, 1.5 N/m
- Joint 1: -2 N/m, -1 N/m, 1 N/m, 2 N/m

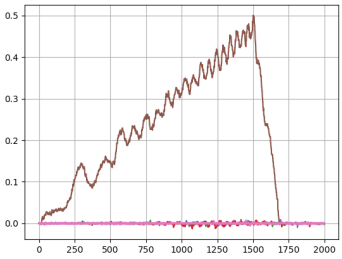
These torque applications resulted in a total of 20 velocity trajectories. The subsequent plots illustrate the motion of each joint individually under the influence of various torque values.



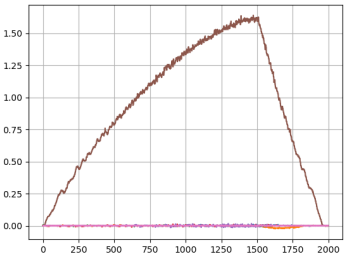
(a) Torque = -0.45 N/m



(b) Torque = -0.6 N/m

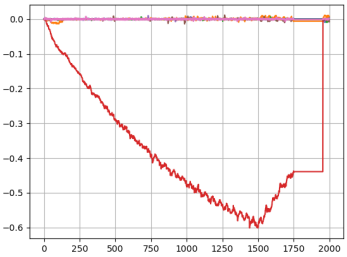


(c) Torque = 0.45 N/m

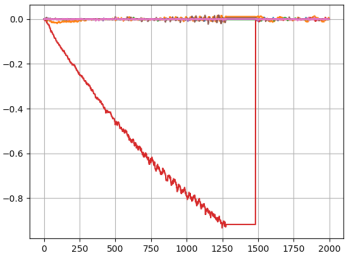


(d) Torque = 0.6 N/m

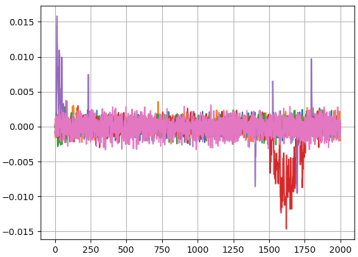
Figure 3.1: Velocity trajectories before offset for Joint 6.



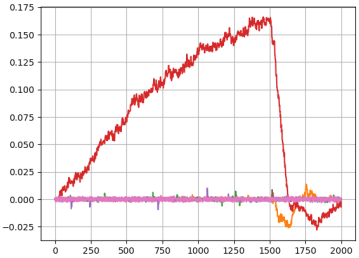
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

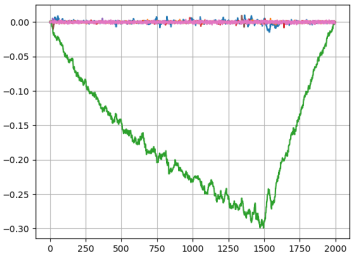


(c) Torque = 1 N/m

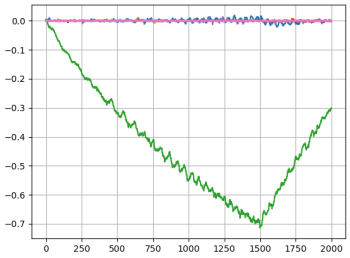


(d) Torque = 1.5 N/m

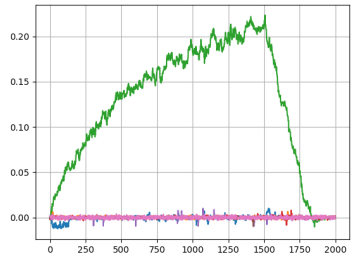
Figure 3.2: Velocity trajectories before offset for Joint 4.



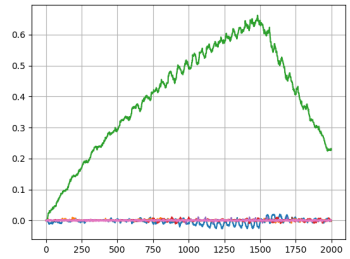
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

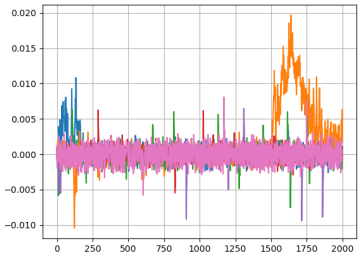


(c) Torque = 1 N/m

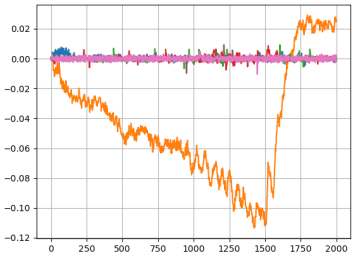


(d) Torque = 1.5 N/m

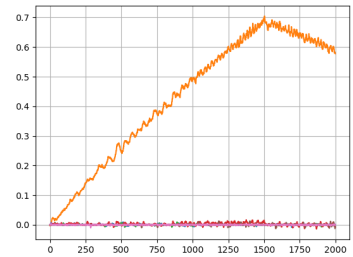
Figure 3.3: Velocity trajectories before offset for Joint 3.



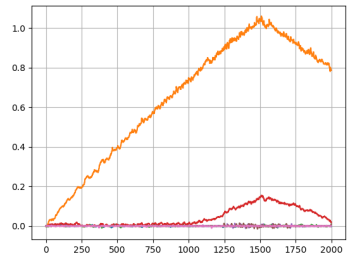
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

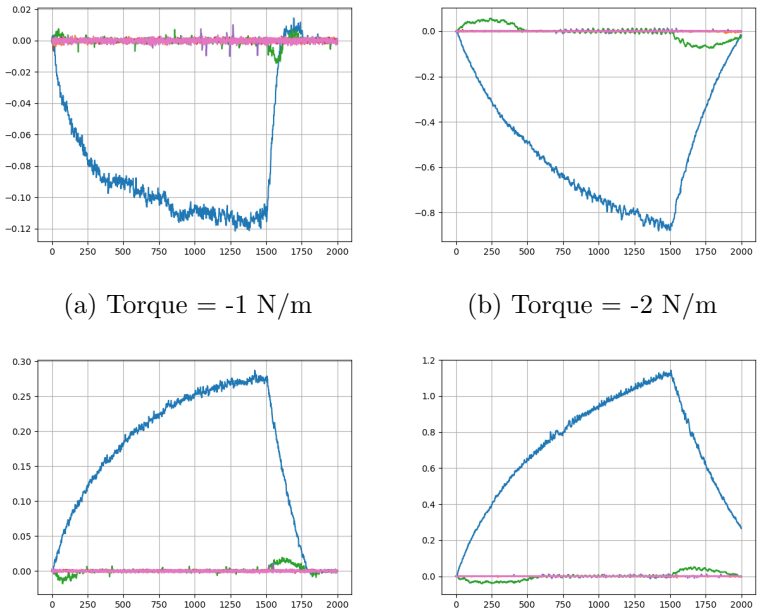


(c) Torque = 1 N/m



(d) Torque = 1.5 N/m

Figure 3.4: Velocity trajectories before offset for Joint 2.



(a) Torque = -1 N/m

(b) Torque = -2 N/m

(c) Torque = 1 N/m

(d) Torque = 2 N/m

Figure 3.5: Velocity trajectories before offset for Joint 1.

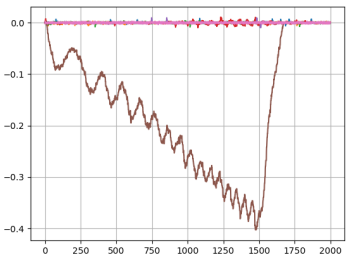
As depicted in Figures 3.1a, 3.2c, and 3.4a, when applying torques of -0.45 N/m, 1 N/m, and -1 N/m, respectively, Joint 6, Joint 4, and Joint 2 exhibit no movement. Interestingly, these problems are not observed with torques of the opposite sign. This issue is caused by both incorrect gravity compensation and varying kinetic friction. It can be assumed that in the idle configuration, the system is not situated at the midpoint of the friction cone. Generally, trajectories corresponding to equal positive and negative torques are not comparable.

To solve these issues, an offset was introduced for each joint, which aimed to make positive and negative trajectories visually similar. The offset values applied to each joint were determined through a manual adjustment process. The resulting offset values are as follows:

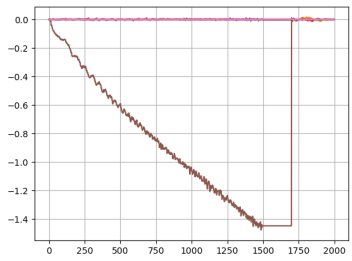
Joint	Offset Value
Joint 1	-0.127
Joint 2	-0.725
Joint 3	0.027
Joint 4	0.53
Joint 5	0
Joint 6	-0.06
Joint 7	0

Table 3.1: Offset Values for Gravity Compensation and Kinetic Friction

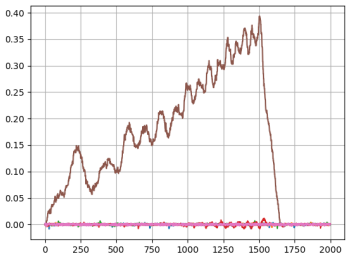
By adding these offset values to the joints for the actions that the policy gives we were able to improve gravity compensation and kinetic friction, creating consistent velocity trajectories for positive and negative torques across all joints.



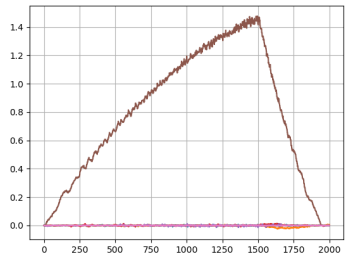
(a) Torque = -0.45 N/m



(b) Torque = -0.6 N/m

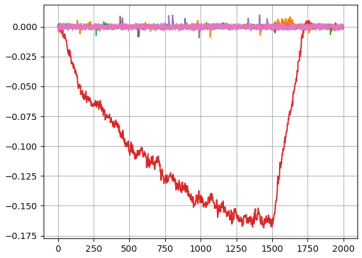


(c) Torque = 0.45 N/m



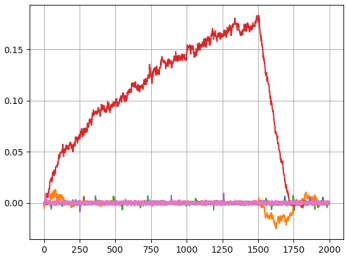
(d) Torque = 0.6 N/m

Figure 3.6: Velocity trajectories after offset for Joint 6.

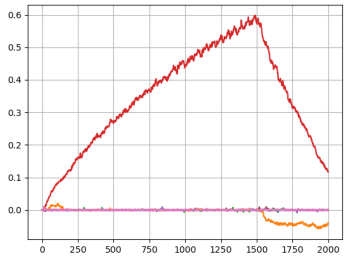


(a) Torque = -1 N/m

(b) Torque = -1.5 N/m

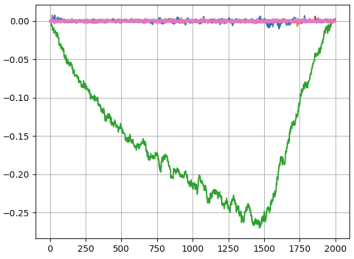


(c) Torque = 1 N/m

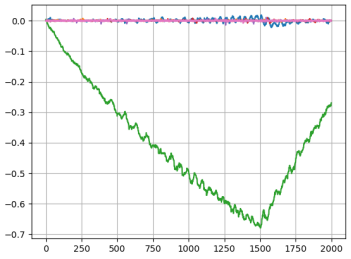


(d) Torque = 1.5 N/m

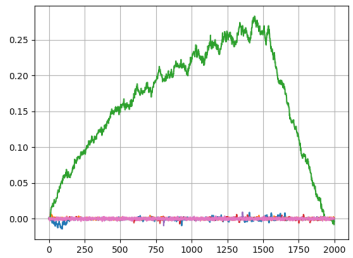
Figure 3.7: Velocity trajectories after offset for Joint 4.



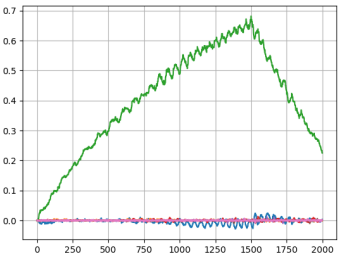
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

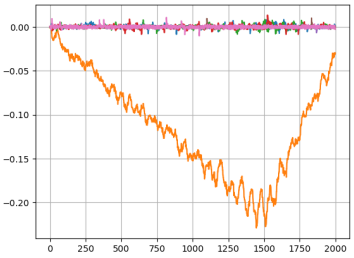


(c) Torque = 1 N/m

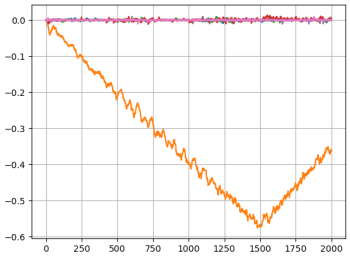


(d) Torque = 1.5 N/m

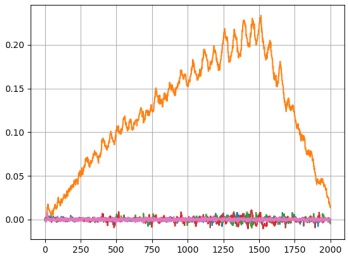
Figure 3.8: Velocity trajectories after offset for Joint 3.



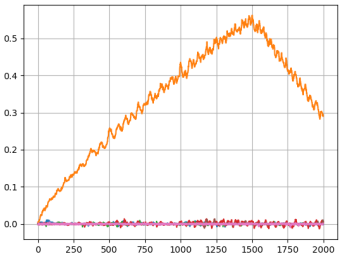
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

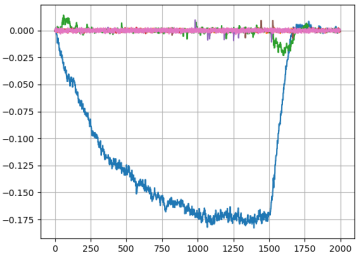


(c) Torque = 1 N/m

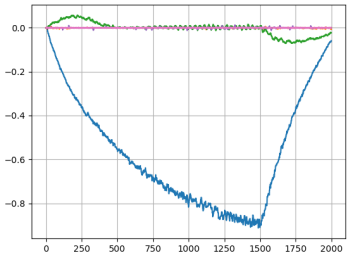


(d) Torque = 1.5 N/m

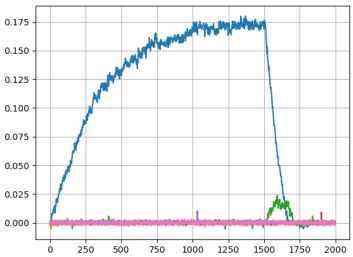
Figure 3.9: Velocity trajectories after offset for Joint 2.



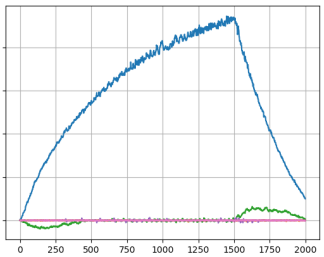
(a) Torque = -1 N/m



(b) Torque = -2 N/m



(c) Torque = 1 N/m



(d) Torque = 2 N/m

Figure 3.10: Velocity trajectories after offset for Joint 1.

3.1.2 Static Friction Compensation

Another prominent issue in the real Panda robot was the existence of different kinetic friction for positive and negative torques applied to joints.

To solve this problem, linearly increasing torques were applied to each joint, both positively and negatively. Torque values were incremented until joint movement was detected, indicating the point where static friction was overcome. This process was repeated for each joint, resulting in torque values needed to break static friction in both directions.

The following plots present, for each joint, the applied action on the left, and to the right, a zoomed-in view highlighting the specific moment where static friction is overcome.

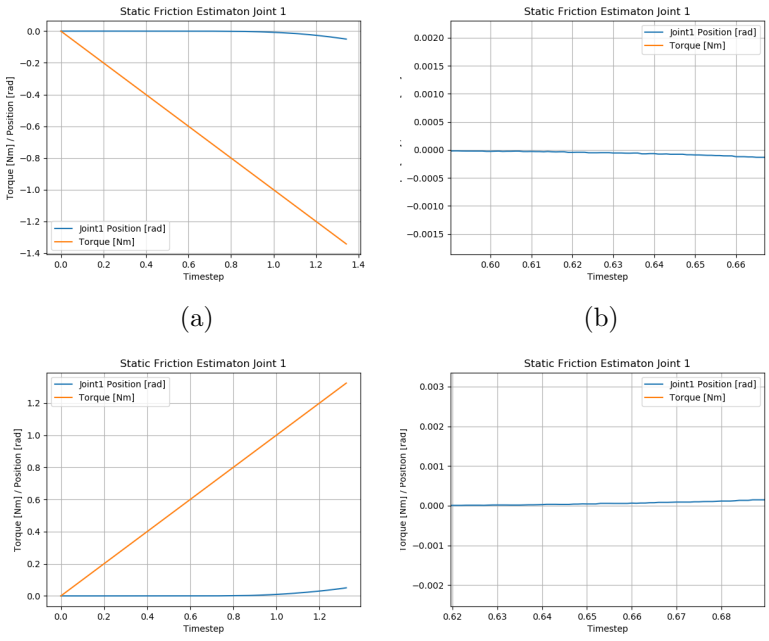


Figure 3.11: Static friction for Joint 1.

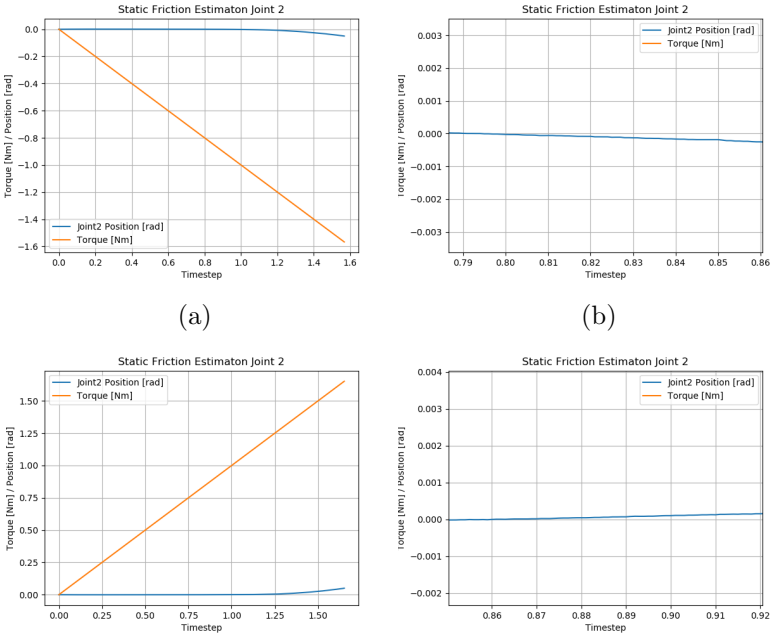


Figure 3.12: Static friction for Joint 2.

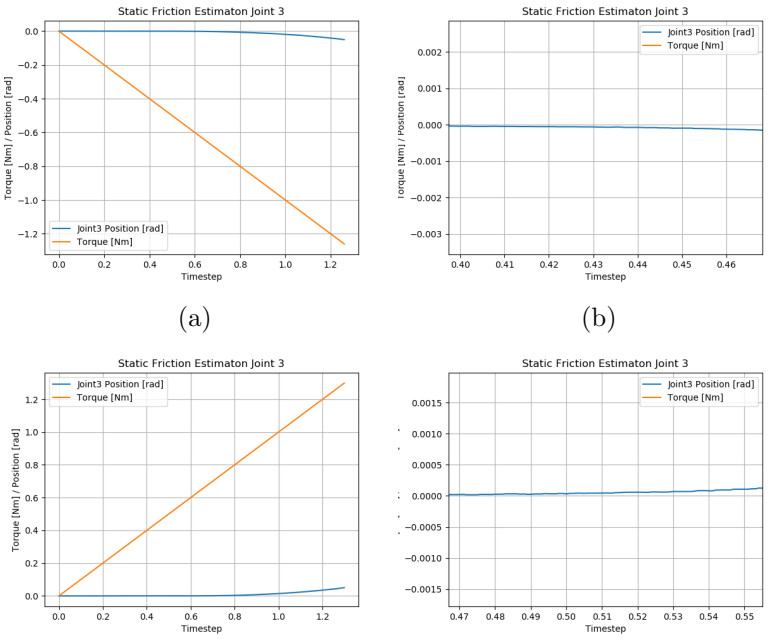


Figure 3.13: Static friction for Joint 3.

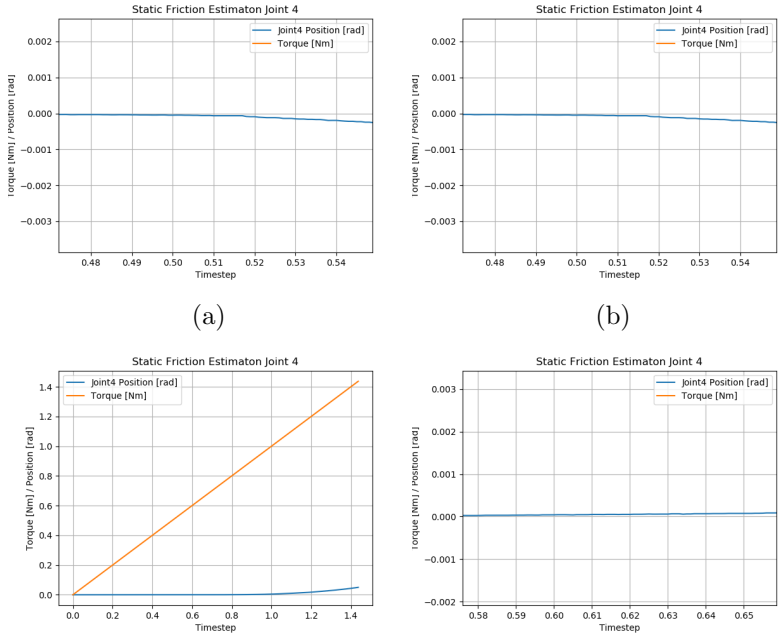


Figure 3.14: Static friction for Joint 4.

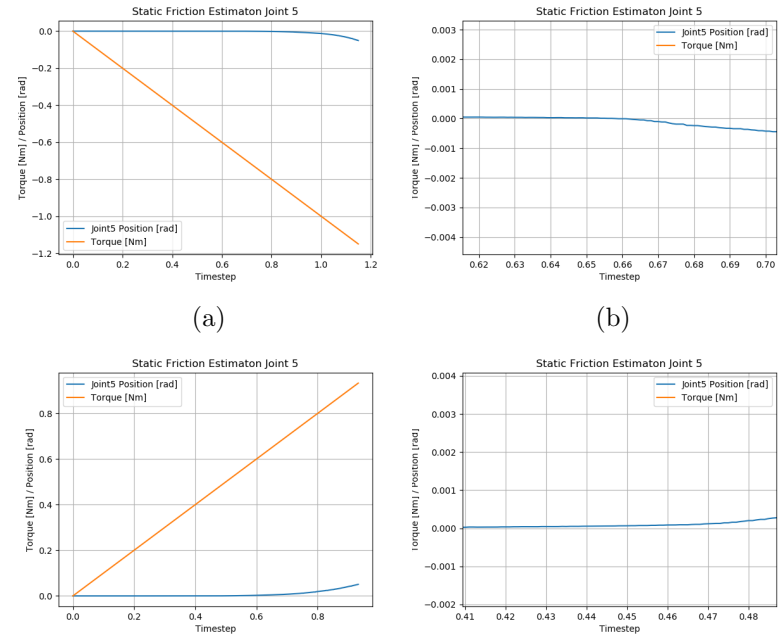
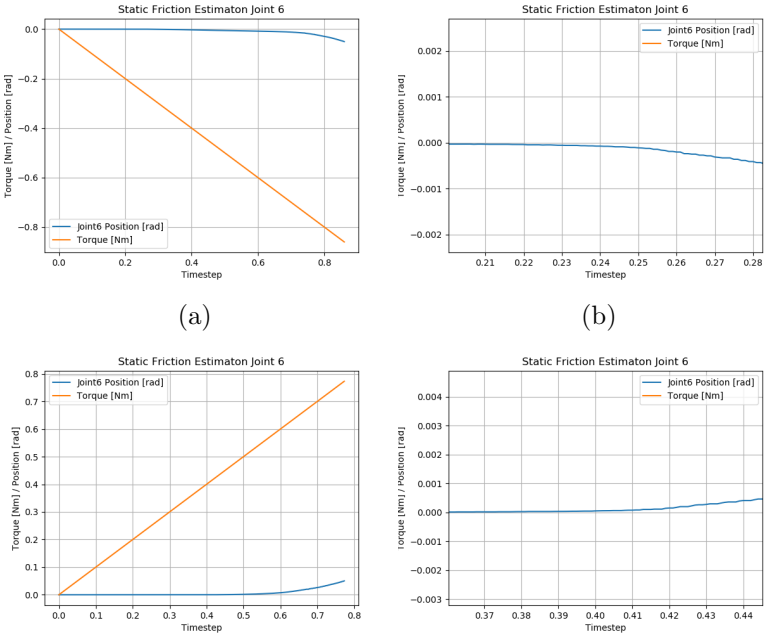


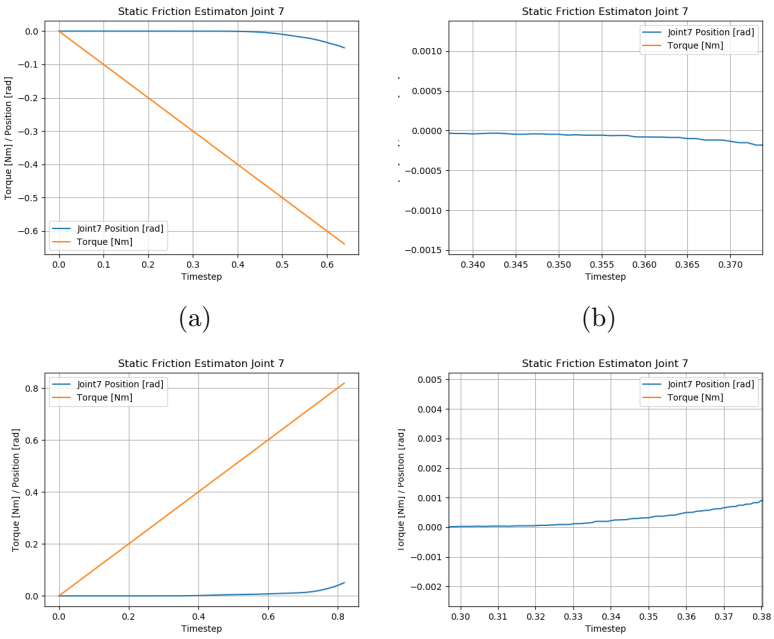
Figure 3.15: Static friction for Joint 5.



(a) (b)

(c) (d)

Figure 3.16: Static friction for Joint 6.



(a) (b)

(c) (d)

Figure 3.17: Static friction for Joint 7.

The obtained torque values for joints J1 to J7 are summarized as follows:

Joint	Positive Torque (Nm)	Negative Torque (Nm)
J1	0.65	-0.62
J2	0.88	-0.81
J3	0.51	-0.42
J4	0.61	-0.51
J5	0.44	-0.67
J6	0.40	-0.23
J7	0.32	-0.35

Table 3.2: Positive and Negative Torque Values for Each Joint

These torque values were then used to simulate static friction in both MuJoCo and Gazebo during training. A condition was introduced: if the joint velocity was within the range of -0.03 to 0.03 (indicating stationary), and the action generated by the policy fell within the positive and negative torque values found for that joint, then the applied action was set to 0, simulating the action's inability to break static friction.

3.2 Trajectory Comparison and Evolution Strategy

In this section, we discuss the optimization process for dynamic friction and inertia parameters, the approach involves an evolution strategy with a single population.

To address the vastly different dynamic friction observed in the real Panda robot compared to simulation environments, an optimization process was implemented. The goal was to find friction and inertia parameters that minimize the error between simulated and real trajectories. To generate the trajectory for each joint, a constant torque was applied for the initial 1500 timesteps of the simulation. During this period, the joint moved in response

to the applied torque. This phase allowed capturing the dynamic response of the system under torque influence.

After the initial 1500 timesteps, no external torque was applied to the joint for the remaining 500 timesteps. This intentional interruption of torque application enabled the observation of dynamic friction and inertia effects on the joint's behaviour when it was in motion without any external driving force.

3.2.1 Friction and inertia Parameters Adjustment

The optimization process involves adjusting friction parameters ($FI1X$, $FI2X$, $FI3X$) for each joint (X) and the six inertia values of the matrix for each link using an evolution strategy. The parameters were adjusted based on the error between simulated and real trajectories. The algorithm involved the following steps:

1. For each joint in the real robot, different trajectories were obtained.

2. For each joint, friction and inertia parameters were randomly perturbed in a neighbourhood of the value to explore the parameter space.

3. A check is made so that the eigenvalues of the inertia matrix obtained with the new parameters have positive eigenvalues, if not repeat the randomization of inertia parameters.

4. Simulated trajectories were generated using the adjusted parameters.

5. The error between real and simulated trajectories was calculated.

6. If the new parameters resulted in a lower error, they were accepted; otherwise, the parameters were reverted.

7. The process was repeated from point 2 until a satisfying set of parameters was found.

Error Calculation

The error between real and simulated trajectories was calculated by summing the absolute values of the difference between the joint positions in the real and simulated environment with all the trajectories given at regular intervals. This error served as the fitness function for the evolution strategy. Additionally, another type of error was computed without considering the absolute value. Instead, the sign of this error before the sum was adjusted based on the applied torque for that trajectory (negative or positive) before summing the errors together for that joint. This adjustment allows for discerning whether the friction in the simulation is too high or too low. The sign of this error is used in the smart adjustment strategy to determine where parameter randomization is needed.

Smart Adjustment Strategy

The smart adjustment strategy was introduced to guide the evolution strategy based on the sign of the error. This strategy takes into account whether the error between real and simulated trajectories is positive or negative.

1. When the error is positive (indicating too much friction in the simulation), the strategy selectively decreases friction for certain parameters while increasing others to reduce the overall friction.

2. Conversely, when the error is negative (indicating too little friction in the simulation), the strategy adjusts the parameters to increase friction selectively.

3. This selective adjustment aims to address specific aspects of friction behaviour, allowing for targeted improvements.

The smart adjustment strategy helped reach an optimal solution faster, adapting the parameters search based on the observed errors.

Worse Counter

To avoid getting stuck in local minima during the optimization process, a "worse counter" mechanism was implemented. This counter monitors the consecutive iterations where the new parameter set results in a worse (higher) error compared to the previous one. If the counter is bigger than a threshold, the smart adjustment strategy is ignored, allowing the randomization in both directions independently from the value of the error, this is especially useful when we are in a local minimum and no improvement was found due to the forced directional randomization. If the worse counter surpasses a second predefined threshold, the scaling factor for parameter adjustments becomes more aggressive, allowing for larger perturbations. The worse counter is reset when a new better set of parameters is found. This mechanism introduces controlled exploration during the optimization process, enhancing the algorithm's ability to escape local minima.

The combination of the smart adjustment strategy and the worse counter contributes to the adaptability and robustness of the evolution strategy in optimizing friction and inertia parameters for the Panda robot simulation.

3.2.2 Trajectory Generation and Comparison

To evaluate the effectiveness of the optimized parameters, trajectories were generated from the real robot and compared with simulated trajectories. Trajectories from the real Panda robot were recorded by applying various static torques to each joint. The resulting trajectories were stored and used as a reference for comparison. In particular by applying the following torques to the following joints:

Joint	Torques Applied
Joint1	-2, -1, 1, 2
Joint2	-1.5, -1, 1.5
Joint3	-1.5, -1, 1, 1.5
Joint4	-1.5, -1, 1, 1.5
Joint5	-1.3, -1, 1, 1.3
Joint6	-0.6, -0.5, 0.5, 0.6
Joint7	-0.6, -0.5, 0.4, 0.5, 0.65

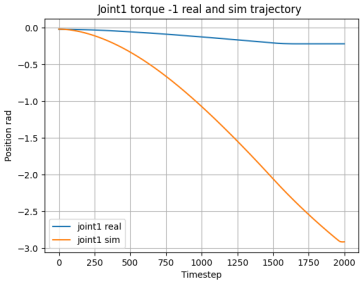
Table 3.3: Torques Applied for Trajectory Generation.

We obtained a total of 28 trajectories. Before randomizing friction and inertia parameters, we compared the real-world trajectories with their corresponding simulated trajectories with the friction and inertia values proposed on [23] for the inertia and [21] for the friction. The purpose of this comparison was to observe the impact of the optimization algorithm after obtaining new parameters.

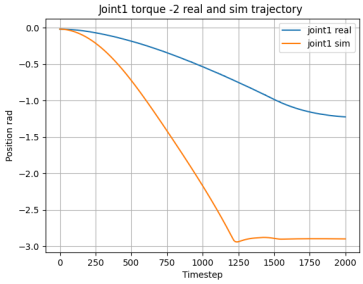
Considering the trade-off between computational time and accuracy, the number of trajectories generated was carefully considered. While having more trajectories generally improves evaluation comprehensiveness, we opted for a balance. Approximately four trajectories per task were selected for analysis, providing a meaningful representation of the robot's behaviour without significantly increasing computational demands.

Trajectory Comparison Plots

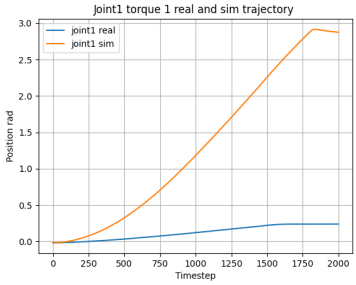
The following plots depict the trajectories of selected joints before any randomization of friction and inertia parameters. The blue lines represent the real-world trajectories obtained from the physical Panda robot, while the orange lines represent the corresponding simulated trajectories.



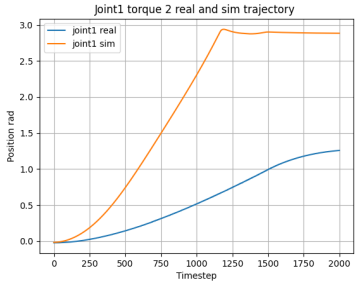
(a) Torque = -1 N/m



(b) Torque = -2 N/m

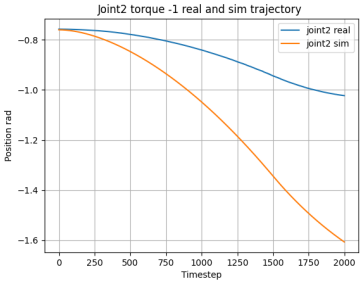


(c) Torque = 1 N/m

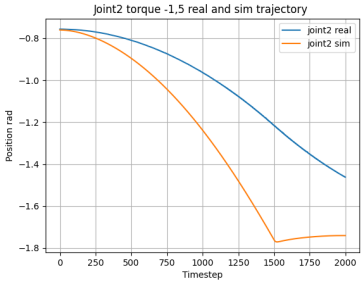


(d) Torque = 2 N/m

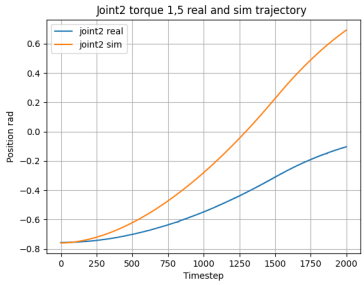
Figure 3.18: Trajectory comparisons for Joint 1 with initial parameters.



(a) Torque = -1 N/m

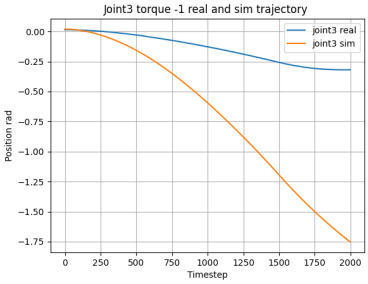


(b) Torque = -1.5 N/m

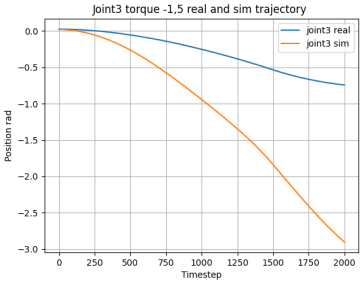


(c) Torque = 1.5 N/m

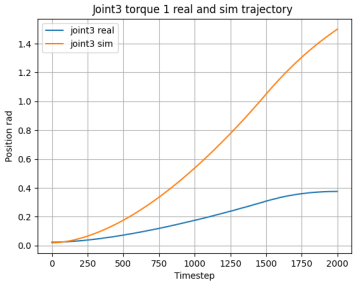
Figure 3.19: Trajectory comparisons for Joint 2 with initial parameters.



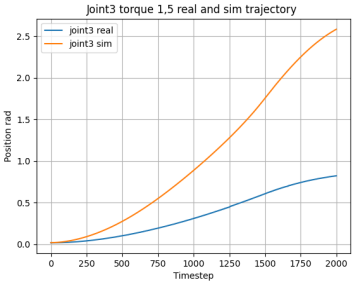
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

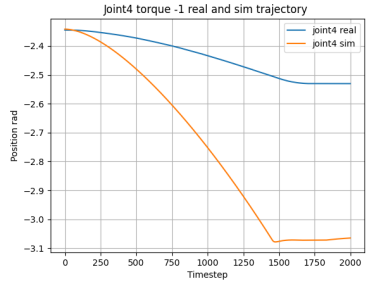


(c) Torque = 1 N/m

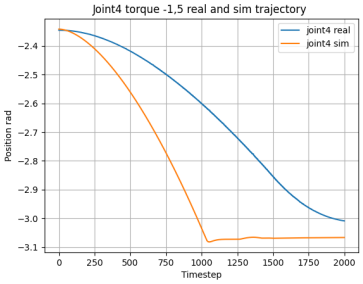


(d) Torque = 1.5 N/m

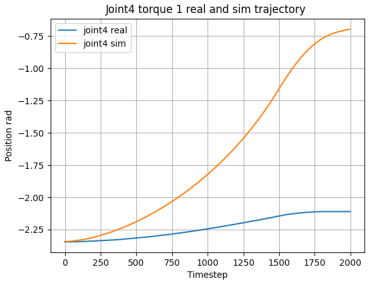
Figure 3.20: Trajectory comparisons for Joint 3 with initial parameters.



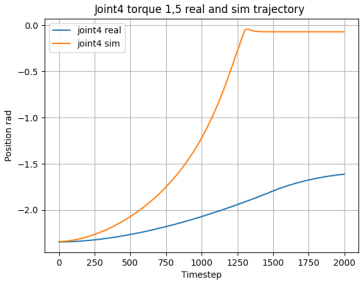
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

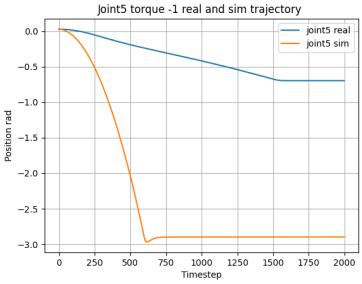


(c) Torque = 1 N/m

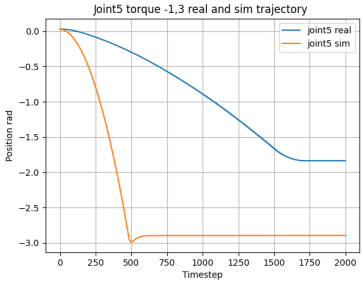


(d) Torque = 1.5 N/m

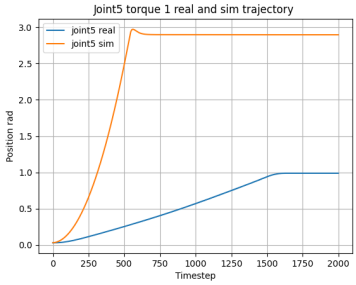
Figure 3.21: Trajectory comparisons for Joint 4 with initial parameters.



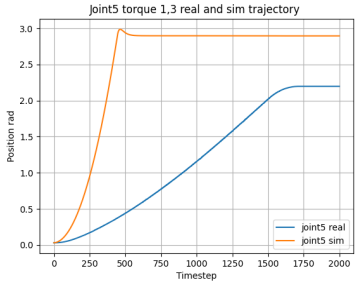
(a) Torque = -1 N/m



(b) Torque = -1.3 N/m

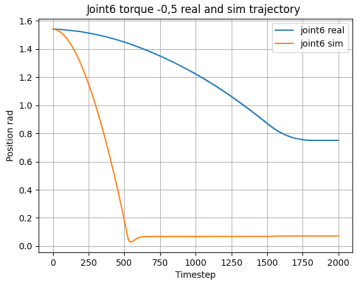


(c) Torque = 1 N/m

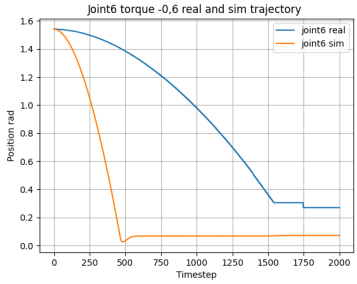


(d) Torque = 1.3 N/m

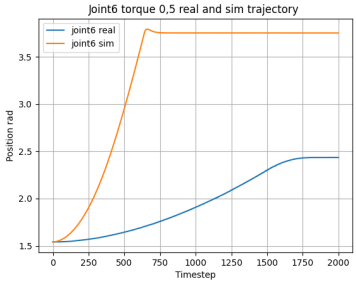
Figure 3.22: Trajectory comparisons for Joint 5 with initial parameters.



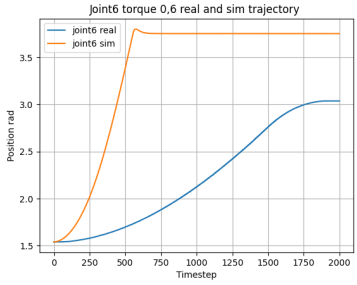
(a) Torque = -0.5 N/m



(b) Torque = -0.6 N/m

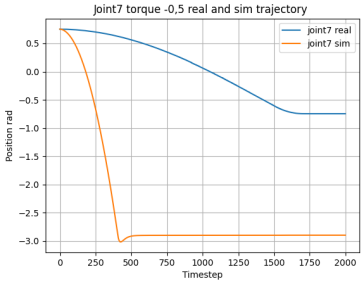


(c) Torque = 0.5 N/m

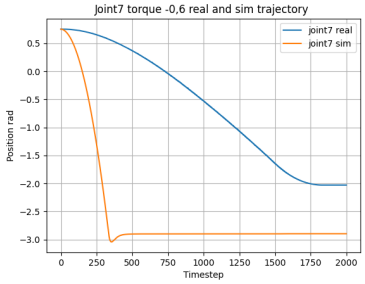


(d) Torque = 0.6 N/m

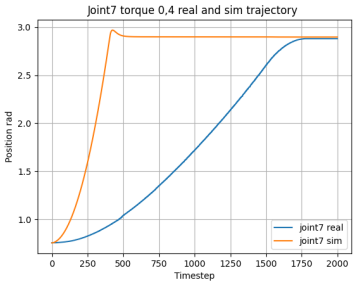
Figure 3.23: Trajectory comparisons for Joint 6 with initial parameters.



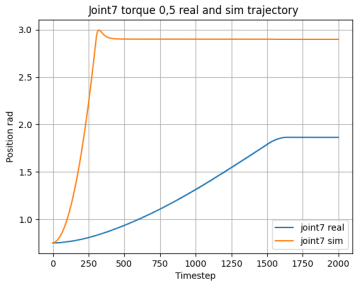
(a) Torque = -0.5 N/m



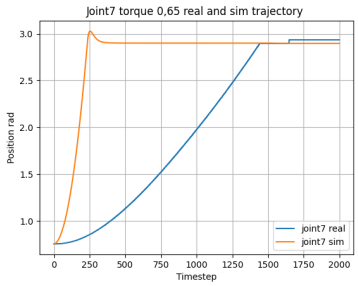
(b) Torque = -0.6 N/m



(c) Torque = 0.4 N/m



(d) Torque = 0.5 N/m



(e) Torque = 0.65 N/m

Figure 3.24: Trajectory comparisons for Joint 7 with initial parameters.

It's evident from the plots that the friction in simulation is significantly lower compared to reality. This discrepancy is apparent across all joints, where the trajectories for positive and negative torques diverge notably from the real ones.

New Parameters

The following table presents the updated friction parameters for each joint after optimization. The optimization process aimed to enhance the simulation's fidelity to real-world trajectories.

Joint	Parameter	Old Value	New Value
1	FI_10	0.54615	3.40395
	FI_20	5.1181	9.67118
	FI_30	0.039533	6.748e-05
2	FI_11	0.87224	1.97802
	FI_21	9.0657	28.66575
	FI_31	0.025882	0.00012
3	FI_12	0.64068	2.51351
	FI_22	10.136	9.82965
	FI_32	-0.04607	-0.00050
4	FI_13	1.2794	2.43842
	FI_23	5.5903	18.77921
	FI_33	0.036194	0.00057
5	FI_14	0.83904	2.59531
	FI_24	8.3469	4.19342
	FI_34	0.026226	0.00541
6	FI_15	0.30301	1.19220
	FI_25	17.133	5.74691
	FI_35	-0.021047	-0.00048
7	FI_16	0.56489	1.54853
	FI_26	10.336	1.69810
	FI_36	0.0035526	0.00169

Table 3.4: Comparison of Friction Parameters Before and After Optimization.

The optimization process not only focused on refining friction parameters but also targeted improvements in inertia values to enhance the simulation's accuracy. Table 3.5 showcases a side-by-side comparison of the original and optimized inertia parameters for each link.

Link	**Old Inertia Values**	**New Inertia Values**
1	0.70337, 0.70661, 0.009117, -0.000139, 0.006772, 0.019169	0.5123, 0.5147, 0.0066, -0.0001, 0.0049, 0.0140
2	0.007962, 0.02811, 0.025995, -0.003925, 0.01025, 0.000704	0.01089, 0.03844, 0.03555, -0.00537, 0.01402, 0.00096
3	0.03724, 0.03616, 0.01083, -0.004761, -0.01140, -0.01281	0.04234, 0.04110, 0.01231, -0.00541, -0.01296, -0.01456
4	0.02585, 0.01955, 0.02832, 0.007796, -0.001332, 0.008641	0.02525, 0.01910, 0.02766, 0.007614, -0.001301, 0.00844
5	0.03555, 0.02947, 0.008627, -0.002117, -0.004037, 0.000229	0.04287, 0.03554, 0.01040, -0.00255, -0.00487, 0.00028
6	0.001964, 0.004354, 0.005433, 0.000109, -0.001158, 0.000341	0.00148, 0.00329, 0.00410, 0.000082, -0.00087, 0.00026
7	0.01252, 0.01003, 0.004815, -0.000428, -0.001196, -0.000741	0.01482, 0.01188, 0.00570, -0.00051, -0.00142, -0.00088

Table 3.5: Comparison of Old and New Inertia Values.

Joint	Initial Error	Final Error
1	188.8488	3.0121
2	31.5475	1.7076
3	96.0152	3.1312
4	77.5024	2.1821
5	258.6358	10.8730
6	156.5996	15.6212
7	288.9131	54.1302

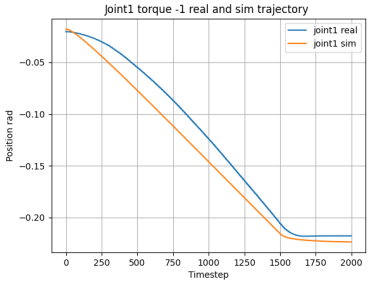
Table 3.6: Comparison of Initial and Final Errors

The optimization process led to significant adjustments in the friction parameters. Notably, these changes were drastic, with some friction coefficients experiencing several-fold increases or decreases. The inertia parameters presented in Table 3.5 for each joint correspond to MuJoCo's full inertia matrix (`fullinertia`). This matrix is expressed using six numbers in the following order: $M(1,1)$, $M(2,2)$, $M(3,3)$, $M(1,2)$, $M(1,3)$, and $M(2,3)$. It is a symmetric 3-by-3 matrix. The compiler computes the eigenvalue decomposition of M and adjusts the frame orientation and diagonal inertia accordingly. The values in the table represent both the old and new inertia parameters for each joint, providing information about the modifications made during the optimization process.

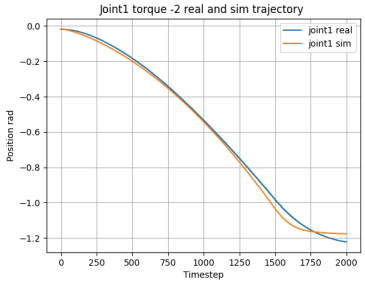
Unlike the friction parameters, the changes in inertia parameters are more moderate. While still significant, they do not exhibit the same level of magnitude of variation as observed in the friction values.

Trajectory Comparison

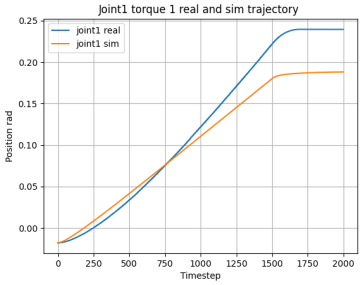
The following figures illustrate the comparison between real and simulated trajectories for a specific joint and torque condition after the optimization algorithm.



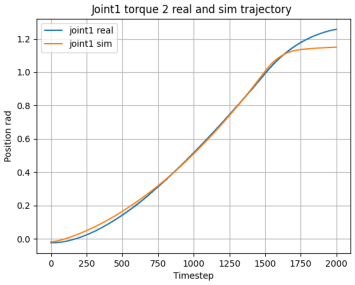
(a) Torque = -1 N/m



(b) Torque = -2 N/m

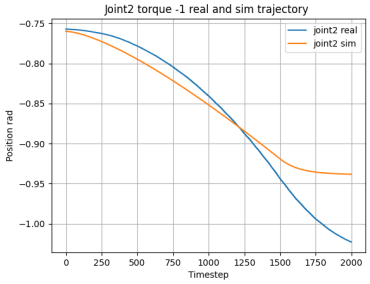


(c) Torque = 1 N/m

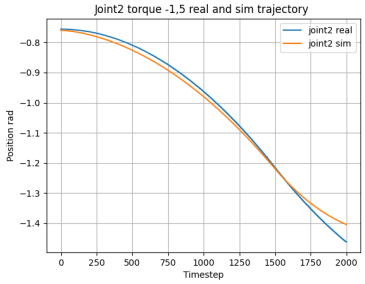


(d) Torque = 2 N/m

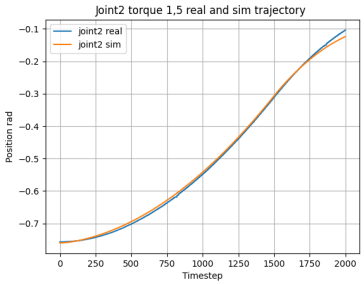
Figure 3.25: Trajectory comparisons for Joint 1 with new parameters.



(a) Torque = -1 N/m

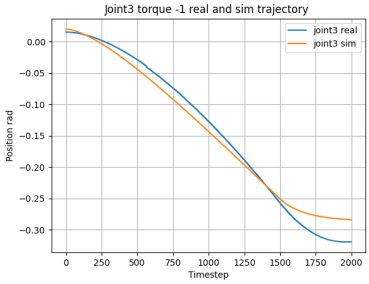


(b) Torque = -1.5 N/m

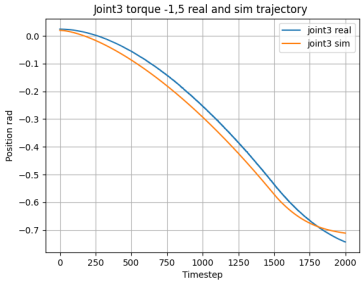


(c) Torque = 1.5 N/m

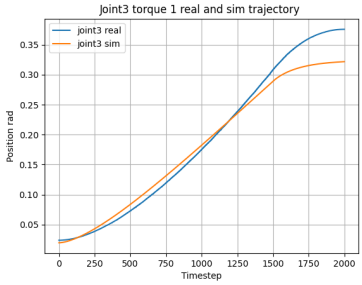
Figure 3.26: Trajectory comparisons for Joint 2 with new parameters.



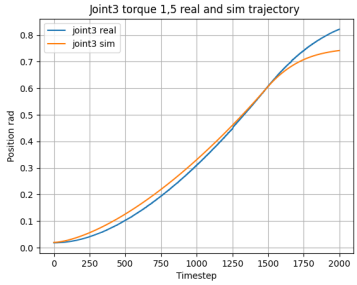
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

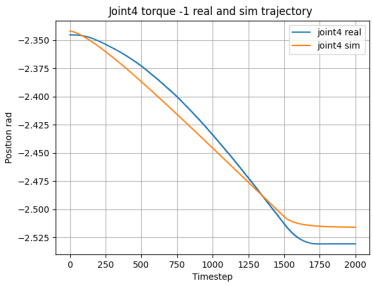


(c) Torque = 1 N/m

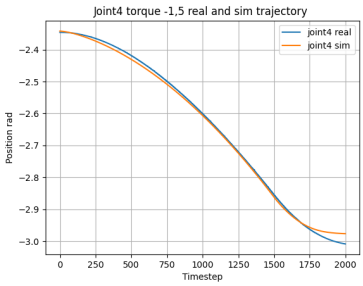


(d) Torque = 1.5 N/m

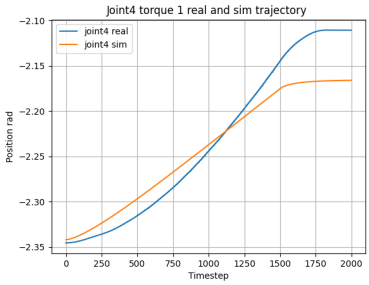
Figure 3.27: Trajectory comparisons for Joint 3 with new parameters.



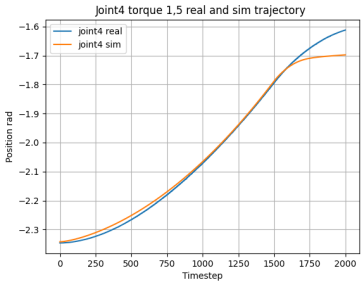
(a) Torque = -1 N/m



(b) Torque = -1.5 N/m

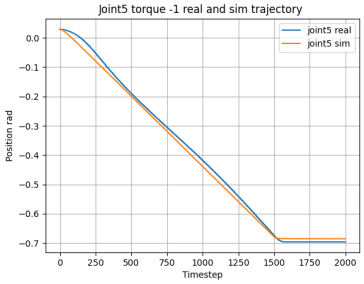


(c) Torque = 1 N/m

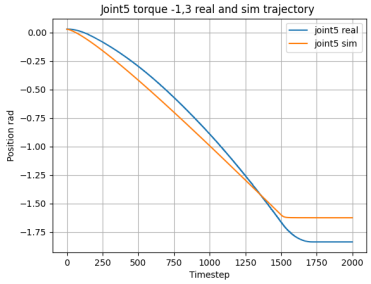


(d) Torque = 1.5 N/m

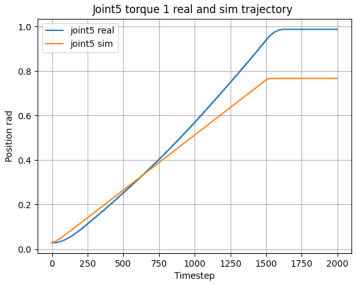
Figure 3.28: Trajectory comparisons for Joint 4 with new parameters.



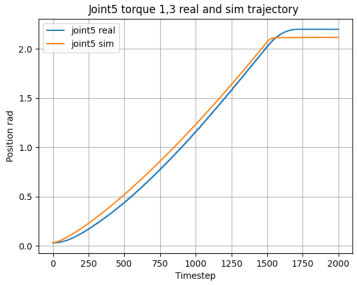
(a) Torque = -1 N/m



(b) Torque = -1.3 N/m

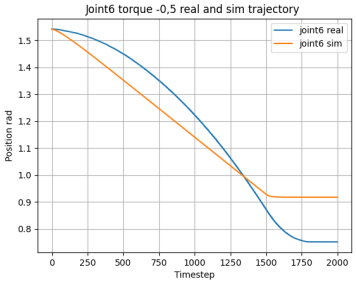


(c) Torque = 1 N/m

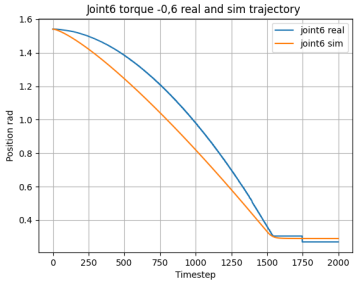


(d) Torque = 1.3 N/m

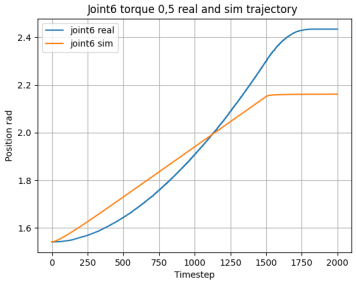
Figure 3.29: Trajectory comparisons for Joint 5 with new parameters.



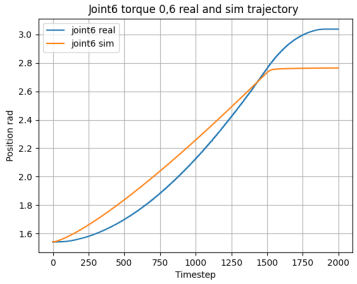
(a) Torque = -0.5 N/m



(b) Torque = -0.6 N/m

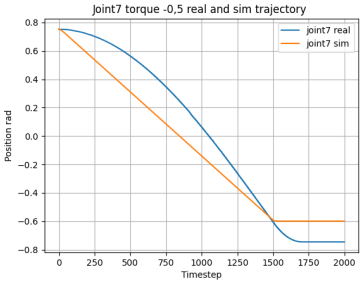


(c) Torque = 0.5 N/m

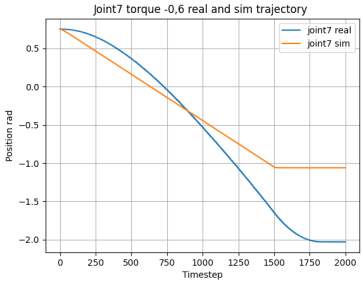


(d) Torque = 0.6 N/m

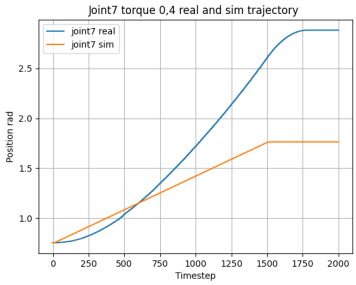
Figure 3.30: Trajectory comparisons for Joint 6 with new parameters.



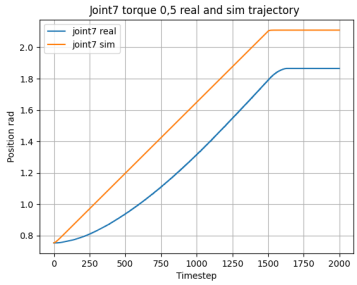
(a) Torque = -0.5 N/m



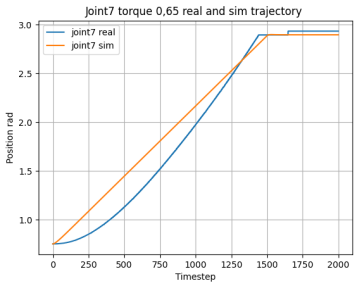
(b) Torque = -0.6 N/m



(c) Torque = 0.4 N/m



(d) Torque = 0.5 N/m



(e) Torque = 0.65 N/m

Figure 3.31: Trajectory comparisons for Joint 7 with new parameters.

Evolution Strategy Performance

The evolution strategy demonstrated its ability to adapt friction and inertia parameters to improve trajectory matching. The final parameters resulted in reduced errors and more accurate simulation results.

3.2.3 Overall Impact

The initial total errors for all joints were relatively high (as visible in Table 3.6), indicating a significant deviation between the simulated and real trajectories. After the optimization, the final total errors for all joints significantly decreased. This suggests that the algorithm successfully identified better parameter values that align the simulated trajectories more closely with the real trajectories. The reduction in total error is a positive outcome, indicating improved accuracy in the simulation.

However, it's worth mentioning that despite the substantial error reduction, there might still be room for further improvement. Fine-tuning or employing more sophisticated optimization techniques could potentially lead to even better alignment between simulated and real trajectories.

Chapter 4

Sim-to-Real Transfer

This chapter provides an overview of the Sim-to-Real Transfer and introduces the context for discussing the results obtained with domain randomization. The goal is to assess the robustness of learned policies under diverse conditions by introducing randomization in the system's physical parameters.

4.1 Results of Sim-to-Real Transfer Experiments with Domain Randomization

In this section, we discuss the outcomes of Sim-to-Real Transfer experiments with the implementation of domain randomization. The aim is to investigate the robustness of the learned policies under varying conditions by introducing randomization in the physical parameters of the system.

4.1.1 Domain Randomization Actuated

The domain randomization is integrated into the environment to enhance the transferability of the learned policies. Specifically, after each iteration of the reinforcement learning process, the friction and inertia parameters are randomized around their initial optimal values. The randomization involves varying friction within a range of $\pm 10\%$ and inertia within a range of $\pm 5\%$ of their initial values. This process is repeated after every episode, providing a

diverse set of training scenarios.

Reinforcement Learning Performance

Graphical representations of the reinforcement learning performance are shown in Figure 4.1 and Figure 4.2. These graphs demonstrate the learning progress of the TQC algorithm during the grid search, both before and after implementing domain randomization.

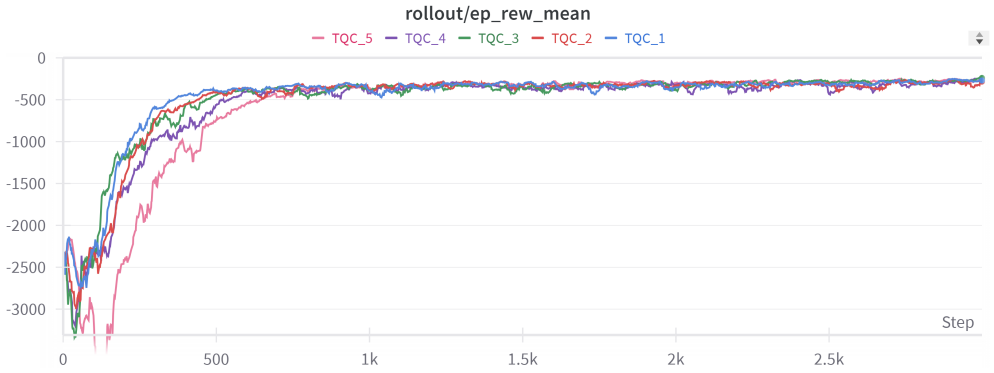


Figure 4.1: TQC performance during the grid search without domain randomization.

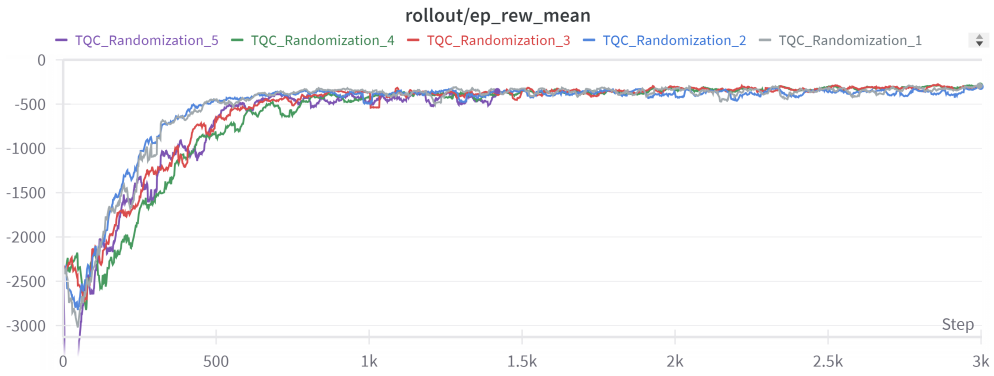


Figure 4.2: TQC performance during the grid search with domain randomization.

The visual representations collectively offer valuable insights into the influence of domain randomization on Sim-to-Real transfer experiments, revealing the robustness and adaptability of the learned policies under diverse real-world conditions. Notably, the mean reward per episode, as observed in the confrontation of the final training episodes of the two most successful runs presented in Figure 4.3, indicates that the randomized scenarios exhibit a slight performance dip compared to their non-randomized counterparts. This marginal difference underscores the efficacy of domain randomization in sustaining competitive performance despite the inherent variations introduced during the training process.

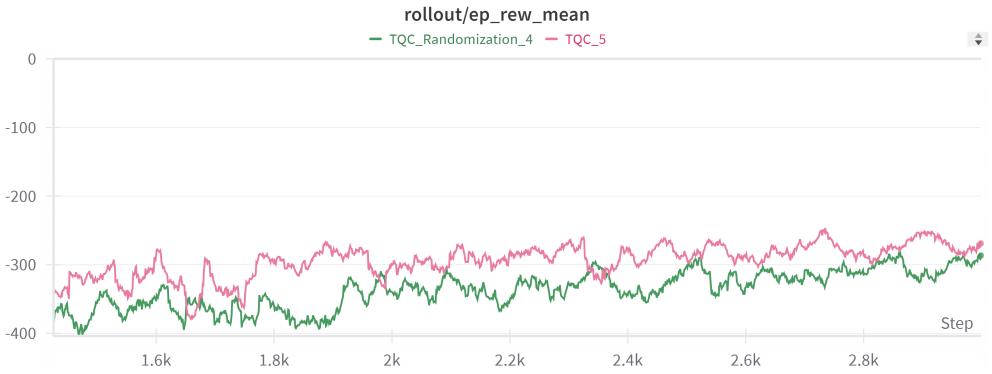


Figure 4.3: Comparison of mean reward per episode between scenarios with and without domain randomization.

4.1.2 Sim and Real Model Testing

In this section, we evaluate the performance of the newly trained model, which was trained with domain randomization, in different environments. The evaluation encompasses testing in MuJoCo, Gazebo, and real-world scenarios.

MuJoCo Evaluation

The model's performance is first evaluated in the MuJoCo simulation environment. Figure 4.4 presents the results of the evaluation, in particular, the plots

represent the value of the reward obtained at each step during the evaluation.

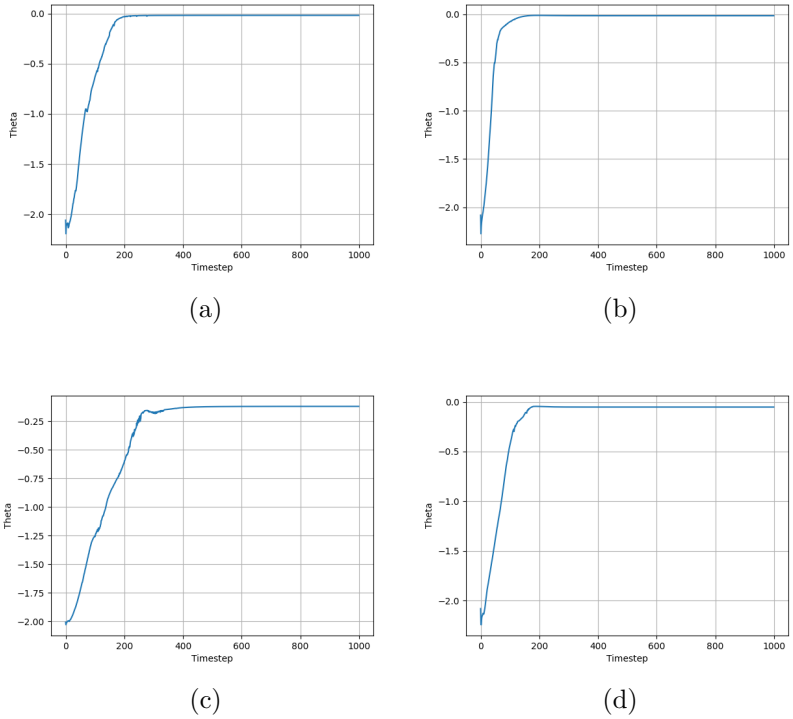
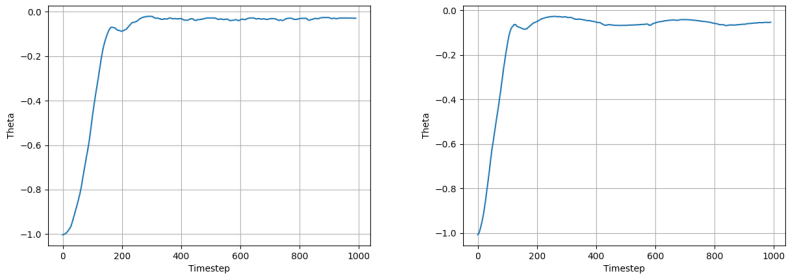


Figure 4.4: Results for MuJoCo.

The evaluation in the MuJoCo simulation environment demonstrates the effectiveness of the trained model. The plots of the rewards highlight the model's ability to efficiently navigate and manipulate the robotic arm getting almost null rewards for some positions. The end effector successfully reaches the randomized target and maintains its position steadily throughout the remaining timesteps.

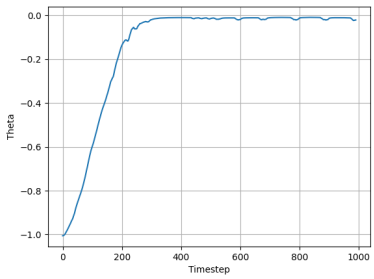
Gazebo Evaluation

Next, we assess the model's performance in the Gazebo simulation environment. Figure 4.5 showcases the reward plots obtained during the Gazebo evaluation.

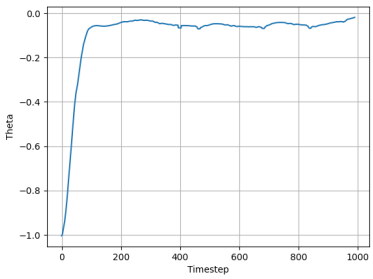


(a)

(b)



(c)



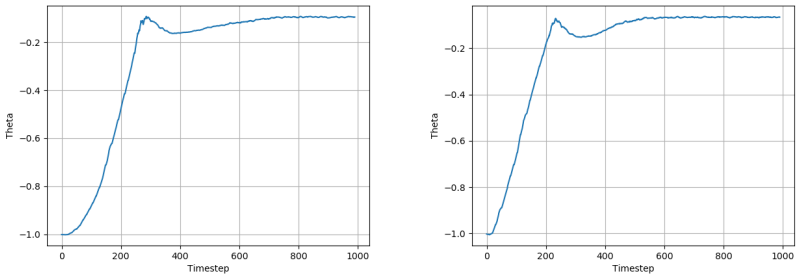
(d)

Figure 4.5: Results for Gazebo.

The evaluation in the Gazebo simulation, being a simulator closer to reality, reveals optimal results. Figure 4.5 demonstrates that the model performs exceptionally well, exhibiting only minor fluctuations in rewards as the end effector precisely reaches the randomized target as shown in Figure 4.5b and Figure 4.5d. The slight variations observed in the rewards are attributed to subtle movements of the end effector upon reaching the target. Overall, these results suggest a high level of accuracy and stability in the model's performance, indicating a successful transfer of learned behaviours to a simulation environment more representative of real-world conditions.

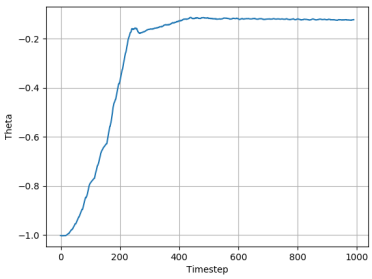
Real-world Evaluation

It's important to note that the real-world setup differs from the simulation environments. Unlike in simulations, the policy's actions are no longer modified by subtracting dynamic friction. Instead, the policy's actions are directly summed with the pre-determined gravity compensation offset values.

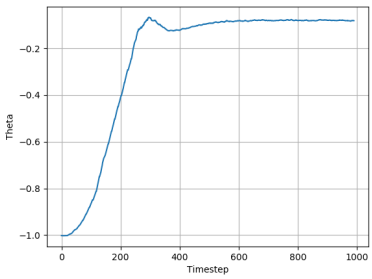


(a)

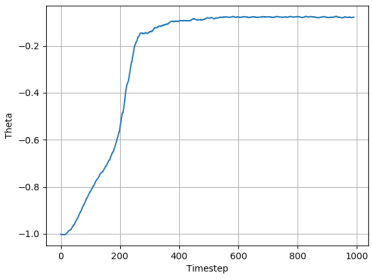
(b)



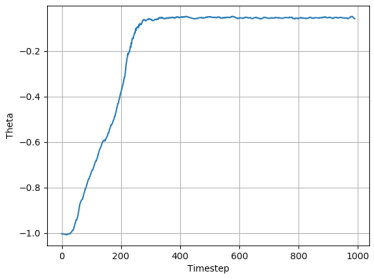
(c)



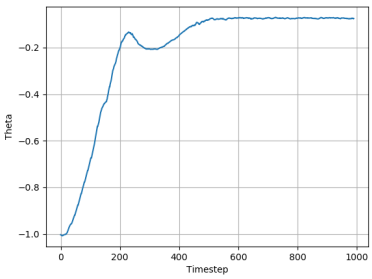
(d)



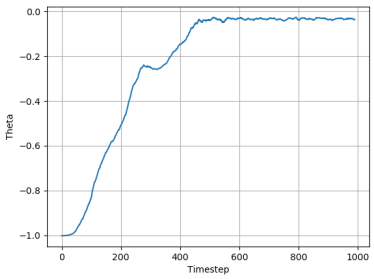
(e)



(f)



(g)



(h)

Figure 4.6: Results in real scenario.

The plots illustrating the real-world evaluation indicate that the model trained with domain randomization performs remarkably well in the actual physical environment. The end effector consistently approaches and maintains its position near the randomized goal. Notably, some minor overshooting is observed in reaching the target, as depicted in Figure 4.6a, 4.6b and Figure 4.6d. Despite these slight deviations, the overall performance demonstrates the effectiveness of the trained model in a real-world setting.

The evaluation across different environments shows the generalizability and robustness of the trained model, allowing possible applications in real-world settings.

Chapter 5

Conclusion

The research findings highlighted the challenges encountered during the transfer from MuJoCo to Gazebo, particularly in handling damping and friction discrepancies. The study emphasized the importance of simulation effectiveness in Gazebo for real-world applicability. The evaluation in different environments demonstrated the model's generalizability and robustness, showcasing its potential in real-world settings. The impact of domain randomization on the Sim-to-Real Transfer experiments was evident, revealing the adaptability and performance of the learned policies under diverse conditions. Although significant error reduction was achieved through optimization, further improvements could be explored through fine-tuning or advanced optimization techniques to enhance alignment between simulated and real trajectories.

5.1 Contributions to Sim-to-Real Transfer

This thesis delves into the challenges of transferring simulated robotic control policies to the real world, focusing on torque-controlled robotic arms. Using reinforcement learning techniques, the research addresses issues related to friction compensation, parameter optimization, and domain randomization, aiming to enhance the adaptability of robotic systems in varied environments.

One significant aspect of this research is the in-depth exploration of parameter estimation, specifically dynamic friction and inertia. The conducted

experiments and analyses showed how to accurately estimate these crucial parameters, emphasizing their impact on simulated and real-world dynamics. These techniques contribute not only to robust control policies but also to a better understanding of challenges posed by unknown parameters.

The successful implementation of parameter optimization strategies demonstrated in this work is central to improving the adaptability and performance of robotic systems in real-world scenarios. Optimizing inertia and dynamic friction helps bridge the reality gap, adding a valuable dimension to creating resilient and versatile robotic systems.

Beyond discovery, this work highlights the practical implications of sim-to-real transfer challenges, emphasizing the importance of developing robust strategies for generalizing robotic arms effectively. The research advances the state-of-the-art in robotic control, laying a foundation for future efforts to enhance performance and adaptability in real-world settings.

The detailed investigation into simulation environments, training methodologies, and adaptation strategies provides a comprehensive understanding of sim-to-real transfer complexities for torque-controlled robotic arms. By analyzing existing literature, introducing innovative adaptations, and sharing empirical results, this thesis enriches the discourse on the efficacy of sim-to-real transfer methodologies.

The implications of this work extend towards creating more efficient and adaptable robotic systems capable of navigating diverse real-world environments with precision and reliability.

5.2 Future Research Directions

Moving forward, specific avenues for future research can be explored based on the identified limitations and potential improvements. Some potential directions include:

- Further investigate the impact of domain randomization on sim-to-real transfer across a broader range of robotic applications to assess its effectiveness and robustness in diverse scenarios.

- Explore new strategies for adapting simulation-trained models to complex real-world environments with unknown parameters, focusing on enhancing adaptability and generalization capabilities.
- Evaluate the scalability of the proposed methodology to different robotic platforms and tasks to enhance its applicability and effectiveness in various real-world settings.
- Explore the potential of reinforcement learning algorithms and hyperparameter optimization techniques to further enhance training effectiveness and generalization capabilities for sim-to-real transfer challenges in robotic control.

By addressing these future research directions, the field of sim-to-real transfer for torque-controlled robotic arms can continue to evolve and innovate, leading to more efficient and adaptable robotic systems for a wide range of applications.

Bibliography

[1] "Robot articolato Panda". In: (). URL: https://www.directindustry.it/prod/franka-emika/product-187686-1906234.html.

[2] Stanford University. *https://www.ros.org/*. URL: https://www.ros.org/.

[3] "MuJoCo". In: (). URL: https://mujoco.org/.

[4] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. "Sim-to-real transfer in deep reinforcement learning for robotics: a survey". In: *2020 IEEE symposium series on computational intelligence (SSCI)*. IEEE. 2020, pp. 737–744.

[5] Xue Bin Peng et al. "Sim-to-real transfer of robotic control with dynamics randomization". In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2018, pp. 3803–3810.

[6] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[7] Tymoteusz Lindner, Andrzej Milecki, and Daniel Wyrwał. "Positioning of the robotic arm using different reinforcement learning algorithms". In: *International Journal of Control, Automation and Systems* 19 (2021), pp. 1661–1676.

[8] Yuqing Du et al. "Auto-Tuned Sim-to-Real Transfer". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 1290–1296. DOI: 10.1109/ICRA48506.2021.9562091.

[9] Yevgen Chebotar et al. “Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience”. In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8973–8979. DOI: 10.1109/ICRA.2019.8793789.

[10] *Gymnasium documentation*. 2024. URL: https://gymnasium.farama.org/.

[11] “Robot and interface specifications”. In: (). URL: https://frankaemika.github.io/docs/control_parameters.html.

[12] “Stable-Baselines3”. In: (2024). URL: https://stable-baselines3.readthedocs.io/en/master/index.htm/.

[13] “Deep Deterministic Policy Gradient”. In: (2024). URL: https://stable-baselines3.readthedocs.io/en/master/modules/ddpg.html.

[14] “Soft Actor-Critic”. In: (2024). URL: https://stable-baselines3.readthedocs.io/en/master/modules/sac.html.

[15] “Twin Delayed DDPG”. In: (2024). URL: https://stable-baselines3.readthedocs.io/en/master/modules/td3.html.

[16] “Stable Baselines3 Contrib”. In: (2024). URL: https://sb3-contrib.readthedocs.io/en/master/index.html.

[17] “Truncated Quantile Critics”. In: (2024). URL: https://sb3-contrib.readthedocs.io/en/master/modules/tqc.html.

[18] Jakob Hollenstein et al. “Action noise in off-policy deep reinforcement learning: Impact on exploration and performance”. In: *arXiv preprint arXiv:2206.03787* (2022).

[19] Stephen Dankwa and Wenfeng Zheng. “Twin-delayed ddpg: A deep reinforcement learning technique to model a continuous movement of an intelligent robot agent”. In: *Proceedings of the 3rd international conference on vision, image and signal processing*. 2019, pp. 1–5.

[20] Arsenii Kuznetsov et al. “Controlling overestimation bias with truncated mixture of continuous distributional quantile critics”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5556–5566.

[21] Claudio Gaz et al. *Dynamic Identification of the Franka Emika Panda Robot with Retrieval of Feasible Parameters Using Penalty-Based Optimization*. Supplementary material. Oct. 2019. URL: https://inria.hal.science/hal-02265294.

[22] “joint_state_controller”. In: (2024). URL: https://wiki.ros.org/joint_state_controller.

[23] “franka_panda_description”. In: (2024). URL: https://github.com/justagist/franka_panda_description.