Costa Rica Institute of Technology

School of Mechatronics Engineering

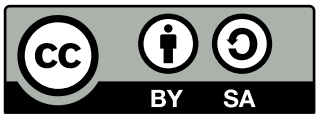
Control System for Safe and Optimal Trajectory Execution in Robots

Graduation Project Report to Opt for

the Title of Mechatronics Engineer

Jeaustin Calderón Quesada

Cartago, June 24, 2025



I hereby declare that the present Graduation Project has been carried out entirely by my person, using and applying literature referring to the subject and introducing my own knowledge.

Cases in which I used bibliography, I proceeded to indicate the sources by the respective bibliographic citations. Consequently, I assume full responsibility for the work presented here and for all the content of the corresponding final report.

Jeaustin Calderón

Digitally signed by Jeaustin Calderón
Date: 2025.07.01 14:07:12 +02'00'

Jeaustin Calderón Quesada

Cartago, June 24, 2025

ID: 1-1818-0302

INSTITUTO TECNOLÓGICO DE COSTA RICA
PROGRAMA DE LICENCIATURA EN INGENIERÍA MECATRÓNICA
PROYECTO FINAL DE GRADUACIÓN
ACTA DE APROBACIÓN

El profesor asesor del presente trabajo final de graduación, indica que el documento presentado por el estudiante cumple con las normas establecidas por el programa de Licenciatura en Ingeniería Mecatrónica del Instituto Tecnológico de Costa Rica para ser defendido ante el jurado evaluador, como requisito final para aprobar el curso Proyecto Final de Graduación y optar así por el título de Ingeniero(a) en Mecatrónica, con el grado académico de Licenciatura.

Estudiante: Jeaustin Calderón Quesada

Proyecto: Control System for Safe and Optimal Trajectory Execution in Robots

JUAN LUIS CRESPO MARIÑO (FIRMA)
PERSONA FISICA, CPF-08-0113-0166.
Fecha declarada: 27/06/2025 02:30:40 PM
Esta es una representación gráfica únicamente,
verifique la validez de la firma.

Dr. Ing. Juan Luis Crespo Mariño
Asesor

Cartago, 24 de junio de 2025.

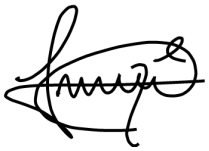
INSTITUTO TECNOLÓGICO DE COSTA RICA
PROGRAMA DE LICENCIATURA EN INGENIERÍA MECATRÓNICA
PROYECTO FINAL DE GRADUACIÓN
ACTA DE APROBACIÓN

Proyecto final de graduación defendido ante el presente jurado evaluador como requisito para optar por el título de Ingeniero(a) en Mecatrónica con el grado académico de Licenciatura, según lo establecido por el programa de Licenciatura en Ingeniería Mecatrónica, del Instituto Tecnológico de Costa Rica.

Estudiante: Jeaustin Calderón Quesada

Proyecto: Control System for Safe and Optimal Trajectory Execution in Robots

Miembros del jurado evaluador



Digitally signed by FELIPE GERARDO MEZA OBANDO (FIRMA)
Date: 2025.06.25 11:21:54 -06'00'

Dr. Ing. Felipe Meza Obando

Jurado

TEC | Tecnológico de Costa Rica

Firmado digitalmente por YEINER ARIAS ESQUIVEL (FIRMA)
Fecha: 2025.06.24 14:17:41 -06'00'

Dr. Ing. Yeiner Arias Esquivel

Jurado

Los miembros de este jurado dan fe de que el presente proyecto final de graduación ha sido aprobado y cumple con las normas establecidas por el programa de Licenciatura en Ingeniería Mecatrónica.

Cartago, 24 de junio de 2025.

Abstract

This work presents a learning-based control framework for torque-driven robotic manipulators operating under physical constraints. The proposed system is trained using an innovative approach that combines concepts from Proximal Policy Optimization (PPO) and Generative Adversarial Networks (GANs). The controller's behavior is shaped to imitate reference trajectories generated by an Model Predictive Controller (MPC). The architecture is designed to handle physical constraints through complementary strategies that enhance safety and generalization.

The system is evaluated on a planar two-degree-of-freedom robot using realistic torque constraints, under two different actuation limits. Performance is assessed in terms of the mean and standard deviation of position and velocity tracking errors, inference time and its variability, and compliance with physical constraints.

This work demonstrates that the proposed constraint-aware deep learning controller achieves inference times over 17 times faster than the MPC baseline, enabling real-time deployment with minimal computational resources. The results confirm that the learned policy is both efficient and robust, offering a viable and competitive alternative to classical model-based controllers, especially in applications requiring fast, safe, and generalizable control under real-world constraints.

Keywords: Artificial Intelligence, Automatic Control, Machine Learning, Real-Time Control, Reinforcement Learning, Robotics, Simulation

Resumen

Este trabajo presenta un marco de control basado en aprendizaje para manipuladores robóticos accionados por torque que operan bajo restricciones físicas. El sistema propuesto se entrena mediante un enfoque innovador que combina conceptos de Optimización Proximal de la Política (PPO) y Redes Generativas Antagónicas (GANs). El comportamiento del controlador se ajusta para imitar trayectorias de referencia generadas por un controlador de modelo predictivo (MPC). La arquitectura está diseñada para manejar restricciones físicas mediante estrategias complementarias que refuerzan la seguridad y la generalización.

El sistema se evalúa en un robot planar de dos grados de libertad utilizando restricciones de torque realistas, bajo dos diferentes límites de actuación. El desempeño se evalúa en términos del promedio y la desviación estándar de los errores de seguimiento de posición y velocidad, el tiempo de inferencia y su variabilidad, y el cumplimiento de las restricciones físicas.

Este trabajo demuestra que el controlador propuesto, basado en aprendizaje profundo con conciencia de restricciones, alcanza tiempos de inferencia más de 17 veces más rápidos que la línea base basada en MPC, lo que permite su implementación en tiempo real con recursos computacionales mínimos. Los resultados confirman que la política aprendida es tanto eficiente como robusta, ofreciendo una alternativa viable y competitiva frente a los controladores clásicos basados en modelos, especialmente en aplicaciones que requieren control rápido, seguro y generalizable bajo restricciones del mundo real.

Palabras clave: Aprendizaje Automático, Aprendizaje por Refuerzo, Simulación, Control Automático, Control en Tiempo Real , Inteligencia Artificial, Robótica

This work is dedicated first and foremost to my parents. To my mother, for being a constant source of support throughout my life, always believing in my goals and dreams. To my father, for teaching me the values and principles that helped shape the person I am today and guided me to this point.

I dedicate this thesis to my grandmother Au, who showed me the importance of hard work and has always been there for me, offering her support no matter the circumstances.

To Alex, for the lessons he taught me, without knowing how meaningful they would become, and for being present in ways that made a lasting difference.

To my sister Sofía, may she always know that she can count on me as unconditionally as I've counted on her.

And finally, to my partner Matilda, whose support during every stage of this thesis meant more than words can express. Thank you for encouraging me when I doubted myself and lifting my spirits when things did not go as planned.

dedication

Acknowledgments

I would like to express my sincere gratitude to the Applied Mechanics Laboratory at the Technical University of Munich for providing the resources and environment that made this work possible. In particular, I am deeply thankful to my advisors, M.Sc. Tanmay Goyal and M.Sc. Tomas Slimak, for their valuable time, insightful feedback, and continuous guidance. I am also grateful to Prof. Dr. Daniel J. Rixen for the opportunity to carry out my thesis at such a prestigious chair.

I also extend my heartfelt thanks to Dr. Juan Luis Crespo Mariño for his unwavering dedication, constant availability, and exceptional support throughout every stage of this graduation project.

Jeaustin Calderón Quesada

Cartago, June 24, 2025

Contents

List of Figures ii

List of Tables v

List of Symbols and Abbreviations ix

1 Introduction 1
- 1.1 Objectives and Document Structure . . . 2

2 Theoretical Framework 5
- 2.1 Overview of Robotics . . . 5
  - 2.1.1 Robotic Manipulator . . . 5
  - 2.1.2 Kinematics . . . 8
  - 2.1.3 Dynamics . . . 9
  - 2.1.4 Trajectory planning . . . 13
- 2.2 Control theory . . . 17
  - 2.2.1 Optimal Control . . . 17
- 2.3 Artificial Intelligence (AI) . . . 19
  - 2.3.1 Machine Learning (ML) . . . 19
  - 2.3.2 Artificial Neural Networks (ANNs) . . . 20
  - 2.3.3 Generative Adversarial Networks (GANs) . . . 22
  - 2.3.4 Reinforcement Learning (RL) . . . 25
  - 2.3.5 Proximal Policy Optimization (PPO) Algorithm . . . 27
- 2.4 Learning-Based Controllers . . . 30

3 Design of a Control System for Execution of Trajectories in Robots 33
- 3.1 Design Methodology . . . 33
  - 3.1.1 Stage 1: Initial Evaluation . . . 34
  - 3.1.2 Stage 2: Needs Identification . . . 35
  - 3.1.3 Stage 3: Specification Definition . . . 36
  - 3.1.4 Stage 4: Exploration of Options . . . 40
- 3.2 Proof of Concept . . . 61
  - 3.2.1 Simulation and reference setup . . . 61
  - 3.2.2 Framework for Unconstrained Learning . . . 73
  - 3.2.3 Framework for Constrained Learning . . . 79

3.2.4 Hyperparameter Tuning . . . 81
3.3 Economic Feasibility Study . . . 82
3.3.1 Estimated Costs Incurred During the Proof of Concept . . . 83
3.3.2 Available Funding and Investment Opportunities . . . 84
3.3.3 Projected Costs for Industrial-Scale Implementation . . . 86
3.3.4 Economic Appeal of Collaborative Robots in Industry . . . 88
3.3.5 Discussion . . . 90

4 Results and Analysis **95**
4.1 Baseline . . . 95
4.1.1 Unconstrained MPC . . . 96
4.1.2 Constrained MPC . . . 98
4.2 Hyperparameter Tuning . . . 99
4.3 Unconstrained Implementation . . . 103
4.4 Constrained Implementation: Nominal Scenario . . . 105
4.5 Constrained Implementation: Aggressive Scenario . . . 108

5 Conclusions and Recommendations **113**

A Appendix **133**
A.1 Hyperparameter Tuning 1 . . . 133
A.2 FLOPs Estimation . . . 136

List of Figures

1.1 High-level diagram of the solution. 2

2.1 Parts of a robotic arm manipulator with 6 Degrees of Freedom (DoF). 6
2.2 Types of joints for a robotic manipulator. 7
2.3 Kinematics of a robotic manipulator 8
2.4 Task and joint spaces. 14
2.5 Illustration of a sampling-based motion planning method. Sampled trajectories that result in collisions with obstacles are discarded (red points), while a collision-free trajectory is identified as feasible (green points). 15
2.6 Representation of an artificial neural network. 20
2.7 Representation of long-short term memory network. 22
2.8 Representation of a gated recurrent unit. 23
2.9 GAN high-level block diagram. 24
2.10 Generic reinforcement learning setup 25
2.11 Representation of online RL. 28
2.12 Representation of offline RL. 28

3.1 ABB GoFa 15000. 35
3.2 Functional decomposition of the problem. 40
3.3 General block diagram of the trajectory generation pipeline. 62
3.4 Detailed block diagram of the trajectory generation pipeline. 63
3.5 Heatmap of the end-effector position in the generated dataset (200 samples). 66
3.6 Heatmap of the joint positions in the generated dataset (200 samples). . . . 67
3.7 Heatmap of the end-effector position in the generated dataset (5000 samples). 67
3.8 Heatmap of the joint positions in the generated dataset (5000 samples). . . . 68
3.9 Histograms of RMSE of different weights for the cost function of the MPC implementation. 70
3.10 Boxplots of the control effort of different weights for the cost function of the MPC implementation. 71
3.11 Histograms of the control actions for a cost function with weight of 0.01. . . 71
3.12 Histograms of the control actions for a cost function with weight of 0.02. . . 72
3.13 Detailed histograms of the control actions for a cost function with weight of 0.02. 72
3.14 High-level block diagram of the unconstrained learning framework. 74
3.15 Second-level block diagram of the policy rollout block. 75

3.16 Third-level block diagram of the rollout block. 76
3.17 Representation of the observation space. 76
3.18 Architecture of the PPO agent, showing the shared GRU encoder and separate actor-critic branches 79
3.19 Architecture of the discriminator ensemble, featuring a shared GRU and MLP, followed by multiple independent output heads. 80

4.1 Histogram of control actions from the unconstrained MPC under nominal torque limits. 97
4.2 Histogram of control actions from the unconstrained MPC under aggressive torque limits. 97
4.3 Distribution of control actions with nominal constrained MPC 99
4.4 Distribution of control actions with aggressively constrained MPC 100
4.5 Distribution of final losses across different configurations during the first hyperparameter tuning stage. 100
4.6 Loss distribution for the second hyperparameter tuning. 101
4.7 Loss evolution using the original discriminator learning rate (4.12×10^{-5}), showing delayed policy improvement due to slow discriminator convergence. 102
4.8 Loss evolution using an increased discriminator learning rate (5.12×10^{-4}), leading to faster policy convergence. 103
4.9 Rendered snapshots from different episodes of the simulation, showing distinct steps in the execution of trajectory-following tasks. 103
4.10 Histograms of control actions for the unconstrained policy under (a) nominal and (b) aggressive torque constraint bounds. 104
4.11 Histograms of τ_1 and τ_2 for the constrained implementations without clipping. 106
4.12 Histograms of τ_1 and τ_2 for the constrained implementations with clipping. . 106
4.13 Results of the same trajectory for the four different methods of constraint handling. 109
4.14 Histograms of τ_1 and τ_2 for the aggressively constrained implementations without clipping. 110
4.15 Histograms of τ_1 and τ_2 for the aggressively constrained implementations with clipping. 110
4.16 Comparison of tracking performance for the same trajectory using controllers trained with different torque limits. 112

A.1 Scatter plot of the entropy coefficient (*ent_coef*) versus the loss. 134
A.2 Scatter plot of the initial standard deviation (*std_init*) versus the loss. 134
A.3 Boxplot of the hidden dimension size (*hidden_dim*) versus the loss. 135
A.4 Scatter plot of the gradient penalty coefficient (*lambda_gp*) versus the loss. . 135
A.5 Boxplot of the prediction horizon parameter (config/M) versus the loss. . . . 136
A.6 Kernel density estimation of the top 20 values for the prediction horizon parameter (config/M). 136
A.7 Boxplot of the history window length (N) versus the loss. 137
A.8 Boxplot of number of heads of the discriminator (*num_heads*) versus loss. . . 137

A.9 Boxplot of the number of steps per rollout (*num_steps*) versus the loss. . . . 138
A.10 Boxplot of the discriminator batch size (*batch_disc*) versus the loss. 138

List of Tables

3.1 Designed system's needs and their associated metrics . . . 37
3.2 Specifications of the design system (DS) with the corresponding units, marginal value and ideal value. . . . 38
3.3 Comparison of Training Hardware: Google TPU v4, NVIDIA A100, and NVIDIA H100 . . . 45
3.4 Comparison of Training Hardware: GTX 1650, RTX 3070, and RTX 4090 . . 46
3.5 Comparison of Inference Deployment Proposals . . . 51
3.6 Ranking of proposals across key criteria. Lower values indicate better performance. Importance values range from 3 (low) to 5 (high). . . . 60
3.7 Mean and standard deviation of RMSE for different weight values in the MPC cost function. . . . 70
3.8 Statistical summary of position and velocity variables across the dataset. . . 73
3.9 Summary of chosen position, velocity, and torque constraints. . . . 73
3.11 Estimated Costs Incurred During the Proof of Concept . . . 84
3.10 Search space for hyperparameter tuning. . . . 92
3.12 Summary Costs of Industrial-Scale Implementation . . . 93

4.1 Time metrics for the unconstrained MPC implementation based on three repeated trials. . . . 96
4.2 Trajectory tracking metrics for MPC with control effort consideration (weight = 0.01). . . . 97
4.3 Tracking metrics for MPC with joint torque constraints set to [-38, 33] and [-20, 18] for τ_1 and τ_2 respectively. . . . 98
4.4 Time metrics for the MPC constrained implementation . . . 98
4.5 Tracking performance of MPC with stricter torque constraints [-18, 17] for τ_1 and [-12, 12] for τ_2. . . . 99
4.6 Execution time metrics for MPC with strict joint torque limits [-18, 17] and [-12, 12]. . . . 99
4.7 Best hyperparameter configuration selected after hyperparameter tuning. . . 102
4.8 Trajectory tracking metrics for the unconstrained policy trained for 30 million steps. . . . 105
4.9 Average, standard deviation, and maximum inference time per step for the nominal constraints configurations. . . . 105
4.10 Percentage of control actions exceeding nominal torque boundaries. . . . 107

4.11 Trajectory tracking performance for each configuration. 107
4.12 Average, standard deviation, and maximum inference time per step for the nominal constrained configurations. 109
4.13 Percentage of control actions exceeding aggressive torque boundaries. 111
4.14 Trajectory tracking performance for each configuration for aggressive torque boundaries. 111

A.1 Search space for the first iteration of hyperparameter tuning. 133

List of Symbols and Abbreviations

Abreviaciones

ANN	Artificial Neural Networks
API	Application Programming Interface
AutoML	Automated Machine Learning
BIT*	Batch Informed Trees Star
BIT	Batch Informed Trees
CPU	Central Processing Unit
DoF	Degrees of Freedom
DS	Design System
FC	Fully Connected
FLOPs	Floating Point Operations per Second
FP32	32-bit Floating Point
GAIL	Generative Adversarial Imitation Learning
GAN	Generative Adversarial Networks
GP	Gradient Penalty
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HBM	High Bandwidth Memory
HPC	High Performance Computing
HRC	Human-Robot Collaboration
IPOPT	Interior Point OPTimizer
LQR	Linear Quadratic Regulator
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MPC	Model Predictive Control
MSE	Mean Squared Error
NLP	Natural Language Processing
ODE	Ordinary Differential Equation
PAM	Position-Actuated Manipulator
PD	Proportional-Derivative
PID	Proportional-Integral-Derivative
PINNs	Physics-Informed Neural Networks
PPO	Proximal Policy Optimization
RIA	Robot Institute of America
RK4	Fourth-Order Runge-Kutta

RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
RRT*	Optimal Rapidly-exploring Random Tree
RRT	Rapidly-exploring Random Tree
SRAM	Static Random Access Memory
TCP	Tool Center Point
TPU	Tensor Processing Unit
TRPO	Trust Region Policy Optimization
TUM	Technische Universität München
URDF	Unified Robot Description Format
WGAN	Wasserstein Generative Adversarial Network

Chapter 1

Introduction

This research project is developed within the Chair of Applied Mechanics at the Technical University of Munich (TUM) and is situated in the context of intelligent manufacturing systems, where collaborative robots play a key role in enabling safe and adaptive human-machine interaction.

In the context of Industry 4.0, manufacturing undergoes a profound transformation driven by cyber-physical systems, intelligent automation, and digital communication. This shift enables adaptive, resource-efficient, and globally competitive production systems through the integration of autonomous, sensor-equipped machines capable of responding dynamically to changing conditions [1]. Within these smart and reconfigurable environments, human-machine collaboration becomes central, making collaborative robots (cobots) essential for ensuring safe and efficient interaction between human workers and intelligent production lines [2].

Cobots offer clear advantages by performing repetitive, dangerous, or ergonomically demanding tasks, thereby improving workplace safety, reducing injuries, and increasing productivity. They are particularly suitable for small- and medium-sized enterprises due to their flexibility and cost-effectiveness [3]. However, despite these benefits, conventional cobot controllers still face significant limitations in complex or unstructured environments. Current systems often lack the cognitive capabilities, contextual awareness, and adaptive decision-making required for fully flexible collaboration. They are generally unable to autonomously learn new tasks, generalize across variations, or respond effectively to dynamic and uncertain conditions [3] [4].

Machine learning, especially deep learning and reinforcement learning, emerges as a promising solution to these limitations. By leveraging large volumes of data from sensor-rich environments, machine learning can enable cobots to perceive complex states, predict outcomes, and learn behaviors through interaction, rather than relying on manually programmed responses. This allows for more natural, robust, and flexible human-robot collaboration (HRC), in line with the goals of Industry 4.0 [3] [4].

Against this complications, the central research question of this thesis arises: How can a ma-

chine learning–based controller be designed to enable safe, adaptive, and efficient behaviors in a collaborative robot operating within the intelligent and flexible manufacturing systems?

Figure 1.1 presents a high-level overview of the proposed solution. The system begins with the specification of the environment, including the robot's initial and target positions, the robot's characteristics, and any static obstacles that may affect navigation. Based on this information, a trajectory planner generates a desired trajectory composed of position, velocity, and optionally acceleration profiles for each joint over time. This trajectory is then passed to the controller, which processes the current state of the robot and computes the appropriate torque commands to follow the desired motion. The controller operates in a closed loop, continuously updating the control signals to ensure accurate and safe trajectory tracking.

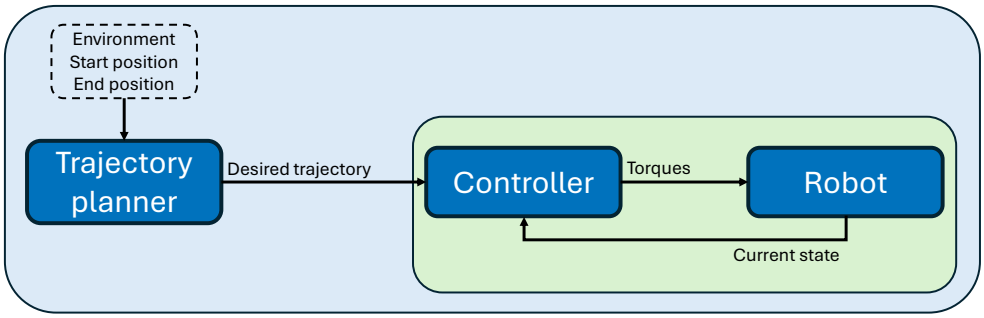


Figure 1.1: High-level diagram of the solution.

1.1 Objectives and Document Structure

The general objective of this work is to design a control signal generation system that optimizes the navigation of a robot with at least two degrees of freedom. To achieve this, four specific objectives are established, each addressing a key stage in the development and validation process.

The first objective is to analyze the requirements for achieving safe and efficient control of robots operating in environments with static obstacles. This involves identifying the main challenges associated with generating optimal control signals, evaluating the limitations of existing control strategies, and reviewing the performance metrics commonly used in current approaches. This analytical phase serves to establish the conceptual foundation of the project, clearly defining the challenges to address and highlighting the critical aspects that require focused attention.

The second objective is to develop a system capable of generating control signals for a robot with at least two degrees of freedom. This stage is the most technically demanding, as it entails designing, implementing, and refining the system from the ground up. It involves researching existing methods and architectures to identify suitable design references and to inform the proposed solution. This objective is considered complete once a functional system is in place that can produce control signals, even in the absence of physical or safety

constraints. The system's performance will be assessed using metrics such as root mean squared error (RMSE) for trajectory tracking, statistics of time duration per control step, and percentage of compliance for the imposed constraints.

The third objective is to extend the developed system by incorporating modifications that allow it to accommodate system constraints and variations in dynamic modeling. Building upon the implementation from the previous stage, this objective focuses on enhancing the system's adaptability and robustness. Its success will be measured by evaluating the system's compliance with predefined physical and safety constraints during operation.

The final objective is to validate the overall effectiveness of the proposed system through simulations in obstacle-rich environments, assessing its ability to track desired trajectories under realistic conditions. The evaluation will be based on the combined use of RMSE, average and maximum inference time per step, and the constraint compliance rate, thereby providing a comprehensive assessment of both performance and safety.

The structure of this work begins with chapter 2, which introduces three foundational topics: robotics, control theory, and artificial intelligence. The section on robotics presents the essential and simplified concepts required to contextualize the development of this work. This is followed by an overview of control theory, which is particularly relevant given its status as the prevailing standard in industry at the time this research was conducted. Lastly, the chapter outlines the fundamental principles of artificial intelligence necessary to understand the reasoning behind the proposed solution and to support the implementation of the corresponding proof of concept.

Chapter 3 presents a detailed explanation of the proposed design methodology, structured in six clearly defined and sequential stages. This chapter also includes a comprehensive description of the proof of concept, addressing both conceptual design and implementation aspects in depth. Chapter 4 contains the key results obtained from the system's implementation, followed by a technical analysis of its performance with respect to the established evaluation criteria. Finally, chapter 5 offers a comprehensive discussion of the outcomes and presents the main conclusions drawn from this work. It also includes recommendations and proposes directions for future research in the topic.

[1]

[1]The complete source code associated with this work is publicly available at `https://github.com/jeauscq/jcq_thesis`.

Chapter 2

Theoretical Framework

2.1 Overview of Robotics

A robot, as officially defined by the Robot Institute of America (RIA), is a reprogrammable multifunctional manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions to perform a variety of tasks [5]. A key aspect of this definition is the concept of reprogrammability, which distinguishes robots from other automated systems. The integration of computational control enables robots to adapt to diverse tasks, making them valuable in industrial environments. Some of the notable advantages of robots include reduced labor costs, enhanced precision and productivity, greater flexibility compared to specialized machines, and improved workplace comfort and safety by automating repetitive or dangerous tasks [5].

2.1.1 Robotic Manipulator

A robotic manipulator is a specific type of robot consisting of a series of rigid bodies, referred to as links, which are interconnected by joints. The primary structural components of a typical manipulator include an arm, which provides reach and mobility; a wrist, which offers additional dexterity; and an end-effector, which performs the designated task. Figure 2.1 [6] illustrates a robotic manipulator with six degrees of freedom. This particular example is slightly more complex than previously described, as it also includes a waist joint and distinguishes between the upper arm and lower arm segments. The structural configuration of a manipulator is based on a kinematic chain, which can be classified as either open (serial) or closed. In an open kinematic chain, a single sequence of links connects the base to the end-effector, allowing independent motion at each joint. In contrast, a closed kinematic chain forms a loop, introducing additional constraints that limit the system's degrees of freedom (DoF) [7].

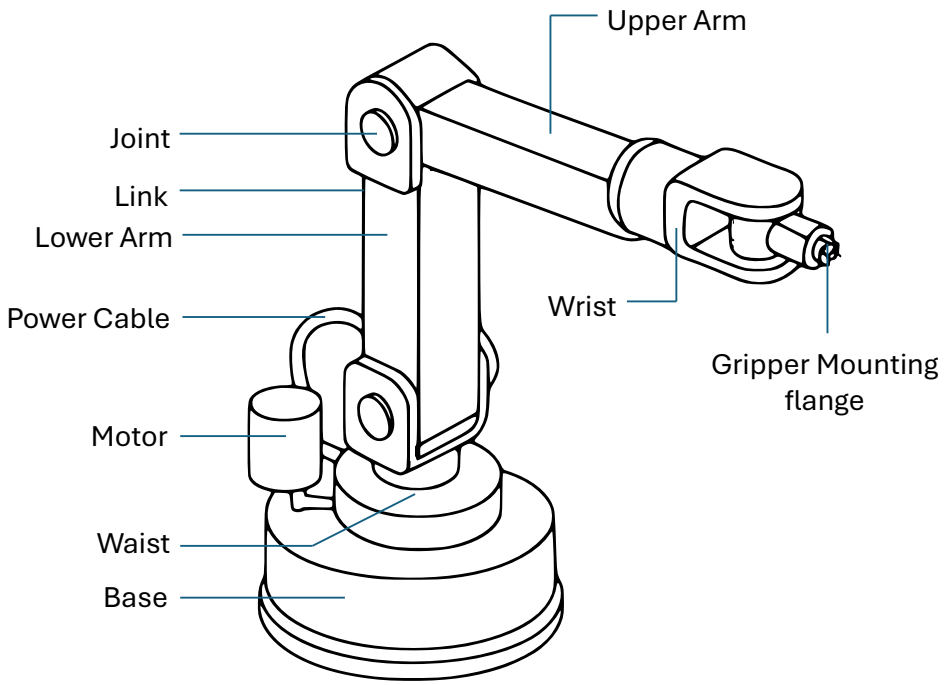


Figure 2.1: Parts of a robotic arm manipulator with 6 Degrees of Freedom (DoF).

Parts of a Robotic Manipulator

As mentioned, manipulators consist of nearly rigid links, which are connected by joints that allow relative motion of neighboring links, they are also shown in Figure 2.1. These joints are typically equipped with position sensors that measure the relative position between adjacent links. For rotary or revolute joints, this displacement is referred to as a joint angle. Some manipulators also feature sliding, or prismatic, joints, where the relative displacement corresponds to a linear translation between links. [8].

The end of the manipulator's link chain is equipped with the end-effector, which varies according to the robot's intended function. It may serve as a gripper, a welding tool, an electromagnet, or another application-specific device. The position of the manipulator is typically defined by the pose of the tool frame, which is rigidly attached to the end-effector and expressed relative to the base frame located at the manipulator's fixed base. [8].

Types of Robotic Joints

Manipulator joints play a crucial role in defining movement capabilities. Two primary types of joints are used: prismatic and revolute. A prismatic joint allows translational motion between two connected links, while a revolute joint enables rotational motion. Revolute joints are typically preferred due to their compact design and high reliability. In an open kinematic chain, each prismatic or revolute joint provides the structure with a single DoF. In a closed kinematic chain, the number of DoFs is reduced due to the constraints imposed

by the loop [7]. Some well-known types of joints are discussed below and shown in Figure 2.2 [9].

Revolute Joint (R): Also known as a hinge, it consists of two congruent surfaces of revolution, one external and one internal. This joint allows only rotational motion about a single axis and has one DoF [10].

Prismatic Joint (P): Also called a sliding joint, it permits only translational motion along a fixed axis. The displacement between two points on the sliding axis determines the relative position of the connected links, and it has one DoF [10].

Cylindrical Joint (C): Formed by two circular cylinders in contact, this joint permits both rotation and sliding along the cylinder axis, providing two DoFs [10].

Spherical Joint (S): Created by two congruent spherical surfaces, this joint allows rotation in up to three independent directions and has three DoFs. It is often replaced by a combination of three revolute joints with intersecting axes [10].

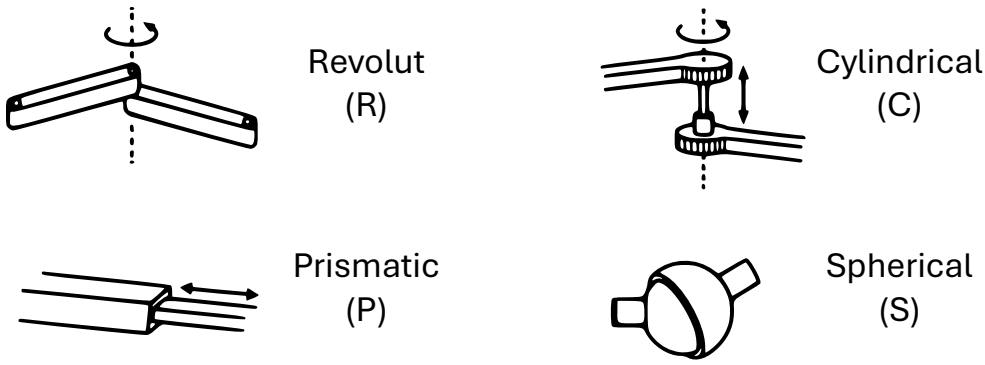


Figure 2.2: Types of joints for a robotic manipulator.

Degrees of Freedom (DoF)

The number of degrees of freedom that a manipulator possesses corresponds to the number of independent position variables required to fully specify the configuration of all its components. This concept applies to any mechanism. In the case of industrial robots, manipulators are typically structured as open kinematic chains, where each joint contributes a single DoF. As a result, the total number of joints in such a manipulator directly determines its number of DoFs [8].

Proper distribution of DoF across the mechanical structure is crucial to ensuring the manipulator has sufficient motion capability to perform a given task. To fully control the position and orientation of an object in three-dimensional space, six DoFs are required, three for positioning a point on the space and three for orienting it relative to a reference coordinate frame. When a manipulator has more DoFs than the minimum required for a given task, it is considered kinematically redundant. Although this redundancy increases control complexity, it can be advantageous by enabling greater dexterity, improved obstacle avoidance,

and alternative motion paths [7]. The kinematic structure of robotic manipulators is often represented as a serial chain, where joints are listed in sequence [7].

2.1.2 Kinematics

Kinematics is the branch of mechanics that studies motion without considering the forces that cause it. In robotics, manipulator kinematics analyzes the position, orientation, velocity, and acceleration of links and actuators as functions of their joint variables. The complete description of a rigid body in space is called the pose, which includes both position and orientation [7] [8]. Figure 2.3 [11] presents a simplified representation of the forward and inverse kinematics of a 2-DoF manipulator. The joint angles are denoted by θ_n, while the end-effector pose is expressed as (x, y). The principles of both forward and inverse kinematics are discussed in the following sections.

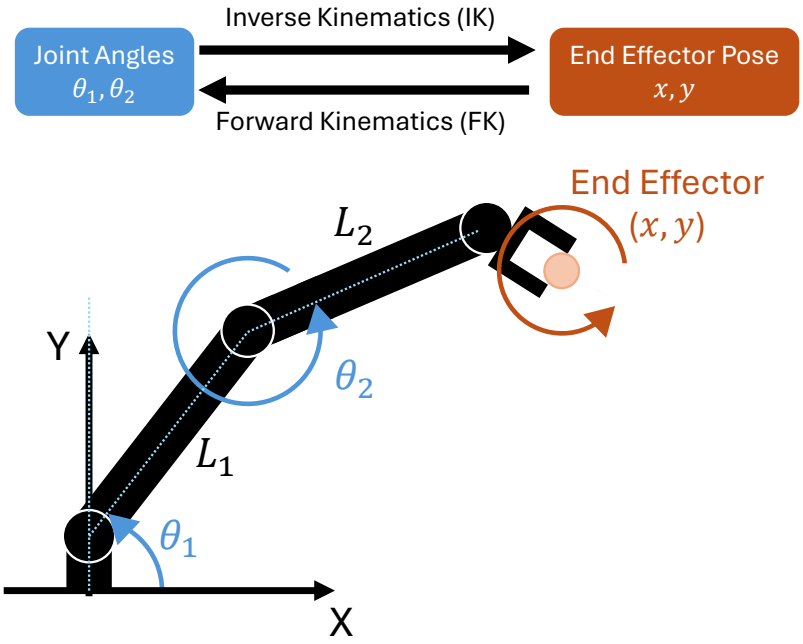


Figure 2.3: Kinematics of a robotic manipulator

.

Forward Kinematics

The forward kinematics problem for a serial-chain manipulator consists of determining the position and orientation of the end-effector relative to the base, given the joint values and the geometric parameters of the robot. To solve this problem, a sequence of homogeneous transformations could be used to relate each link to its predecessor [8].

For manipulators with a tree-like structure, forward kinematics is straightforward. However, in closed-chain mechanisms, additional constraints arise, requiring specialized techniques [8]. Given a serial manipulator with two revolute joints [12], the end-effector position (x, y) is

determined using 2.1 and 2.2. This is a relatively simple case compared to systems with more degrees of freedom and links.

$$x = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) \tag{2.1}$$

$$y = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) \tag{2.2}$$

In which L_1 and L_2 are the lengths of the links and θ_1 and θ_2 are the angular positions of the revolute joints. These equations are exemplified in Figure 2.3.

Inverse Kinematics

The inverse kinematics problem consists of determining the joint values from a desired position and orientation of the end-effector. For a manipulator with n DoF, this requires solving n nonlinear equations derived from the homogeneous transformation matrix. Three equations define the position, and three define the orientation [8]. The equations to solve are typically nonlinear, and thus it is not always possible to find a closed-form solution. The possible scenarios are:

- Multiple solutions.
- Infinite solutions, e.g., in the case of a kinematically redundant manipulator.
- No admissible solutions, in view of the manipulator kinematic structure.

Given a serial manipulator with two revolute joints [13], the angular position of the joints given an end-effector position (x, y) is determined using 2.3 and 2.4. Once again, this is a simplified case in which the end effector is considered static to the end of the second link and the system could be delimited to two dimensions. A graphical intuiton can be obtained from Figure 2.3.

$$\theta_1 = \tan^{-1}\frac{y}{x} - \tan^{-1}\frac{L_2 \sin(\theta_2)}{L_1 + L_2 \cos(\theta_2)} \tag{2.3}$$

$$\theta_2 = \cos^{-1}\frac{x^2 + y^2 + L_1^2 + L^2)}{2L_1 L_2} \tag{2.4}$$

2.1.3 Dynamics

In the context of robotics, dynamics refers to the study of forces and torques that govern the motion of a robotic system. Unlike kinematics, which describes motion in terms of positions, velocities, and accelerations without considering the underlying forces, dynamics explicitly accounts for the physical properties of the system, such as mass, inertia, and external forces. The study of dynamics is fundamental in robotic control, as it enables the accurate prediction and regulation of a robot's movement [10].

The configuration of a manipulator refers to the set of parameters that define its instantaneous geometric arrangement in space. For a robotic arm, the configuration is typically described by the set of joint variables q, which determine the position and orientation of the manipulator's links. However, configuration alone does not provide any information about the system's motion or response to forces [8]. In contrast, the state of the manipulator includes both its configuration and its velocity, forming a set of variables that, when combined with a mathematical model of the system's dynamics and applied inputs, are sufficient to predict its future motion. The state space is the set of all possible states the system can assume [8]. For a robotic manipulator, the dynamics follow Newtonian mechanics, extending the classical equation $F = ma$ to multi-body systems. The state of the manipulator is typically represented by the joint positions q and joint velocities $\dot{q}$, as accelerations are directly related to the time derivatives of velocities. This is expressed as $x = (q, \dot{q})$. This representation allows the manipulator's motion to be described using first-order differential equations, which are fundamental in control design, trajectory planning, and dynamic simulation [8].

Dynamic Model

The dynamic model of a manipulator describes the relationship between the manipulator's motion (joint positions, velocities, and accelerations) and the forces or torques applied to it. This model is essential for understanding the system's response to external and internal forces, as well as for control and simulation purposes [10].

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = \tau \tag{2.5}$$

The components of the dynamic model of a robotic manipulator are explained ahead.

Inertia Matrix $M(q)$: This matrix represents the inertia of the manipulator, describing how mass is distributed across the robot's links. It depends on the robot's configuration q and is symmetric and positive-definite. The inertia matrix is expressed in 2.6. Each element m_{ij} has its own mathematical equation, an example for a 2DoF robot is shown in 2.9 - 2.12.

$$M(q) = \begin{bmatrix} m_{11}(q) & m_{12}(q) & \cdots \\ m_{21}(q) & m_{22}(q) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{2.6}$$

Coriolis and Centrifugal Forces $C(q, \dot{q})$: These terms account for the forces due to the motion of the robot, including Coriolis and centrifugal effects. They are velocity-dependent and are expressed in 2.7. The specific terms depend on the analyzed robot, however, the terms for a 2DoF manipulator are detailed in 2.13 and 2.14.

$$C(q, \dot{q})\dot{q} = \begin{bmatrix} c_1(q, \dot{q}) \\ c_2(q, \dot{q}) \\ \vdots \end{bmatrix} \tag{2.7}$$

Gravitational Forces $G(q)$: Gravitational forces are due to the weight of the robot's links and vary with the robot's configuration q. These forces are represented in vector 2.8. The specific terms for a 2DoF robotic manipulator are presented in 2.15 and 2.16.

$$G(q) = \begin{bmatrix} g_1(q) \\ g_2(q) \\ \vdots \end{bmatrix} \tag{2.8}$$

Applied Torques/Forces τ: These represent the external torques or forces applied at the joints, typically determined by control inputs or external disturbances.

Given a serial manipulator with two revolute joints [14], the mass matrix values from 2.6 are determined using 2.9-2.12, the Coriolis and centrifugal forces from 2.7 are obtained using 2.13 and 2.14, and lastly, the values of the gravitational forces vector from 2.8 are obtained using 2.15 and 2.16.

$$M(1,1) = (m_1 + m_2)L_1^2 + m_2L_2^2 + 2m_2L_1L_2\cos(\theta_2) \tag{2.9}$$
$$M(1,2) = m_2L_2^2 + m_2L_1L_2\cos(\theta_2) \tag{2.10}$$
$$M(2,1) = m_2L_2^2 + m_2L_1L_2\cos(\theta_2) \tag{2.11}$$
$$M(2,2) = m_2L_2^2 \tag{2.12}$$
$$C(1,1) = -2m_2L_1L_2\sin(\theta_2)\,\dot{\theta}_1\dot{\theta}_2 - m_2L_1L_2sin(\theta_2)\,\dot{\theta}_2^2 \tag{2.13}$$
$$C(2,1) = m_2L_1L_2sin(\theta_2)\,\dot{\theta}_1^2 \tag{2.14}$$
$$G(1,1) = (m_1 + m_2)gL_1\sin(\theta_1) + m_2gL_2sin(\theta_1 + \theta_2) \tag{2.15}$$
$$G(2,1) = m_2gL_2sin(\theta_1 + \theta_2) \tag{2.16}$$

In which L_1 and L_2 are the lengths of the links, m_1 and m_2 represent the masses of the links, and θ_1, θ_2, $\dot{\theta}_1$ and $\dot{\theta}_2$ are the angular positions and angular velocities of the revolute joints.

Forward Dynamics

The forward dynamics problem requires the specification of initial conditions, including the manipulator's initial configuration q_0 and initial joint velocities $\dot{q}_0$. These values provide the starting point for the system's motion. Additionally, the applied joint torques τ, which can be generated by the controller or influenced by external forces acting on the manipulator, are given as inputs. These torques play a crucial role in determining the system's dynamics and its motion over time [7] [10].

Forward dynamics is governed by a set of second-order differential equations that describe how the system evolves in response to the applied forces and torques. These equations relate joint accelerations to the applied forces, allowing for the prediction of the manipulator's motion. To solve these equations, numerical methods such as Runge-Kutta or Euler integration are typically employed. These methods calculate the joint accelerations $\ddot{q}$, which are then integrated over discrete time steps to update the joint velocities $\dot{q}(t)$ and joint positions $q(t)$.

This iterative process enables the simulation of the manipulator's motion under the influence of the given inputs and initial conditions [7].

Numerical integration is used to solve ordinary differential equations (ODEs) when an analytical solution is difficult or impossible to obtain. Both the Euler method and the Runge-Kutta method are popular techniques for integrating the equations of motion in robotics, especially in forward dynamics where joint accelerations must be computed over time [15].

The Runge-Kutta method is a more accurate and sophisticated family of numerical methods that provide better approximations than the Euler method by considering additional intermediate points within each time step. The most commonly used version is the 4th-order Runge-Kutta method (RK4), which achieves a high level of accuracy without requiring excessively small time steps [15].

For the differential equation 2.17, the Runge-Kutta of 4th order method computes [15] the new state $y(t + \Delta t)$ as shown in 2.18.

$$\frac{d}{dt}y(t) = f(t, y(t)) \tag{2.17}$$

$$y(t + \Delta t) = y(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.18}$$

The k values computations are computed as shown in 2.19-2.22.

$$k_1 = f(t, y(t)) \tag{2.19}$$

$$k_2 = f\left(t + \frac{\Delta t}{2}, y(t) + \frac{\Delta t}{2}k_1\right) \tag{2.20}$$

$$k_3 = f\left(t + \frac{\Delta t}{2}, y(t) + \frac{\Delta t}{2}k_2\right) \tag{2.21}$$

$$k_4 = f(t + \Delta t, y(t) + \Delta t k_3) \tag{2.22}$$

The Runge-Kutta method improves the accuracy by taking into account the slope of the function at several intermediate points within each time step, rather than just using the slope at the current time point as in the Euler method. As a result, the RK4 method provides a significantly better approximation for many applications, including simulating the forward dynamics of robotic manipulators [15].

Inverse Dynamics

Inverse dynamics refers to the process of determining the joint torques or forces required to produce a desired motion in a robotic manipulator. Unlike forward dynamics, which calculates the system's motion based on applied forces and initial conditions, inverse dynamics works backward from the desired trajectory to compute the necessary control inputs (torques or forces) to achieve that trajectory.

The inverse dynamics problem is formulated using the manipulator's dynamics, previously described in 2.5. Given a desired joint trajectory $q(t)$ along with its velocities $\dot{q}(t)$ and accelerations $\ddot{q}(t)$, the inverse dynamics problem consists of computing the torques $\tau(t)$ needed to follow that motion. It is commonly used in control applications to determine the required joint forces for a planned trajectory. Unlike forward dynamics, which predicts motion from known torques, inverse dynamics compute the torques that produce a specified motion.

In practice, inverse dynamics is computationally intensive, especially for high-degree-of-freedom manipulators, since the manipulator's dynamic model may be complex and nonlinear. Nonetheless, it remains a fundamental tool in robotics, especially when high-precision control of the manipulator's motion is required.

2.1.4 Trajectory planning

Trajectory planning is a crucial step within the broader framework of motion planning, which consists of three main stages: task planning, path planning, and trajectory planning. Task planning defines the high-level goals and sequence of actions required to complete a given operation, such as picking up an object or navigating to a target location. Path planning then determines a feasible route that avoids obstacles and satisfies system constraints, typically by computing a collision-free path in the environment [16].

Finally, trajectory planning refines this path by incorporating time-dependent parameters, such as velocity, acceleration, and dynamic feasibility, ensuring smooth and efficient motion execution.
Trajectory planning is a multi-step process that converts task-space waypoints into joint position commands. To fully understand how this process works, it is essential to define the two main spaces involved. Task space refers to the representation of motion based on the position and orientation of the end effector, while joint space, also known as actuation space, represents movement in terms of the individual positions of the actuators [16]. Figure 2.4 [17] illustrates the relationship between the joint space and the task space, established through forward and inverse kinematics. The joint space is represented by the joint variables q and q', corresponding to the angular positions and velocities of the manipulator's joints, respectively. In contrast, the task space is described in terms of the end-effector's position and velocity in Cartesian coordinates, denoted as P and V, respectively.

Planning spaces

When defining a trajectory, it is possible to take one of two approaches: planning in task space, or planning in joint space. The term "space trajectory" refers to the continuous sequence of points required to transition smoothly from one waypoint to another.

If the trajectory is defined in task space, it ensures higher precision, as interpolation occurs directly within the space where the end effector's movement is being planned. However, this

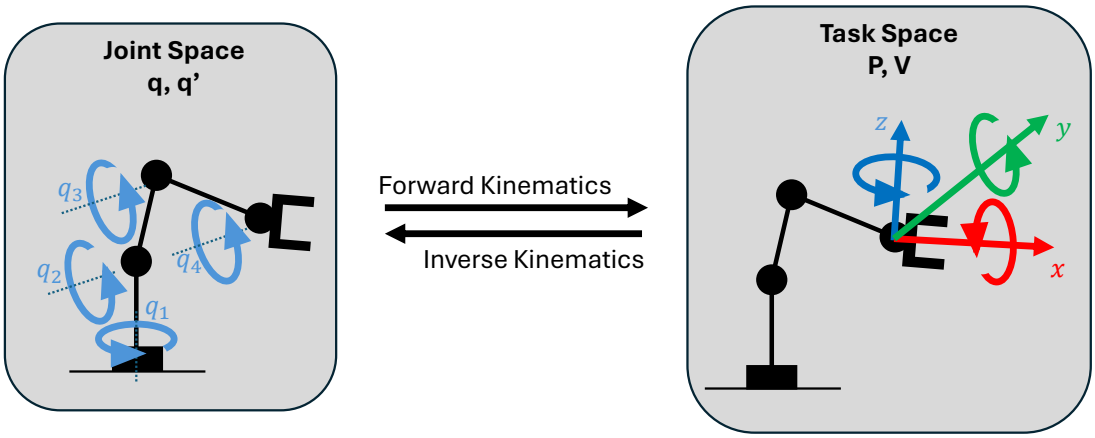


Figure 2.4: Task and joint spaces.

method comes at a higher computational cost because inverse kinematics must be solved at each interpolated point along the trajectory [18].

On the other hand, if the trajectory is planned in joint space, interpolation is performed on the actuator positions rather than in task space. This approach is computationally more efficient, as inverse kinematics calculations are only applied at the waypoints, significantly reducing the number of computations required. However, the trade-off is a loss of precision, since interpolating in joint space does not necessarily result in a smooth or accurate path in task space. As a result, joint space trajectories may deviate from the intended motion, especially when working with complex robotic systems that have highly nonlinear kinematics [18].

By considering the advantages and limitations of each approach, the choice of trajectory planning method depends on the specific requirements of the application, balancing the need for accuracy with computational efficiency.

Trajectory exploration

Sampling methods are a class of algorithms used in motion planning to explore the configuration space of a robotic system by generating a set of discrete points that guide the construction of a feasible trajectory. Unlike traditional deterministic methods that require explicit path optimization, sampling-based techniques provide efficient solutions for high-dimensional spaces by incrementally building paths through random or systematic point selection. These methods are particularly useful when dealing with complex environments with obstacles, where an exhaustive search would be computationally prohibitive. The fundamental principle behind sampling methods is to represent the space in a way that allows for efficient connectivity checking, ensuring that a valid path can be constructed while balancing exploration and computational efficiency [19]. Figure 2.5 [20] presents a generic example in which the method explores four possible trajectories. Three of these paths lead to positions that result in collisions with obstacles represented with dashed boxes and are therefore discarded (indicated in red), while the fourth trajectory, shown in green, is selected as a feasible

and collision-free option.

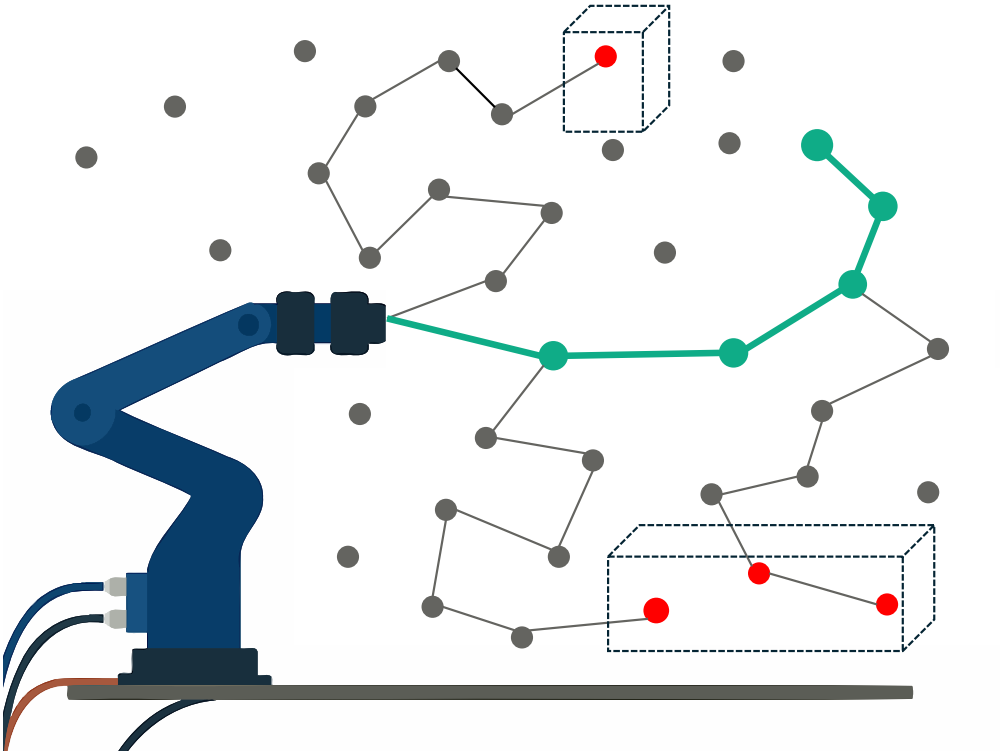


Figure 2.5: Illustration of a sampling-based motion planning method. Sampled trajectories that result in collisions with obstacles are discarded (red points), while a collision-free trajectory is identified as feasible (green points).

The Rapidly-Exploring Random Tree (RRT) algorithm is a widely used sampling-based method designed for path planning in high-dimensional and constrained environments [21]. RRT incrementally builds a tree by randomly sampling points in the configuration space and connecting them to the nearest existing node in the tree, provided that the new connection is collision-free [21]. The key advantage of RRT is its ability to efficiently explore large and complex spaces without requiring an explicit representation of the free space. However, RRT does not inherently guarantee an optimal path; instead, it focuses on rapidly finding a feasible solution [21].

RRT* is an improved version of RRT that introduces an optimization mechanism to ensure asymptotic optimality [22]. While RRT prioritizes rapid exploration, RRT* refines the generated paths by incorporating a rewiring step. Instead of merely connecting a new sample to its nearest neighbor, RRT* evaluates multiple nearby nodes and selects the one that minimizes the total cost from the starting point. Additionally, after adding a new node, the algorithm attempts to reconnect its neighboring nodes to further reduce the overall path cost. This process gradually improves the trajectory as more samples are added, ensuring that the final path converges toward the shortest or most optimal solution given the constraints. Although RRT* requires more computation compared to standard RRT, it is particularly valuable in applications where path quality is crucial [22].

Metrics for Trajectory Comparison

When evaluating the performance of a controller that executes a desired trajectory, it is crucial to establish quantitative metrics that reflect how closely the executed trajectory matches the reference. This section presents a set of widely-used error metrics for such evaluations.

Mean Absolute Error (MAE): It is a straightforward metric that computes the average absolute difference between predicted values and ground truth values over time. For a given signal $y(t)^*$ and its reference $y(t)$, the MAE expression is shown in 2.23 [23].

$$MAE = \frac{1}{T}\sum_{t=1}^{T}|y(t) - y^*(t)| \tag{2.23}$$

MAE is scale-dependent and penalizes all deviations linearly, making it robust to outliers and easy to interpret [23].

Root Mean Square Error (RMSE): This is a quadratic metric that computes the square root of the mean squared differences between predicted and reference values, as shown in 2.24.

$$RMSE = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(y(t) - y^*(t))^2} \tag{2.24}$$

RMSE penalizes larger errors more heavily due to the squaring operation, making it sensitive to peak deviations [23]. This can be useful when significant deviations are particularly undesirable.

Mean Squared Error (MSE): It is similar to RMSE, but without the square root, its formula is shown in 2.25.

$$MSE = \frac{1}{T}\sum_{t=1}^{T}(y(t) - y^*(t))^2 \tag{2.25}$$

While MSE and RMSE are often used interchangeably, MSE is not in the same unit as the original signal and tends to be less interpretable in practical terms [23].

Dynamic Time Warping (DTW): This is a more advanced metric that accounts for temporal misalignment by allowing non-linear mappings between time indices [24]. It is commonly applied to complex time-series or gesture recognition tasks. This metric is represented in 2.26 [24], where π is a warping path aligning the sequences, $\mathcal{A}$ is the set of all valid warping paths that satisfy boundary conditions, continuity, and monotonicity.

$$DTW(\mathbf{y}, \mathbf{y}^{)} = \min_{\pi\in\mathcal{A}}\sum_{(i,j)\in\pi}|y_i - y_j| \tag{2.26}$$

2.2 Control theory

Automatic control is the use of control systems to regulate the behavior of machines or processes with minimal or no human intervention. It involves measuring the system's output, comparing it to a desired set point, and adjusting the system's inputs accordingly. The main components of an automatic control system include sensors (to measure the output), actuators (to influence the system), and controllers (to process feedback and make adjustments). Feedback, whether positive or negative, is crucial for maintaining system stability and ensuring the desired performance. Automatic control can be categorized into open-loop systems, which operate without feedback, and closed-loop systems, which rely on feedback to achieve more precise and stable performance. This field sets the foundation for many modern applications, including industrial automation, electronic systems and robotics.

2.2.1 Optimal Control

Optimal control is a subset of automatic control that focuses on finding the best possible control strategy to achieve a system's desired performance while minimizing a cost function. The key idea is to determine the control inputs that not only steer the system toward the desired outcome but also minimize energy usage, time, or any other performance metric. The optimization process typically involves mathematical modeling, dynamic programming, or calculation of variations to find the control laws that optimize the system's response [25].

LQR (Linear Quadratic Regulator)

LQR is a method in optimal control used to determine the optimal control input for a linear system [26]. The objective of LQR is to minimize a cost function that typically consists of two components: the state error (the difference between the desired and actual state) and the control effort (the magnitude of the control input). The cost function is quadratic in terms of the state and control variables. In mathematical terms, the typically defined cost function is shown in 2.27.

$$J = \int_0^\infty \left(x(t)^T Q x(t) + u(t)^T R u(t) \right) dt \tag{2.27}$$

In which $x(t)$ is the state vector, $u(t)$ is the control vector, and Q and R are positive semi-definite weight matrices that penalize deviations in state and control effort, respectively.

The LQR method assumes that the system dynamics can be described by a linear differential equation such as 2.28, where A and B are system matrices.

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{2.28}$$

The solution to the LQR problem is the calculation of an optimal control law where K is the gain matrix obtained by solving a Riccati differential equation, as shown in 2.29.

$$u(t) = -Kx(t) \tag{2.29}$$

LQR is particularly useful for linear systems with Gaussian disturbances, providing a simple and efficient way to ensure minimal cost in system operation.

Model Predictive Control

Model Predictive Control (MPC) is an advanced control strategy that can handle multivariable, constrained, and nonlinear systems. Unlike LQR, MPC involves solving an optimization problem at each control step, where the control inputs are computed by predicting the future behavior of the system over a finite horizon. MPC is distinct from LQR because it explicitly takes into account system constraints and can handle nonlinearities in the model, making it suitable for a broader range of applications [27].

In MPC, the objective is to minimize a cost function over a finite prediction horizon N [27]. The cost function generally includes terms for the state and control input, and often penalizes deviations from a desired state or trajectory, as well as excessive control effort. The general form of the cost function can be written as shown in 2.30.

$$J = \sum_{k=0}^{N-1} \left((x_k - x_{\text{ref},k})^T Q (x_k - x_{\text{ref},k}) + (u_k)^T R u_k \right) \tag{2.30}$$

In which x_k, u_k and $x_{\text{ref},k}$ are the state, control input, and the reference state trajectory at time step k. While Q and R are weight matrices for state and control effort, and N is the prediction horizon.

At each time step, the MPC controller optimizes the future sequence of control actions over the prediction horizon by solving this optimization problem. However, only the first control input in the sequence is applied to the system. After executing this first control input, the process is repeated at the next time step with updated measurements of the system's state [27].

Model Predictive Control (MPC) is a powerful control strategy known for its ability to handle both hard and soft constraints, making it suitable for systems with strict operational limits on states and control inputs. Unlike LQR, MPC can manage nonlinear system dynamics, which is particularly advantageous in complex applications such as robotics, autonomous vehicles, and process control.

A key limitation of Model Predictive Control (MPC) is its inherent computational complexity, as it requires solving an optimization problem at every control step. This becomes particularly demanding for large-scale systems or when using long prediction horizons [27]. Although MPC is highly effective in handling constraints and predicting collisions within the control horizon, it often depends on nonlinear optimization problems that must be solved iteratively. These problems introduce variability in computation time, making it difficult to guarantee real-time feasibility. Additionally, MPC solvers may converge to local optima, which can reduce performance in dynamic or uncertain environments [28].

Another important consideration is that MPC performance depends heavily on the accuracy

of the underlying model. Classical MPC frameworks typically rely on first-principles models, derived from physics-based formulations. However, obtaining these models can be both tedious and costly, especially when dealing with complex nonlinear systems [29]. These challenges motivate the exploration of learning-based alternatives that aim to reduce modeling effort and improve adaptability.

2.3 Artificial Intelligence (AI)

Artificial Intelligence (AI) is a broad field of computer science and engineering concerned with creating machines or software that exhibit intelligent behavior. A classic definition based on the the ideas of AI pioneer John McCarthy [30] describes AI as science and engineering together, making intelligent machines, specifically intelligent computer programs. An AI system aims to choose actions that maximize its chances of achieving goals based on its perceptions, in other words, it achieves success when it acts rationally in its environment. This requires various capabilities such as perception, reasoning, learning, and decision-making [30].

In the context of robotics, AI provides the foundational principles for producing robots with autonomy and intelligence. An agent is any entity that perceives its environment through sensors and acts upon that environment through actuators [31]. A robot can be seen as an AI agent that senses the physical world (through cameras, lidar, sensors, etc.) and acts on it (through motors and actuators) [31]. Techniques from AI enable robots to plan motions, recognize objects or situations, and make decisions to perform tasks in unstructured or dynamic environments [31].

2.3.1 Machine Learning (ML)

Machine Learning (ML) is a subfield of AI focused on algorithms that improve their performance at a task through experience (data). Rather than being explicitly programmed with fixed rules, an ML system learns a model or behavior from example data. A widely cited formal definition by Tom Mitchell [32] states :

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning methods are typically categorized by the type of learning experience, some of the most common methods are described below [32].

Supervised learning: The algorithm is trained on labeled examples (input-output pairs), learning a mapping from inputs to desired outputs.

Unsupervised learning: The algorithm tries to find patterns or structure in unlabeled data.

Reinforcement learning: The algorithm learns by interacting with an environment, receiving feedback in the form of rewards or penalties.

The common idea is that ML algorithms build a model from data. This model could be a set of rules, a mathematical function, or a complex statistical structure. Many learning algorithms are based in statistics and optimization theory. In robotics, ML techniques allow robots to acquire skills that are difficult to acquire by hard-coding, such as sensor calibration, object recognition, or adaptive control [31] [32].

2.3.2 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs), often just called neural networks (NN), are a family of machine learning models inspired by the biological neural networks in animal brains. A neural network consists of layers of interconnected units (neurons) that compute weighted sums of inputs and apply nonlinear activation functions [33].

Haykin describes a neural network as a highly parallel and distributed computing system composed of simple processing units, or neurons, that is inherently suited to capturing and storing knowledge gained from experience, and retrieving it when needed (inference) [33]. Learning in neural networks involves adjusting weights to minimize a loss function, typically using the backpropagation algorithm and gradient descent [33] [34].

Feed-forward neural networks, also known as multilayer perceptrons (MLPs), represent the fundamental architecture of artificial neural networks, where information flows unidirectionally from the input layer to the output layer without any feedback loops. Figure 2.6 [35] illustrates the structure of such a network, where each circle represents a neuron, and every neuron is fully connected to all neurons in the subsequent layer. Deep learning refers to neural networks with many layers, capable of learning representations from raw data [34] [36]. Neural networks have proven effective for complex tasks such as object detection, image classification, and even control policy learning in robotics [31] [36].

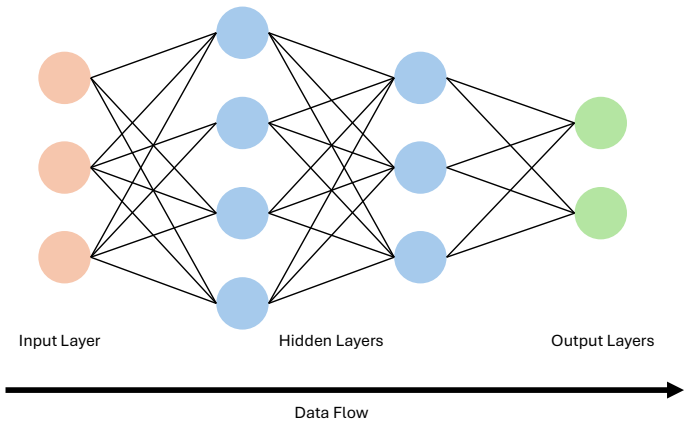


Figure 2.6: Representation of an artificial neural network.

Despite their advantages, neural networks also present challenges such as large data re-

quirements, high computational cost during training, and difficulty in interpretability [36]. However, their capacity to approximate nonlinear functions and generalize from data makes them a crucial part of modern robotics.

Long Short-Term Memory (LSTM) Networks

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that evolves over time. However, standard RNNs suffer from the vanishing gradient problem, making them ineffective at capturing long-term dependencies [37].

To address this, Long Short-Term Memory (LSTM) networks were introduced by Hochreiter and Schmidhuber [37]. LSTMs introduce a memory cell and three gates (input, forget, output) to regulate the flow of information. These mechanisms allow the network to retain relevant information over long sequences [37] [38]. Figure 2.7 presents a high-level representation of the implementation of LSTM networks.

Figure 2.7 [39] illustrates the internal flow of data in an LSTM cell. The leftmost part of the diagram shows the input x_t and the previous hidden state h_{t-1} , which are used to compute the activations of the three gates. The forget gate decides which parts of the previous cell state C_{t-1} to discard. The input gate regulates the incorporation of new candidate values (obtained through a tanh activation) into the cell state. These components combine to produce the updated cell state C_t. The output gate then determines which parts of the new cell state are exposed to the next layer or time step, producing the new hidden state h_t. Each gate's output is computed using a sigmoid activation (σ), resulting in values between 0 and 1, which act as dynamic filters for information flow.

Each gate is controlled by a sigmoid activation function, determining how much information to pass through. The internal cell state is updated through element-wise operations involving the gates, and the output is derived by filtering the updated cell state [38].

LSTMs have become a standard choice for tasks involving temporal dependencies, such as speech recognition, time series forecasting, and sequence prediction. In robotics, LSTMs have been applied to predict trajectories, detect anomalies in sensor data, and enhance control systems by incorporating memory of past states [38] [40].

Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) are a simplified variant of LSTMs, introduced by Cho et al. [41]. GRUs combine the input and forget gates into a single update gate and include a reset gate to control the influence of past states. They do not maintain a separate cell state, simplifying the architecture. GRUs retain many of the benefits of LSTMs while requiring fewer parameters, making them more computationally efficient [41] . Like LSTMs, they are capable of learning long-term dependencies and are used in a wide range of sequence modeling tasks [42].

In robotics applications, GRUs offer advantages for onboard computation where hardware

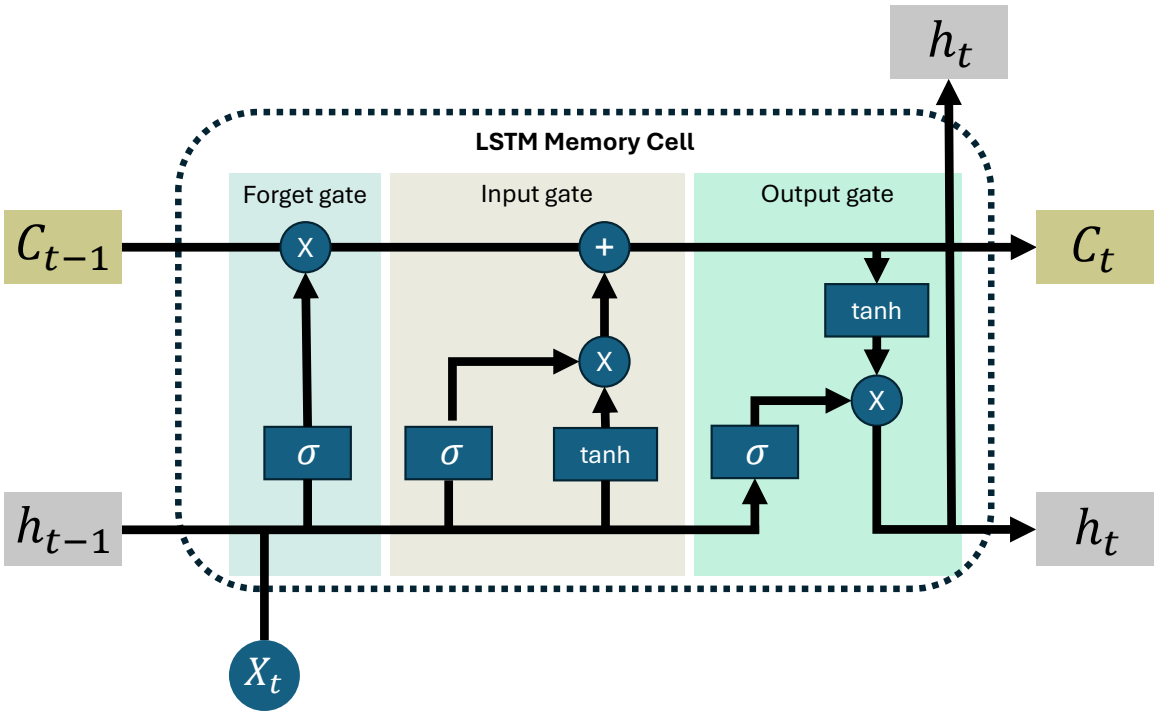


Figure 2.7: Representation of long-short term memory network.

resources are limited. GRUs are well-suited for processing time-series data due to their ability to capture temporal dependencies with a simplified architecture. This reduced complexity enables faster training and deployment compared to more complex recurrent models, while still maintaining competitive performance [42, 43].

Figure 2.8 [44] shows the data flow within a GRU cell. The inputs x_t and previous hidden state h_{t-1} are used to compute two gates: the reset gate R_t and the update gate z_t, both regulated by sigmoid activations. The reset gate controls how much of the past information is considered when computing the candidate activation h_t, which is generated via a tanh function. The update gate then interpolates between the previous hidden state and the candidate activation to produce the new hidden state h_t. This structure allows the GRU to flexibly decide what information to retain or overwrite at each time step, without the need for a separate memory cell.

2.3.3 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a class of generative models introduced by Goodfellow et al. [45], designed to learn how to generate new data samples that resemble a given data set. GANs have become highly popular in the field of machine learning due to their ability to model complex data distributions without the need for explicitly defining them.

The fundamental idea behind GANs is to train two neural networks simultaneously in a competitive framework: the Generator and the Discriminator . These networks are structured in

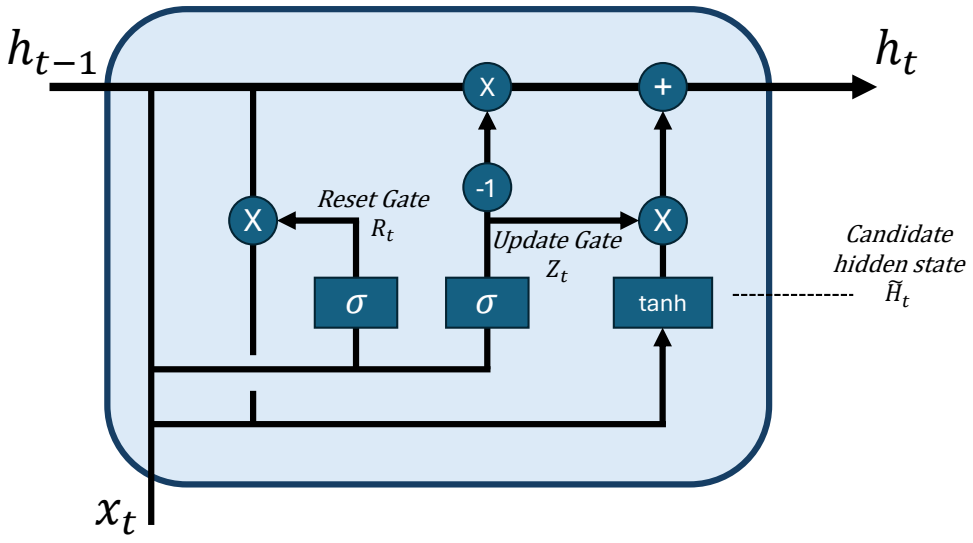


Figure 2.8: Representation of a gated recurrent unit.

an adversarial setting, where each one has an opposing objective relative to the other. The networks play a min-max game in which the generator tries to fool the discriminator into thinking that the synthetic samples are real, meanwhile the discriminator tries to keep up by learning to differentiate between the reference data and generated samples [45]. Figure 2.9 [46] shows the typical architecture of a GAN, widely used in image generation tasks. In this setup, a generator creates synthetic images from random noise, while a discriminator evaluates whether the input images are real (extracted from the dataset) or fake (produced by the generator). The feedback from the discriminator is used to optimize both networks in an adversarial training loop. This image-based implementation exemplifies one of the most common and well-known use cases of GANs.

Architecture of a GAN

As mentioned earlier, GANs consist of two neural networks, the generator and the discriminator, which can be constructed using various types of layers, activation functions, and regularization techniques. The specific architecture depends on the characteristics of the problem being addressed. A more detailed description of each component is provided in the following paragraphs, based on the original formulation in [45].

Generator (G): The generator is a neural network whose objective is to generate synthetic samples that resemble those from the real data set. It does so by transforming a random noise vector z, usually sampled from a simple distribution such as a uniform or Gaussian distribution, into a data sample $G(z)$ in the same space as the real data. This action can be formally represented as $G : z \rightarrow x_{fake}$, where x_{fake} represents the synthetic sample generated by the network.

Discriminator (D): The discriminator is a neural network that acts as a binary classifier. Its goal is to distinguish between real samples x_{real}, taken directly from the data set, and

fake samples x_{fake}, created by the generator. The output of the discriminator $D(x)$ is a scalar value between 0 and 1, representing the probability that a given input sample is real. This can be formally represented as $D : x \rightarrow [0, 1]$, where x could be either x_{fake} or x_{real}.

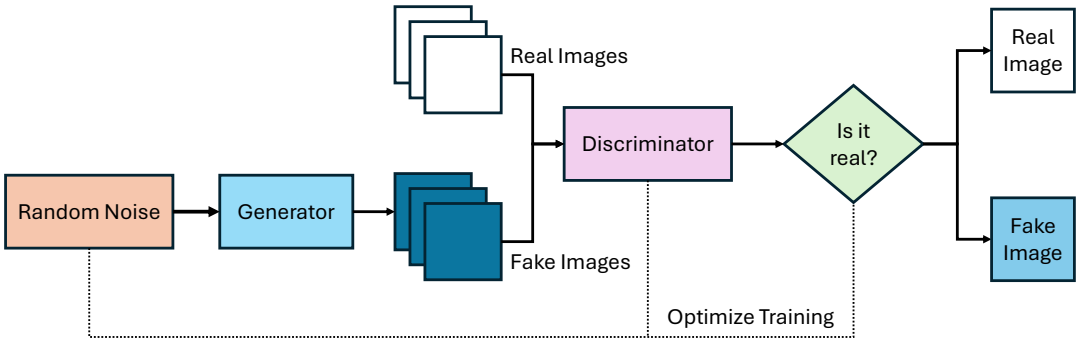


Figure 2.9: GAN high-level block diagram.

Adversarial Training Setup

As mentioned before, the training of a GAN can be described as a minimax game [45], where the discriminator aims to correctly classify real samples as real ($D(x_{real}) \approx 1$) and fake samples as fake ($D(G(z)) \approx 0$). Meanwhile the generator tries to fool the discriminator into believing that the generated samples are real ($D(G(z)) \approx 1$).

This interaction is formalized through the minimax objective function shown in 2.31, where $p_{data}(x)$ denotes the real data distribution and $p_z(z)$ denotes the prior distribution of the noise vector.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[log(D(x))] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{2.31}$$

Optimization Procedure

Training of a GAN typically alternates between two steps: discriminator update and generator update.

For the discriminator update, a batch of real data x_{real} from the data set, and a batch of noise vectors z turned into generated fake data ($G(z) = x_{fake}$) are sampled. Then the discriminator parameters are updated to maximize the probability of correctly classifying real and fake samples. This step corresponds to maximizing 2.32.

$$log(D(x_{real})) + log(1 - D(G(z))) \tag{2.32}$$

On the other hand, for the generator update a batch of noise vectors z is firstly sampled and then fake data ($G(z) = x_{fake}$) is generated. Then the generator parameters are updated

to minimize the ability of the discriminator to distinguish the fake samples. This steps corresponds to maximizing 2.33.

$$logD(G(z)) \tag{2.33}$$

2.3.4 Reinforcement Learning (RL)

Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment. The idea is that the agent takes actions, observes the results of those actions, and receives feedback in the form of rewards or penalties [47]. Over time, by trying different actions and seeing which ones lead to better outcomes, the agent gradually learns a strategy, called a policy, that allows it to achieve its goals more effectively [47]. Reinforcement learning is inspired by the way humans and animals learn from experience, using trial and error to improve their behavior in order to maximize long-term success [47].

Figure 2.10 [48] illustrates the standard RL loop. The agent interacts with the environment by observing its current state and selecting actions based on a policy. The environment responds by transitioning to a new state and returning a reward. The reinforcement learning algorithm updates the policy based on the observed outcomes, aiming to maximize cumulative reward over time. This closed-loop interaction forms the basis of most RL-based control systems.

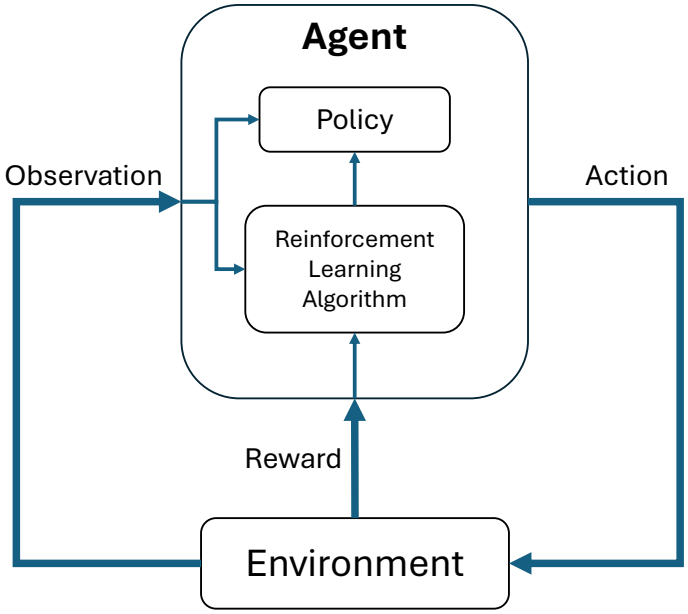


Figure 2.10: Generic reinforcement learning setup

.

Key Concepts in RL

In reinforcement learning, there are several key concepts [47] that define how the learning process works and they will be discussed in the following descriptions.

Agent: The agent is the learner or decision-maker. It interacts with its surroundings and makes decisions about which actions to take.
Environment: The environment represents everything that surrounds the agent. It defines the conditions or rules of the world where the agent operates. The agent interacts with the environment by taking actions and receiving observations and rewards in return.
State: The state is a representation of the current situation or condition of the environment. It provides the agent with information about where it is or what is happening at a given moment.
Action: An action is a decision or move that the agent makes to interact with the environment. Depending on the action chosen, the environment changes or evolves.
Reward: The reward is a numerical value that the environment gives the agent after taking an action. This reward tells the agent how good or bad that action was in that specific situation. The goal of the agent is usually to maximize the total rewards it receives over time.
Policy: The policy is the strategy that the agent uses to decide which action to take in each state. It can be thought of as a function or rule that maps states to actions.
Episode: An episode is a complete sequence of interactions between the agent and the environment, starting from an initial state and ending when certain conditions are met, such as reaching a goal or a failure.
Return: The return is the total accumulated reward the agent receives in an episode or over time. The agent's goal is to learn a policy that maximizes this return.

Reinforcement Learning Methods

In reinforcement learning, there are several families of algorithms that differ mainly in how the agent learns to make decisions. One of the most common families is value-based methods, where the agent learns a function that estimates how good it is to be in a certain state or to take a certain action. A well-known example of this approach is the Q-Learning algorithm [49]. Another family is policy-based methods, where instead of learning values, the agent directly learns a policy that tells it what action to take in each situation. Finally, there are actor-critic methods, which combine both ideas by having two separate models: one that estimates values (critic) and one that selects actions (actor) [49].

Among these families, policy gradient methods, which belong to the policy-based and actor-critic approaches, are of particular interest in this work, since they are especially suitable for problems with continuous action spaces, such as the control of robots. In this type of method, the agent improves its policy using gradient ascent, updating its parameters in the direction that increases the expected reward. However, a common challenge in policy gradient methods is that taking very large updates in the policy can lead to unstable behavior

or even performance collapse [49].

To address this problem, algorithms like Trust Region Policy Optimization (TRPO) were proposed. TRPO introduced the idea of limiting how much the policy can change at each update, ensuring that learning is more stable [50]. However, TRPO required complex mathematical tools and second-order optimization methods, which made its implementation and computational cost less practical [50].

Proximal Policy Optimization (PPO) emerged as a simpler and more efficient alternative to TRPO. Instead of enforcing a hard constraint on policy changes, PPO modifies the loss function by introducing a clipping mechanism that penalizes updates that are too large [51]. This prevents the new policy from deviating excessively from the previous one, preserving stability while maintaining simplicity. Moreover, PPO is typically implemented within the actor-critic framework, where the actor represents the policy that selects actions, and the critic estimates the value function that guides the learning process [51]. Thanks to this balance between performance, stability, and ease of implementation, PPO has become one of the most widely used reinforcement learning algorithms, especially in continuous control tasks such as robotics.

2.3.5 Proximal Policy Optimization (PPO) Algorithm

Proximal Policy Optimization (PPO) is categorized as an online reinforcement learning algorithm. This means that the agent improves its policy through direct interaction with the environment, collecting fresh data at each iteration rather than learning from a fixed dataset. As the agent executes actions, it receives corresponding observations and rewards, which are immediately used to update the policy. This continual feedback loop allows the policy to evolve based on up-to-date experiences.

Figure 2.11 [48] provides a visual representation of the online setting, in which the agent communicates in real-time with the environment: sending actions and receiving observations and rewards in return. Conversely, Figure 2.12 [48] represnets the offline reinforcement learning, which uses previously recorded (logged) interactions, without further environmental queries. This distinction is key in many real-world robotic applications, where online learning offers greater adaptability and responsiveness.

In this section, a more detailed and organized explanation of the PPO algorithm is presented, based on the original implementation [51].

Data Collection

At the beginning of each iteration, the agent interacts with the environment using the current policy $\pi_\theta(a|s)$, where θ denotes the parameters of the policy network. Through this interaction, the agent collects a set of trajectories or rollouts, consisting of sequences of states (s_t), actions (a_t), rewards (r_t), value estimates $V(s_t)$ given by the critic, log-probabilities of the executed actions $log\pi_\theta(a_t|s_t)$. This data is collected until a pre-defined number of steps

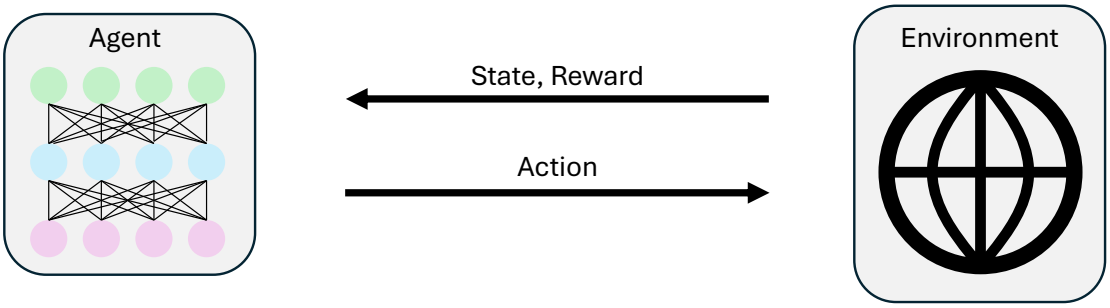


Figure 2.11: Representation of online RL.

Figure 2.12: Representation of offline RL.

is reached.

Advantage Estimation

Once the trajectories are collected, it is necessary to estimate the advantage function A_t, which measures how much better or worse an action performed compared to the expected value of the state. There are several methods commonly used to compute this estimate, depending on the desired trade-off between bias and variance.

A straightforward approach is the Monte Carlo estimation, which uses the empirical return R_t, defined as the sum of the discounted future rewards, as shown in 2.34.

$$A_t = R_t - V(s_t) \tag{2.34}$$

where:

$$R_t = \sum_{l=0}^{T-t} \gamma^l r_{t+l} \tag{2.35}$$

Here, T represents the time step at which the trajectory terminates, r_{t+l} are the rewards obtained from time step t onward, and $V(s_t)$ is the estimated value of state s_t. Although this method provides an unbiased estimate of the advantage, it often suffers from high variance,

which can negatively affect the learning process.

To mitigate this issue, PPO commonly employs the Generalized Advantage Estimation (GAE) method, which introduces a parameter λ [0, 1] to control the trade-off between bias and variance. GAE defines the advantage estimate as a weighted sum of temporal difference (TD) errors δ_t as shown in 2.36.

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots \tag{2.36}$$

Where the TD error at time t is defined as shown in 2.37.

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.37}$$

By adjusting the parameter λ, GAE provides a flexible mechanism to balance the variance reduction offered by bootstrapping through $V(s_{t+1})$ with the unbiased nature of the Monte Carlo estimation. In particular, setting $\lambda = 1$ recovers the Monte Carlo advantage estimate, while setting $\lambda = 0$ corresponds to using the single-step temporal difference error.

Policy Update with Clipped Objective

The core idea of PPO is to limit how much the new policy π_θ is allowed to differ from the old policy $[\pi_\theta]_{old}$ that generated the data, without requiring the complexity of TRPO. This is achieved through the surrogate objective function defined in 2.38.

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{2.38}$$

In 2.38, ϵ is a hyperparameter that determines how far the new policy is allowed to move away from the old one, the ratio $r_t(\theta)$ can be described as the probability ratio between the new and old policies, the clip function ensures that if $r_t(\theta)$ moves outside the interval $[1-\epsilon, 1+\epsilon]$, the objective does not increase further, thus preventing excessively large policy updates and . The ratio is obtained as shown in 2.39.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{[\pi_\theta]_{old}(a_t|s_t)} \tag{2.39}$$

Value Function Update

Simultaneously, the critic is updated by minimizing the squared error between the predicted value and the empirical return R_t, this expression is shown in 2.40.

$$L^{VF}(\theta) = E_t[(V_\theta(s_t) - R_t)^2] \tag{2.40}$$

The empirical return R_t is typically computed as the discounted sum of future rewards.

Entropy Bonus (Optional)

To encourage exploration and avoid premature convergence to deterministic policies, an additional entropy term, shown in 2.41, can be included in the final loss function. Where H denotes the entropy of the policy distribution.

$$L^{ENT}(\theta) = E_t[H[\pi_\theta](s_t)] \tag{2.41}$$

Final Loss Function

The final objective that is optimized combines the policy loss, the value loss, and the entropy bonus is described in 2.42, where c_1 and c_2 are coefficients that weight the importance of the value loss and entropy bonus, respectively.

$$L(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 L^{ENT}(\theta) \tag{2.42}$$

2.4 Learning-Based Controllers

MPC is a mature and widely adopted control technique in industrial applications. Nonetheless, its limitations in terms of model dependence and computational cost have led to recent research that seeks to enhance its capabilities through the integration of data-driven methods.

Liu and Atkeson [52] proposed an approach that improves control performance using a trajectory library constructed from simulations. Their method employs a linear-quadratic regulator (LQR) to generate candidate trajectories and applies a k-nearest neighbor algorithm to select the most appropriate one based on historical execution data and task-specific objectives. Vallon and Borrelli [53] extended this idea into an iterative learning MPC framework. Their task decomposition strategy enables the controller to refine its behavior over time by leveraging accumulated data, enhancing robustness and adaptability.

Neural networks have become essential tools for controlling uncertain and nonlinear dynamical systems [54, 55]. Multiple architectures have been explored, including radial basis function (RBF) networks, multilayer feedforward networks, and RNNs, all applied to trajectory tracking problems [56, 57, 58, 59]. Additionally, the integration of fuzzy logic into neural control has emerged as an active research area, expanding the flexibility of intelligent controllers.

A neural-network-based sliding mode adaptive controller demonstrated promising performance for trajectory tracking under system uncertainties, although it was validated only in simulation [60]. Another adaptive controller designed specifically for robot manipulators utilized a two-layer feedforward neural network [61, 62], reinforcing the potential of neural networks to handle uncertainty and nonlinearities in real-time control applications.

Efforts have also been made to reduce the dependency on explicit physical models within MPC frameworks. One of the most notable examples is the work of Nicodemus et al. [63], which integrates Physics-Informed Neural Networks (PINNs) into an MPC structure. PINNs are trained to approximate solutions of differential equations while incorporating physical laws directly into the learning process. Traditionally used for forward modeling, PINNs in this context are augmented with time-varying inputs and initial conditions, enabling their use as a differentiable dynamics model within the optimization loop. This allows for accurate, data-efficient modeling, especially in systems with high dimensionality or partially known dynamics. The contribution of this work demonstrates a valuable convergence of learning-based modeling with optimization-based control.

Hybrid controllers that combine data-driven feedforward structures with classical feedback mechanisms have also gained traction. In [64], an adaptive controller was proposed for a 3-DoF manipulator using an RBF network to approximate inverse dynamics online. A secondary neural network handled unmodeled nonlinearities, and the combined architecture was embedded within a feedback loop. Stability was formally established through Lyapunov analysis. The approach was later extended to scenarios with partial observability using high-gain observers, achieving reliable performance under various uncertainties.

A comparable hybrid structure is described in [65], where inverse dynamics models are learned offline using several neural architectures. These learned models are then used for real-time feedforward control, supplemented by classical feedback to correct residual errors. The system adapts effectively to nonlinearities and time-varying behaviors, providing a flexible alternative for scenarios where exact system modeling is impractical. These studies affirm that combining learned approximations with feedback correction offers a scalable and stable path for adaptive control.

Although the use of GANs in control remains relatively limited, existing studies highlight significant promise. One of the most influential works in this area is Generative Adversarial Imitation Learning (GAIL) [66]. GAIL allows an agent to learn expert behavior without requiring action labels or explicit rewards. Instead, it uses adversarial training to distinguish between expert demonstrations and policy-generated actions, enabling the agent to imitate behaviors through optimization alone. This framework is particularly well-suited for tasks where reward shaping is challenging.

In the low-level control domain, Zhou et al. [67] proposed a GAN-enhanced PID controller for pneumatic artificial muscle (PAM) driven antagonistic joints. The GAN adjusts PID parameters in real time based on a reference model, helping the controller adapt to the nonlinear and time-varying dynamics of PAM actuators. While effective for immediate regulation, this approach is not designed for trajectory tracking.

A more relevant contribution in terms of trajectory tracking and control is the work by Xu and Karamouzas [68], which develops a physics-based character controller using a GAN-like imitation learning framework. Although the application is in character animation, the methodology is highly applicable to robotic manipulators due to shared reliance on dynamic policies and joint-level control. In their setup, control policies output target joint postures,

which are then translated into torques by proportional-derivative (PD) controllers. The policy is trained with a GAN architecture, where a GRU-based recurrent model captures motion history to ensure smooth transitions and temporal coherence.

The system uses multiple discriminators to stabilize adversarial training and avoid reward collapse, employing hinge loss and a gradient penalty mechanism inspired by WGAN-GP. These design choices improve generalization and ensure smoother learning dynamics, resulting in a robust and scalable framework for trajectory imitation.

Another innovative application of GANs in dynamics is presented by Jegorova et al. [69], where the goal is to improve system identification rather than direct control. The authors use a GAN to generate cyclic trajectories that optimize excitation for parameter identification. A discriminator evaluates both the trajectory validity and the diversity of the resulting dynamic responses. The proposed loss function encourages generation of trajectories that maximize information gain while remaining physically valid and collision-free. Although not focused on trajectory tracking per se, the framework illustrates how GANs can generate structured motion sequences tailored for specific control objectives.

Finally, the Dyn-cGAN architecture described by Rostamijavanani et al. [70] presents a generative model for predicting system behavior under varying initial conditions and parameters. The model is capable of producing both periodic and chaotic responses and can be used to approximate nonlinear systems without explicit differential equations. The authors suggest that this approach could be extended to incorporate safety and operational constraints by embedding limits directly into the generative process. This would enable constraint-aware prediction of control signals, making the model suitable for safety-critical applications where constraints must be respected while transitioning between dynamic states.

Chapter 3

Design of a Control System for Execution of Trajectories in Robots

3.1 Design Methodology

Engineering design requires a systematic approach in which each decision made throughout the process is justified. To achieve this, it is essential to define a design methodology that ensures proposed solutions are aligned with client needs and project objectives. A well-defined methodology establishes objective criteria, warrants decision traceability, and provides a structured framework that facilitates the execution of the design stages. The stages of the design methodology used in this project are defined below, adapted from the Ulrich-Eppinger methodology in [71].

1. Initial Evaluation: The process begins with an initial evaluation consisting of a conversation with the client. At this stage, the goal is to identify the current situation and determine whether the objective is to solve an existing problem or to develop a new idea. What needs to be changed and the reason for doing so are clearly defined in this stage.

2. Needs Identification: After the initial evaluation, the next step is to determine the client's needs. These needs form a list of requirements that the solution must meet in order to be considered satisfactory. It is essential that this list is detailed and accurately reflects the client's expectations.

3. Specification Definition: Based on the identified needs, quantifiable specifications are established to objectively assess whether the solution meets the defined requirements. These specifications serve as evaluation criteria to ensure the final design fulfills what was requested.

4. Exploration of Options: It is essential to analyze different alternatives to ensure all relevant possibilities are considered. In this stage, the most viable options are examined

and compared based on the previously established criteria. The final selection is made based on consistency with the specifications and feasibility of implementation.

5. Implementation and Validation: Finally, the selected solution is implemented through a proof of concept and tested to demonstrate that it meets the defined requirements. This stage verifies that the design is functional and satisfies the needs established in the initial phase of the process.

3.1.1 Stage 1: Initial Evaluation

Client Description

The client in this project is M.Sc. Tanmay Goyal. Mr. Goyal is a researcher and Ph.D. student affiliated with the Chair of Applied Mechanics at TUM, and also works as an R&D robotics engineer at ABB. He has a background in robotics, systems and control, as well as industrial and production engineering, with a specialization in robotics. His work focuses on robotic manipulators (particularly the ABB GoFa model), safe artificial intelligence and machine learning, physics-informed neural networks (PINNs), automated machine learning (AutoML), the Robot Operating System (ROS), and data analysis. His combined experience in industry and academia enables him to address challenges in mechatronics and automation from a broadly informed perspective [72].

The project will be conducted at the Applied Mechanics Laboratory at TUM, a research center within the Department of Mechanical Engineering. This laboratory is known for its work in the dynamics of mechanical and robotic systems, emphasizing the development of innovative simulation and experimental techniques for the analysis of complex structures. It is also dedicated to the design, construction, and control of advanced robotic systems [73].

Raw Information Gathering

Through conversations with the client, a clearer and more detailed understanding of the thesis context is achieved. The project is developed under the supervision of a doctoral student at TUM, who focuses on enhancing the performance of the ABB GoFa 150000 robotic manipulator, shown in Figure 3.1, using AI and ML techniques. As part of his doctoral work, the client mentors undergraduate and master's students on thesis topics that demonstrate solid theoretical foundations and potential for improving robotic systems through AI. This specific project aims to improve the system responsible for controlling the robot during real-time trajectory execution.

During these discussions, the client specifically identifies the following key limitations of existing control approaches:

- Modeling Complexity: There is a critical need for accurate theoretical models that can reliably represent the robot's behavior. However, due to the inherently nonlinear and complex dynamics of robotic systems, developing such models is highly challenging. This is a fundamental limitation of traditional model-based paradigms.

Figure 3.1: ABB GoFa 15000.

- Computational Load: Model-based methods often require solving differential equations to compute control signals from reference states. This process can be computationally intensive, which poses a barrier to real-time implementation, particularly in systems requiring fast, responsive control.
- Constraint Handling: While data-driven approaches generally avoid the need for explicit models and offer faster inference, they typically struggle to enforce physical and safety constraints. This shortcoming becomes especially problematic in collaborative robotics, where ensuring the safety of human-robot interaction is crucial.

3.1.2 Stage 2: Needs Identification

On this stage a list of requirements identified through discussions with the client is established. These define the core capabilities that the designed system (DS) must fulfill. They serve as a reference throughout the coming design stages, guiding each decision to ensure alignment with the client's expectations and project objectives.

- The DS represents an innovation relative to the current state of the art.
- The DS is based on an artificial intelligence paradigm.
- The DS successfully enables the manipulator to follow a desired trajectory.
- The DS operates in real time, within latency limits acceptable for industrial applications.

- The DS generates control signals that respect physical and safety constraints.
- The DS is compatible with manipulators having at least 2 DoF.
- The DS is scalable to manipulators with a higher number of DoF.
- The DS is robust to different environments.
- The DS supports common industrial position and velocity sensors.
- The DS allows the transfer of control policies from simulation to real hardware.

Feasibility considerations may also influence design decisions throughout the execution of the design methodology, particularly those that involve significant time or economic resources. It is important to recognize that the proof of concept must be completed within the stipulated time-frame for this work and must serve as a representative implementation of what the system would look like in a real industrial context.

3.1.3 Stage 3: Specification Definition

At this stage,, specifications are defined to quantify the needs set in the previous phase. These specifications enable an objective verification of whether the needs are met. Well-established specifications are measurable, objective, and precise. They allow for a deterministic assessment of whether a need has been fulfilled. The assignment and definition of these specifications are presented in Table 3.1.

#	Need	Specification
1	The DS represents an innovation relative to the state of the art in manipulator controllers.	1.1 The paradigm was never implemented in manipulators in the literature.
2	The DS is based on an artificial intelligence paradigm.	2.1 The DS's core algorithm is based on artificial intelligence.
3	The DS successfully enables the manipulator to follow the desired trajectory.	3.1 RMSE between the desired and executed positions. 3.2 RMSE between desired and executed velocities.
4	The DS operates in real time within latency limits acceptable for industrial applications.	4.1 Average computation time per control step. 4.2 Average standard deviation of time per control step.
5	The DS generates control commands that respect physical and safety limitations.	5.1 Percentage of control commands within physical and safety limits.
6	The DS works with manipulators with at least 2 DoF.	6.1 System capability to execute tasks with 2-DoF manipulators.
7	The DS is scalable to manipulators with more DoF.	7.1 Estimated adaptation time to manipulators with more DoF.
8	The DS is robust to different environments.	8.1 Standard deviation of the tracking error in different scenarios.
9	The DS is compatible with different perception systems and sensors commonly used in industry.	9.1 System capability to operate with encoders and state sensors.
10	The DS allows control policy transfer from simulation to real hardware.	10.1 Change in MAE or RMSE between simulation and real hardware.

Table 3.1: Designed system's needs and their associated metrics

Once the specifications are defined, the next step is to establish the marginal and ideal values. The marginal value represents the minimum acceptable performance the client would consider satisfactory, while the ideal value reflects the expected performance in the absence of complications during the development and implementation of the solution. Table 3.2 presents the corresponding units, as well as the marginal and ideal values for each specification.

ID	Specification	Units	Marginal	Ideal
1.1	The paradigm has been previously implemented in manipulators in the literature.	Binary	Yes	Yes
2.1	The DS's core algorithm is based on artificial intelligence.	Binary	Yes	Yes
3.1	Average RMSE between the desired and executed position.	rad	<0.2	<0.15
3.2	Average RMSE between the desired and executed velocity.	rad/s	<0.5	<0.4
4.1	Average computation time per control step.	ms	< 4	<3
4.2	Average standard deviation of time per control step.	ms	<2	<1
5.1	Percentage of control commands within physical and safety limits.	%	>90	100
6.1	System capability to execute tasks with 2-DoF manipulators.	Binary	Yes	Yes
7.1	Estimated adaptation time to manipulators with more DoF.	Work weeks	< 4	< 2
8.1	Standard deviation of the positions' RMSE in different scenarios.	Radians	<0.18	<0.15
9.1	System capability to operate with encoders and state sensors.	Binary	Yes	Yes
10.1	Change in RMSE between simulation and real hardware.	%	<5	<2

Table 3.2: Specifications of the design system (DS) with the corresponding units, marginal value and ideal value.

Since this is a research-oriented project focused on exploring an innovative solution beyond the current state of the art, this stage requires a slight adaptation from the traditional approach described in the reference methodology by Ulrich and Eppinger [71]. In this context, marginal and ideal target values are not defined at the outline of the project. Instead, they are progressively established based on the performance of baseline implementations and insights gained during early development. The client has explicitly stated that this is an exploratory research effort, and therefore, the success of the implementation should not be judged solely on whether it immediately surpasses existing state-of-the-art solutions. The

primary objective is to evaluate the feasibility of a novel approach that could support future advancements in the field. The values presented in Table 3.2 were determined by defining a baseline. The results of this baseline are discussed in Section 4.1, and the final values were selected in alignment with those results and through close communication with the client.

The numerical values presented in Table 3.2 were derived through different methods, many of which rely on results obtained during later stages such as the proof of concept and the development of the baseline. These metrics are explained below.

The RMSE values for position and velocity, as well as the standard deviation of position RMSE across different scenarios, were derived from the baseline implementation presented in Section 4.1. This baseline, based on MPC, serves as a reference point for the performance range of a state-of-the-art method. However, since the proposed approach is novel and lacks a direct counterpart in the literature, comparing it strictly against a mature method like MPC is not particularly meaningful. For this reason, target RMSE values were set based on the initial experimental results, proposing an improvement of at least 75 percent in position RMSE and 66 percent in velocity RMSE, relative to the first functional implementation, which was trained for 10 million steps without constraint enforcement. RMSE was chosen as the primary metric because it penalizes larger errors more heavily than MAE or MSE, which is crucial for tasks where occasional high-magnitude deviations are particularly undesirable. Additionally, RMSE offers more interpretability in terms of physical units than DTW, which is better suited for aligning temporal sequences rather than quantifying control accuracy.

The computation time goal was defined relative to the MPC baseline, expecting a significant improvement. In particular, the target was set to approximately 3 percent of the average duration observed in the strictest MPC setup and 6 percent of the duration in the fastest scenario. The target for the standard deviation of inference time was selected with the intention of maintaining step durations within the working frequency of the control system (250 Hz, or 4 ms). While this does not reflect deployment hardware directly, it serves as a reference point to suggest that if performance is achieved on general-purpose hardware, further improvements would be feasible in optimized deployments. This value was also determined after analyzing the timing distributions of early experiments.

The percentage of control commands within the safe operating range is defined as a safety-critical metric. Ideally, no control signal should exceed the specified limits. However, a small margin of tolerance is allowed, acknowledging that this is a novel implementation and there may still be room for improvement. This flexibility is particularly justified if stricter enforcement of the constraint would significantly compromise the system's trajectory tracking performance.

The estimated number of weeks required to adapt the system for robots with more degrees of freedom was defined based on client input. This value reflects the anticipated effort needed should the project be extended in a future work, involving additional DoF and new implementation work.

The standard deviation of position RMSE across different scenarios was motivated by the need to ensure consistent performance under variable conditions. The defined target corre-

sponds to a 50 percent improvement over the initial scenario.

Finally, the RMSE change between simulation and real-world deployment was introduced as a key metric to determine the final viability of proposed solutions. Although the proof of concept in this project does not include a physical implementation, the client specified the acceptable thresholds for this metric in order to consider a physical deployment successful in future work.

3.1.4 Stage 4: Exploration of Options

Functional decomposition enables the division of a system into key components that can be analyzed and optimized independently. This strategy facilitates the identification of potential issues, improves development organization, and allows for assessing the impact of each element on the overall system performance. For this project, functional decomposition has been applied to distinguish two main components: the hardware on which the system runs, and the software executed within it.

Building upon the high-level description of the solution presented in Figure 1.1, Figure 3.2 provides a more detailed view of the controller block. The diagram illustrates a software module encapsulated within a hardware module, representing two distinct domains that must be addressed. This representation highlights the hierarchical relationship between the two: the software operates within and depends on the hardware platform, and together they form the complete control system.

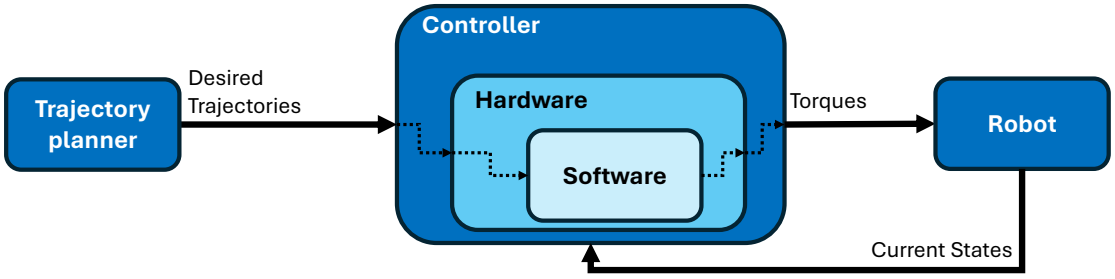


Figure 3.2: Functional decomposition of the problem.

Hardware for System Execution

The hardware involved in the project plays a critical role in its performance and feasibility. It is essential to define the required processing power and memory to ensure that the execution and development of the control algorithm is efficient and reliable. This section focuses specifically on the hardware needed for the control software. All other hardware components, such as sensor acquisition modules, actuator drivers, low-level motor controllers, and the full mechanical system including joints, links, and the end-effector, are assumed to be already covered and do not require modification to support the proposed solution. This assumption is based on the fact that this work relates only to the controller, and that the system should

ideally operate using sensor configurations that are already standard in industrial settings, as established on need number 9.

It is worth mentioning that this section is not the main focus of the work, as the context involves research aimed at proposing an innovative software concept. This concept must still go through several stages before it can be effectively integrated into a real production system. Nevertheless, it is important to anticipate the feasibility of its implementation.

In addition to the requirements inherent to the nature of the problem, needs 4 and 10 also influence the selection of the hardware device for this solution. Need 4 states that the DS must operate in real time within latency constraints, which is critical and must be taken into account when selecting appropriate hardware. Meanwhile, need 11 highlights the requirement for transferring the control policy from simulation to real hardware. To enable this, the system must be capable of operating at an appropriate frequency and leave sufficient margin for possible demands increase due to additional factors such as transmission delays, processing delays in low-level components, and other system latencies.

The rapidly evolving nature of hardware designed for AI applications should be noted. Due to high demand, the emergence of newer, more affordable, and more efficient hardware is inevitable. Therefore, this analysis is based on the most up-to-date information available at the time this work was conducted.

The GoFa CRB 15000 currently operates using ABB's proprietary OmniCore C30 controller. However, due to the proprietary nature of this device, detailed specifications such as its internal processor architecture, memory layout, or computational limits are not publicly available [74]. While ABB does provide an overview of its capabilities, such as high-precision motion control, energy-efficient operation, and broad compatibility with industrial protocols, the underlying hardware remains undocumented.

The required specifications of a controller are tightly coupled with the complexity and demands of the tasks it must perform. In this context, the focus is on evaluating appropriate hardware solutions to support the winning proposal from the 'Software for System Execution' 3.1.4 domain. This proposed solution is based on machine learning, and introduces a clear distinction between two phases: training and inference [75].

During the training phase, the model is optimized using extensive datasets and repeated updates of its internal parameters. This phase is computationally intensive, requiring high-performance hardware with substantial parallel processing capabilities and memory bandwidth [76]. Meanwhile, the inference phase involves deploying the trained model with fixed parameters to generate control signals based on incoming data. Inference is significantly less demanding and can be executed on embedded systems or edge devices with limited resources and form restrictions [77].

Because of these different requirements, the exploration of hardware solutions is split into two sections: Training Hardware and Inference Hardware.

Training Hardware

Training deep neural networks, particularly for control applications involving high-dimensional

inputs and temporal sequences (such as GRU from the wining proposal in the 'Software for System Execution' 3.1.4 section), requires considerable computational resources. The primary needs for training hardware include high-throughput parallel computing, large memory capacity and fast I/O performance for loading datasets and logging outputs.

This level of performance is typically achieved using dedicated workstations, cloud-based Graphics Processing Unit (GPU) instances, or high-performance computing (HPC) clusters. Characteristics related to capabilities to work on edge, offline or under energy restrictions are not important at this stage. The proposals will be evaluated based on the following items:

- Training speed and performance: Describes how fast the hardware can train models, including metrics like float point operations per second (FLOPS), memory capacity, and benchmark results relevant to deep learning workloads.
- Price-to-performance ratio: Compares the hardware's cost to its training capability, helping to evaluate whether its performance justifies the expense for the use at the scale of the project. This characteristic is also evaluated in the economic feasibility study section 3.3.
- Software ecosystem compatibility: Describes how well the hardware supports major ML frameworks and tools, indicating how easily models can be implemented and deployed.

Proposal 1: Google Cloud Tensor Processing Units (TPUs)

Training Speed & Throughput

Google's TPUs are specialized hardware accelerators optimized for mixed-precision training using Google's custom Brain Float 16 format (bfloat16). The TPU v4 delivers over 275 TFLOPS of compute performance for deep learning tasks, primarily through its high-throughput matrix multiplication capabilities [78] [79].

Each TPU v4 chip is equipped with 32 GB of High Bandwidth Memory (HBM), using HBM2 or HBM2e technology to reach memory bandwidths of approximately 1.2 TB/s [80].

Google's TPU pods or clusters have consistently set benchmarks for training speed. A Cloud TPU v3 Pod comprising 1024 chips was able to train models such as ResNet-50 and Transformer architectures to target accuracy in under two minutes each, leading the MLPerf v0.6 leaderboard [81]. TPU v4 further improves scalability and performance: a full TPU v4 Pod (4096 chips) delivers up to 1.1 exaFLOPS of bfloat16 compute power, enabling the training of extremely large models like PaLM (540 billion parameters) using just two pods [80].

Price-to-Performance

TPUs are only available via Google Cloud, and pricing is premium. However, Google has reported better cost-efficiency on large language model (LLM) training with TPU clusters vs. GPU clusters [82].

For TPU v4, the on-demand pricing is around $3.22 per chip-hour, resulting in a total of approximately $12.88 per hour for a full 4-chip host [83]. Preemptible instances, which can be interrupted by the cloud scheduler, offer a cost reduction of about 70%, bringing the price down to roughly $0.97 per chip-hour, or $3.90 per hour for the full 4-chip configuration [83]. Each TPU v4 chip includes 32 GiB of HBM [79], for a total of 128 GiB per host. TPU v4 instances are typically provided in configurations such as 8 chips or more, and benefit from high-speed internal interconnects that enable efficient multi-chip communication [84].

While Google has demonstrated higher cost-efficiency of TPU clusters compared to GPU-based alternatives in large-scale training workloads such as large language models (LLMs) [82], this work does not operate at that scale, nor does it follow the same model structure that TPUs are particularly optimized for. In a full-scale deployment, it would likely be necessary to conduct multiple training runs for hyperparameter tuning, but the system is not expected to process large data volumes or varying formats comparable to those used in LLM or natural language processing (NLP) applications. Thus, the full benefits of TPU scalability and cost-efficiency may not be realized in this specific context.

Framework Compatibility

Google's TPUs offer native support for TensorFlow and JAX, providing seamless integration and performance optimization through Google's XLA compiler [85]. This tight coupling ensures efficient execution of machine learning models on TPUs [85].

PyTorch support on TPUs is available via the PyTorch/XLA library. While this enables PyTorch-based models to run on TPUs, it often requires additional configuration, tuning, and sometimes code modifications to achieve optimal performance [86].

Proposal 2: NVIDIA GPU A100

Training Speed & Throughput

NVIDIA's A100 (Ampere architecture, 7nm) is a versatile accelerator optimized for mixed-precision deep learning. The 80 GB SXM version delivers up to 312 TFLOPS of BF16/FP16 performance using Tensor Cores, and up to 156 TFLOPS of TF32 for general-purpose workloads without sparsity [87, 78]. Sparsity is the proportion of zero-valued elements in a tensor, indicating how much of the data can be skipped during computation [88]. This makes it one of the most powerful general-purpose GPUs for training tasks, particularly those involving large batch sizes and dense tensor operations.

The A100 is equipped with 80 GB of HBM2e memory and offers up to 2.0 TB/s of memory bandwidth, enabling sustained throughput for data-intensive training workloads [87].

In MLPerf v1.0 benchmarks, an 8×A100 system (NVIDIA DGX A100) trained BERT-Large to target accuracy in 28.7 minutes, while ResNet-50 reached 75% top-1 accuracy on ImageNet in just ∼37 seconds [89]. A100 clusters demonstrated near-linear scalability: for instance, ResNet-50 was trained in 24 seconds on a large A100 setup, equivalent to ∼219 minutes on a single A100 [90]. Across various MLPerf tasks, the A100 set multiple training records, underscoring its balanced performance in both single-device and multi-GPU scenarios [90].

Price-to-Performance

NVIDIA A100 GPUs are widely available on major cloud platforms, with on-demand pricing typically ranging from $2.90 to $3.80 per hour, depending on the configuration and provider [91] [92]. Preemptible instances offer significantly lower rates, as low as $1.15/hour for the 40GB model and $1.57/hour for the 80GB version [93]. All A100 instances benefit from high-speed interconnects and support multi-GPU scaling without additional bandwidth charges.

Despite its price, its strong performance, mature software ecosystem, and broad compatibility make it cost-effective for diverse training tasks. The A100 offers low per-hour costs and strong price-performance, especially for mid to large-scale workloads that benefit from GPU flexibility and efficient usage [82].

Framework Compatibility

The A100 is fully supported across major ML frameworks including PyTorch, TensorFlow, and JAX, all optimized via CUDA and cuDNN. This allows for seamless implementation of models. NVIDIA's ecosystem and tooling make the A100 a robust platform for research and development. The A100 has native support for all major ML frameworks: PyTorch, TensorFlow, and JAX. These frameworks are fully optimized through CUDA, cuDNN, and NVIDIA's software stack [94]. Features like automatic mixed precision, TensorRT, and NCCL allow seamless deployment and multi-GPU scaling [95]. It is worth noting that, in the current technological landscape, NVIDIA remains the dominant leader in machine learning and GPU computing. As such, adopting NVIDIA hardware represents a strategically sound choice for long-term compatibility, performance, and ecosystem support.

Proposal 3: NVIDIA GPU H100

Training Speed & Throughput

NVIDIA's H100 GPU (Hopper architecture, 4nm) is optimized for high-throughput training, particularly in lower-precision formats . The 80 GB SXM version delivers up to 990 TFLOPS of BF16/FP16 compute and supports up to 3.958 PFLOPS in sparsity-optimized FP8 precision, enabled by NVIDIA's Transformer Engine [82, 96]. This marks a generational leap over the A100, with up to 6 times greater FP16 throughput and significantly improved efficiency for transformer-based models.

The H100 is the first GPU to use HBM3 memory, providing 80 GB of capacity and achieving ∼3.35 TB/s of memory bandwidth [96]. This high memory throughput is critical for feeding its expanded tensor core array during large-scale model training.

In MLPerf Training v3.0, the H100 set new records across several benchmarks. A cluster of 10,752 H100 GPUs was used to train a 175B-parameter model in just 4 minutes, demonstrating near-linear scaling [82]. Compared to the A100, an 8-GPU H100 system achieved $\sim$ 2 times faster training times on workloads such as BERT and ResNet [97]. These gains are largely attributed to the Hopper architecture's increased FLOPS, larger cache, and FP8 acceleration capabilities.

Price-to-Performance

The NVIDIA H100 GPU is significantly more expensive than prior-generation accelerators, both in hardware cost and cloud usage. On-demand cloud pricing ranges between $6.80 and $6.98 per hour per GPU, depending on the provider and region [92]. However, as usual, preemptible alternative pricing offers a more accessible alternative, with rates as low as $2.25 per hour on Google Cloud [93].

H100 instances typically include 80 GB of HBM3 memory and benefit from ultra-high bandwidth networking. Instance types on both Google Cloud and Azure support 1, 2, 4, or 8 GPUs, and data egress fees follow standard cloud billing practices [82].

While H100 offers exceptional training speed, reducing job completion time and potentially lowering total compute hours, the high hourly cost makes it a less cost-effective option for small-scale to medium-scale projects. Its price-to-performance benefits are best realized in large-scale, time-critical workloads, such as training large language models or multi-billion parameter transformer architectures.

Framework Compatibility

NVIDIA's H100 GPU is fully supported by all major deep learning frameworks, including PyTorch, TensorFlow, and JAX. It maintains backward compatibility with CUDA and cuDNN, ensuring seamless integration with existing models and workflows. The H100 introduces advanced support for FP8 precision, available through NVIDIA's Transformer Engine, which is natively integrated into PyTorch and TensorFlow distributions [98]. Using FP8 effectively may require explicit configuration of model layers and training loops [99].

This extensive support makes the H100 well suited for both conventional deep learning tasks and cutting-edge architectures. The mature software stack and driver ecosystem further enhance its ease of deployment and optimization [96]. As noted in the previous proposal, NVIDIA's well-established position in the machine learning ecosystem offers a degree of long-term security, as any acquisition of their hardware is expected to be supported and maintained by the manufacturer for the foreseeable future.

Table 3.3 summarizes the data from the three proposals.

Table 3.3: Comparison of Training Hardware: Google TPU v4, NVIDIA A100, and NVIDIA H100

Metric	Google TPU v4	NVIDIA A100	NVIDIA H100
Peak Compute (TFLOPS)	275	312	990
Memory Capacity	32 GB HBM2	80 GB HBM2e	80 GB HBM3
Memory Bandwidth (TB/s)	1.2	2.0	3.35
On-Demand Price ($)	3.22	2.90-3.80	6.80-6.98
Preemptible Price ($)	0.95	1.15 - 1.57	2.25
TensorFlow Support	Yes	Yes	Yes
PyTorch Support	Partial (via XLA)	Yes	Yes

Among the evaluated hardware options, the NVIDIA A100 offers the best balance between performance, cost, and software compatibility, making it the most practical and sustainable

Table 3.4: Comparison of Training Hardware: GTX 1650, RTX 3070, and RTX 4090

GPU Model	FLOPs (FP32)	Memory Bandwidth
GeForce GTX 1650	∼3.0 TFLOPS	128 GB/s
GeForce RTX 3070	∼20.3 TFLOPS	448 GB/s
GeForce RTX 4090	∼82.6 TFLOPS	1,008 GB/s

choice. With a peak compute performance of 312 TFLOPS and a memory bandwidth of 2.0 TB/s, it provides sufficient computational power for most mid to large-scale training tasks, while remaining significantly more cost-effective than the newer H100. The A100 also supports both TensorFlow and PyTorch natively, offering flexibility without requiring additional abstraction layers or workarounds, as is the case with TPUs.

In contrast, while the NVIDIA H100 offers considerably higher performance (up to 990 TFLOPS and 3.35 TB/s bandwidth), this comes at nearly double the cost per hour. Unless the use case demands cutting-edge computer capabilities, such as training extremely large language models, the marginal performance gains may not justify the increased expense. On the other hand, Google TPU v4, although competitive in raw performance, lacks full native PyTorch support and requires adaptation through XLA, which can hinder ease of development and debugging.

The A100 is therefore identified as the recommended GPU for any future extension of this project, particularly in scenarios where sufficient financial resources are available to support its use. However, during the actual execution of this work, three different GPUs are used, based on availability. The GeForce GTX 1650, installed in the researcher's personal computer, was used for preliminary development. The RTX 3070 was accessed through the high-performance computing (HPC) infrastructure provided by the Chair of Applied Mechanics at TUM. Lastly, the RTX 4090 was made available by the client via remote access to a server at ABB. Table 3.4 shows the capabilities of the used GPUs.

Inference Hardware Inference, by contrast, has significantly lower computational demands. Once the model is trained and its weights are frozen, inference involves only forward passes through the network. The main requirements are:

- Sufficient CPU or low-power GPU/accelerator support.
- Low-latency and real-time processing capabilities.
- Convenience factors for physical deployment on the robot or nearby.

These tasks can often be handled by embedded systems, industrial PCs, or edge AI devices, which are cost-effective and easy to integrate into existing robotic platforms. To set a numerical reference of the needs of the implemented system, A.8 representing the computational cost of a forward pass of a neural network is used, while taking the values from the implemented final policy described in 3.2.2.

A network consisting of a GRU with input size 4 and hidden size 256, followed by fully connected layers of dimensions 256 to 128, 128 to 64, and 64 to 2, results in an estimated computational cost of approximately 440,000 FLOPs per inference.

Assuming a control rate of 250 Hz, the required computational throughput becomes:

$$\text{GFLOPs/sec} = \frac{440,000 \times 250}{10^9} = 0.11 \tag{3.1}$$

Proposal 1: High-Performance CPU (Standard x86 Solution)

A modern Intel i7 CPU (or similar AMD Ryzen) is a straightforward solution. Such CPUs offer strong performance and support real-time OS. With optimized math libraries, the policy can run entirely on the CPU without offloading, avoiding any I/O latency. This approach leverages the robot's onboard PC directly.

Compute Performance

High-end x86 CPUs can deliver several hundred GFLOPS of FP32 throughput. For instance, the Intel Core i7-12700K, a 12th-generation processor, has a peak FP32 performance of approximately 691.2 GFLOPS, while the i7-12700F variant achieves around 403.2 GFLOPS [100], which significantly exceeds the computational demand of 0.11 GLOPS for a policy with 283,000 parameters. This leaves a considerable performance margin for real-time inference.

Latency Previous studies have demonstrated that assigning inference threads to specific CPU cores and shielding them from standard operating system scheduling can significantly reduce latency variability, lowering worst-case latency to just a few tens of microseconds [101]. This solves one of the major disadvantages of using regular CPU cores for inference tasks. However, implementing this method might require more work hours.

Implementation Factors

Thermal Design Power (TDP) refers to the maximum amount of heat (in watts) that a processor's cooling solution must dissipate under typical workload conditions [102]. While general-purpose CPUs such as the Intel i7 have higher TDPs than specialized accelerators, they remain well within the power and thermal budgets of a robotic platforms. For instance, industrial PCs commonly employ 35–65 W CPUs without requiring specialized cooling solutions [103].

A representative example is the Intel i7-9700TE, an 8-core processor with a 35 W TDP, commonly used in compact edge AI controllers such as the Lanner EAI-I730 and LEC-2290B [104] [105]. These systems typically use small form factor industrial PCs or single-board computers already integrated into the robot, without requiring extra hardware. During inference, power consumption usually stays within a few tens of watts, which can be handled efficiently using standard heatsinks and small built-in fans. This design remains thermally manageable in typical robotic environments [104].

From a software perspective, an x86-64 CPU running Ubuntu Linux is fully compatible with PyTorch. This eliminates the need for model conversion or additional runtime frameworks

[104]. PyTorch models can run natively on the CPU using libraries such as OpenBLAS or Intel MKL, which simplifies the development pipeline [106]. The same code used for training can be directly deployed for real-time inference, ensuring a streamlined workflow and reducing integration complexity .

Proposal 2: NVIDIA Jetson Orin NX

The Jetson Orin NX is a compact system-on-module that includes an 8-core ARM CPU and an NVIDIA GPU (Ampere architecture) for AI acceleration. It delivers server-class inferencing performance in a small, power-efficient package, making it ideal for on-robot deployment. The policy model can be executed on the GPU component for high-throughput low-latency inference.

Compute Performance

The Orin NX (16 GB model) provides up to 100 TOPS for 8-bit signed integer (INT8) of AI computation [107]. This equates to roughly to 1.9 TFLOPS FP32 of GPU throughput [108] considerably more than needed for the conservatively approximated requirement established in 3.1. In effect, it can easily handle the policy in real time along with future additional tasks or overlooked requirements.

Latency

Owing to its GPU and Tensor Cores, the Jetson can compute the policy inference with very low latency. In practice, GPU acceleration significantly outperforms CPU for larger models. For example, Jetson Xavier NX achieved ∼20 ms inference on a CNN vs 50 ms on a laptop CPU [109]. The platform is designed for real-time performance [110]. The Orin NX also features 2 dedicated NVDLA engines (deep learning accelerators) that can be used for offloading inferencing with more deterministic timing when needed [107].

Implementation Factors

The NVIDIA Jetson Orin NX module features configurable power modes between 10 W and 25 W, enabling trade-offs between energy efficiency and performance [107]. A recent update via JetPack 6.2 also introduced a new 40 W mode, offering even greater performance headroom [111]. Even at its 40 W setting, the module remains within a typical robot PC's thermal budget and can be cooled passively or with a small fan [107]. In real deployments, running a policy model would typically draw only a fraction of the module's peak power capacity [107] [108].

The Orin NX system-on-module measures only 70 mm × 45 mm [107], making it smaller than a credit card. It can be integrated onto a carrier board the size of a palm. This compact and lightweight form factor enables deployment on payload-constrained platforms such as drones [107]. Thermal management usually requires only a heat spreader or compact heatsink, making it ideal for embedded use in constrained environments.

Jetson runs a Linux-based OS derived from Ubuntu, provided through NVIDIA's JetPack SDK. It includes full support for CUDA and PyTorch [112]. NVIDIA provides ARM64-optimized builds of PyTorch, allowing developers to deploy models directly without con-

version. Additionally, the full NVIDIA AI stack (CUDA, cuDNN, and TensorRT) is pre-integrated [112], and robotics frameworks such as ROS and NVIDIA Isaac are officially supported. In summary, the policy controller can run on Jetson with minimal changes, benefiting from GPU acceleration while using standard Linux and PyTorch tools [112].

Proposal 3: Google Coral Edge TPU

The Google Coral Edge TPU is an application specific integrated circuit (ASIC) based neural accelerator that can offload inference from the main CPU. It is ultra-compact and efficient, making it attractive for low-power or supplemental AI at the edge. The Coral Edge TPU is typically used via a USB 3.0 dongle or M.2 module attached to a host system [113].

Compute Performance

The Coral Edge TPU delivers 4 TOPS of performance (8-bit operations) at peak [114]. This means it can perform with an efficiency of about 2 TOPS/W (each TOPS ∼0.5 W) [114] [115]. In practice, 4 TOPS is sufficient for the implemented policy.

It is documented that when paired with a CPU, the Coral is capable of providing a ∼10 times speedup on vision models [115]. For example, it runs MobileNet v1 in 2.4 ms versus 53 ms on a CPU [113]. This suggests that the policy controller would execute with sub-millisecond latency on the Edge TPU, leaving significant compute headroom.

Latency

The Edge TPU is engineered for real-time inference with minimal latency. It utilizes an on-chip systolic array to pipeline data efficiently, thereby avoiding memory bottlenecks. For standard neural networks, such as MobileNet, the inference delay is remarkably low, often in the range of hundreds of microseconds, with only minor overhead introduced by USB or M.2 data transfer [116].

Unlike GPUs, which often employ time-slicing, a method where multiple processes share GPU resources by allocating them time slots, the Edge TPU operates deterministically. This means it processes tasks in a predictable manner without the variability introduced by resource sharing, leading to consistent and reliable inference times [117].

A critical consideration is the model size relative to the Edge TPU's on-chip memory. The Edge TPU is equipped with approximately 8 MB of SRAM dedicated to storing model parameters. If a model exceeds this memory capacity, it must access external memory, which can introduce latency and reduce performance. However, a model with 283000 parameters is sufficiently small to fit entirely within the Edge TPU's on-chip memory, ensuring efficient, one-shot computation without the need for external memory access [118]. The memory required to store the policy can be conservatively estimated by assuming that each parameter occupies one byte, consistent with 8-bit quantization. Under this assumption, a 283,000-parameter model would require approximately 283 KB of memory, which corresponds to less than 5% of the Edge TPU's total 8 MB on-chip SRAM.

Implementation Factors

The Coral Edge TPU accelerator consumes approximately 2 W under typical usage, with

peak power reaching around 2–4 W at maximum clock frequency [114] [115]. It is powered via either USB or the M.2 slot, and generates minimal heat, even under sustained high-throughput workloads. However, the manufacturer does note that the device may feel hot during heavy use [115]. Importantly, its low power consumption means no active cooling is required and it can operate passively within the robot's ambient thermal environment. This makes it appropriate for embedded applications, staying within thermal and power budgets.

The Edge TPU is available in very compact form factors. The USB accelerator measures approximately 65 × 30 × 8 mm [115], similar to a standard USB flash drive, while the M.2 A+E key module is a slim internal card. Both variants weigh only a few grams. This ultra-compact device allows for straightforward integration: it can be plugged directly into a USB 3.0 port or mounted onto a carrier board. Furthermore, multiple Coral TPUs can be used in parallel to scale performance. The small size and low thermal output make it ideal for space-constrained platforms with limited or no airflow.

The main limimtation lies in the workflow. The Edge TPU does not support native execution of PyTorch models. The policy model must first be exported to the TensorFlow Lite format with full INT8 quantization and then compiled using Google's Edge TPU Compiler [114]. This compilation step ensures that only supported operations are used. The Edge TPU primarily supports CNN layers and basic operations; however, recent versions of the compiler have introduced support for quantized unidirectional LSTM and GRU layers [119].

Once compiled, inference is performed via the Coral TensorFlow Lite runtime or the PyCoral API on a Linux system. The accelerator is compatible with Debian and Ubuntu-based operating systems [115]. For example, deployment can be achieved on devices like the Raspberry Pi or other Linux-based embedded systems by installing the Coral runtime and drivers. In summary, the Coral Edge TPU offers excellent real-time inference performance, but this comes at the cost of additional deployment complexity.

All three hardware options presented in Table 3.5 are technically viable for deploying the trained inference policy. However, they differ significantly in terms of performance, power consumption, integration effort, and cost—factors that become increasingly relevant when transitioning from prototyping to industrial deployment.

While Proposal 1 (x86 CPU) offers seamless integration and native PyTorch support, its relatively high power consumption and large form factor make it less suitable for embedded or mobile applications. Proposal 2 (Jetson Orin NX) delivers strong performance and supports the CUDA ecosystem, but it comes at a significantly higher cost. In contrast, Proposal 3 (Coral Edge TPU) offers a compact and power-efficient platform at a substantially lower price point. This makes it particularly attractive in the context of serial production, where hardware cost directly affects the final unit price of the robot.

For this reason, the Coral Edge TPU is selected as the most appropriate long-term deployment option. Its small footprint and low power draw are well-aligned with industrial constraints, and its cost advantage is critical for scaling beyond a single prototype. Due to proprietary restrictions, the final integration and testing on ABB's hardware will need to be performed internally by ABB engineers. This will involve validating whether the Coral TPU

Table 3.5: Comparison of Inference Deployment Proposals

Factor	Proposal 1: CPU (x86)	Proposal 2: Jetson Orin NX	Proposal 3: Coral Edge TPU
Compute Performance (TFLOPS FP32)	∼0.7	∼1.9	∼4
Thermal Design Power (W)	35-65	10-25	2-4
Form Factor	Embedded PC or SBC	70 mm × 45 mm module	USB: 65 mm × 30 mm × 8 mm
Software Compatibility	Native Pytorch	PyTorch (ARM64) and full CUDA stack	Requires conversion
Integration Complexity	Very low	Low	Moderate to high
Determinism	High	High	High
Memory Fit for Policy	Yes in RAM	Yes in GPU RAM	Yes in SRAM
Price (€, approx.)	300-600 [120]	700-800 [121]	100-150 [122]

can reliably execute the trained policy after quantization and conversion to TensorFlow Lite. Only after this internal verification will its implementation be considered finalized.

Software for System Execution

This section discusses different proposals regarding the algorithms and methods used to control the robotic manipulator. Specifically, it focuses on the controllers responsible for estimating the control signals required to drive the robot manipulator based on its current state and the desired trajectory (defined in the joint space).

Proposal 1: Classical MPC

This proposal considers the use of a classical MPC strategy based on the assumption of known dynamics of the robotic manipulator. At every control step, the MPC solves a constrained optimization problem to determine the control inputs that minimize a predefined cost function over a finite prediction horizon. This structure allows the controller to explicitly optimize trajectory tracking performance while ensuring that both state and input constraints are satisfied at all times. The effectiveness of this approach relies on the availability of a sufficiently accurate system model and the computational resources necessary for solving the optimization problem in real time.

MPC is widely regarded as the industry standard due to its reliability and conservative behavior. Its working principle was discussed in section 2, where its capabilities were highlighted.

In terms of innovation, MPC is a well-established and extensively studied method. Its maturity makes it difficult to introduce new developments within the scope of this project. Although it does not inherently incorporate artificial intelligence, some recent approaches explored in the literature [63] propose replacing the traditional ODE solver with neural network-based approximations to improve computational efficiency. Or methods such as the ones proposed in [52] and [53] extracts new trajectories of a LQR based on overall objectives and data of previous trajectories with the help of the k-nearest-neighborhood algorithm. However, these methods still follow the original structure of MPC, which is based on the prediction of future system states and the optimization of a cost function.

MPC remains the reference in trajectory tracking in industrial applications due to its high precision and robustness. Nevertheless, its performance is strongly dependent on the accuracy of the system model. When the model does not capture the true dynamics of the robot adequately, control performance can be significantly affected.

As discussed in 2 , the computational complexity of MPC is significant, as it requires solving an optimization problem at every control step. This becomes particularly demanding for large-scale systems or when using long prediction horizons [27]. MPC often depends on nonlinear optimization problems that must be solved iteratively. These problems introduce variability in computation time, making it difficult to guarantee real-time feasibility or at least limiting the range of possible control frequencies.

One of the main strengths of MPC is its ability to guarantee constraint satisfaction. Since constraints are directly embedded into the optimization problem, the controller ensures that the system will never exceed the defined limits during operation.

The controller must be capable of handling systems with at least two degrees of freedom. MPC does not present any fundamental limitation in this regard. However, as the number of degrees of freedom increases, the computational complexity of solving the optimization problem also grows. This can lead to challenges in maintaining real-time performance, especially when the system dynamics are highly nonlinear or require fine discretization.

Regarding scalability, MPC only requires accurate models of the systems to be controlled. This makes it suitable for controlling more complex manipulators or systems with a larger number of actuators, provided that reliable models are available. In such cases, trajectory tracking capabilities remain strong, although computational time may become a limiting factor.

The robustness of MPC is one of its key strengths, particularly in handling model uncertainties and external disturbances. Through proper design, MPC can maintain acceptable performance even when the system is subject to deviations from the nominal model. Techniques such as robust MPC formulations, inclusion of uncertainty bounds, and feedback correction mechanisms help the controller remain effective under imperfect modeling or unanticipated changes in system dynamics. However, the level of robustness ultimately depends on the specific implementation choices, including the formulation of constraints and the accuracy of the model used to predict future states.

MPC is fully compatible with the most commonly used sensors in industrial environments. It typically uses position and velocity feedback, and in some applications, force or torque sensing might also be integrated into the control loop.

One important limitation of MPC is its strong reliance on the accuracy of the system model, particularly when transitioning from simulation to real hardware. Discrepancies between the simulated model and the actual system can lead to significant performance degradation, especially when the simulation is based on idealized assumptions that fail to capture real-world complexities. Due to this sensitivity, fine-tuning an MPC controller for deployment on a physical robot can become a complex and time-consuming process, often requiring extensive trial and error through repeated experimental validation.

Proposal 2: LQR-Based Tracking Controller

This proposal focuses on the design of a trajectory tracking controller based on the LQR framework. The common primary objective is to minimize the deviation of the system from a desired reference trajectory while penalizing excessive control effort. The control law is derived from the solution of a quadratic optimization problem, assuming the system is either linear or has been locally linearized along a nominal time-varying trajectory.

To enable effective trajectory tracking, the LQR must be implemented in its time-varying form, where feedback gain matrices are precomputed offline along the entire reference trajectory. During execution, the controller applies these gains according to the current point of

time on the trajectory, allowing it to correct deviations in real time without requiring online optimization. This property makes it particularly suitable for applications where real-time performance is critical and computational resources are limited.

This method is widely used in well-characterized applications, where the system dynamics are known with high accuracy. Its low computational cost makes it especially advantageous in industrial settings where the operation conditions are well controlled and predictable. These are environments in which each component can be reliably modeled, making model-based approaches like LQR feasible. Due to its maturity and extensive use in control literature and applications, it does not represent a significant innovation relative to the current state of the art.

As it relies entirely on a mathematical model, the controller does not incorporate any form of artificial intelligence. Its performance depends strongly on the accuracy of the linearized model used during design. Because the feedback gains are fixed with respect to the nominal trajectory, the controller tends to show limited robustness in the presence of strong nonlinearities or model mismatches, which are common challenges in robotics.

Constraint adherence is not one of its strengths. Although the design encourages small control signals through cost penalization, there is no explicit mechanism to enforce state or input constraints. This makes it unsuitable for applications with strict physical or safety limits unless additional strategies are incorporated.

The controller can be applied to systems with at least two degrees of freedom, and its scalability to more complex systems depends entirely on the quality of the model provided. As long as the dynamics are accurately captured, the method remains valid, although the offline computation of gain matrices may become more demanding.

The controller is compatible with common sensing setups used in robotic systems and can operate using feedback from state variables such as position and velocity. Similar to the previous proposal, transferability from simulation to real hardware is difficult to evaluate independently, as the performance of the controller depends entirely on the fidelity of the underlying model. Since the same model governs both simulation and physical execution, the system does not inherently distinguish between virtual and real implementations, which makes the controller highly sensitive to discrepancies between the modeled and actual system behavior.

Proposal 3: Adaptive Neural Network Controller

This alternative proposes the design of a trajectory tracking controller based on a feedforward neural network trained offline to approximate the inverse dynamics of the robot. The output of the neural network provides a baseline torque signal intended to drive the robot along the desired trajectory. This feedforward term is then corrected in real time using a classical feedback loop, such as a PD or PID controller, to compensate for errors and ensure stable behavior. This hybrid approach combines the function approximation capabilities of neural networks with the robustness of classical control strategies. It is particularly suitable when only partial knowledge of the system dynamics is available and full analytical modeling is

either infeasible or impractical.

This method introduces a higher degree of innovation compared to the previous two proposals. However, as discussed in [64] [65] successful, almost identical, implementations of this approach already exist, and the use of artificial intelligence in this context follows a relatively well-established paradigm. The neural network architecture employed is typically a standard multilayer feedforward network, which, although powerful, does not represent a novel use of AI by current standards.

In terms of trajectory tracking, the controller is expected to follow time-indexed reference points with reasonable accuracy. However, since the structure is not predictive, and relies on real-time correction of errors rather than forecasting future behavior, the tracking performance may not match that of more advanced approaches such as MPC. Additionally, because the neural network replaces the mathematical model, its effectiveness depends heavily on the quality of the training data. A high-quality dataset is essential to ensure that the network can generalize well and produce reliable control signals across the relevant state space.

The computational load associated with this method is expected to be moderate. Once trained, the neural network can produce feedforward torque estimates through a single forward pass, which is computationally efficient. However, if an adaptive component is included to update the network or adjust parameters online, methods such as stochastic gradient descent may be required, increasing the computational demands during execution. Even so, existing implementations have demonstrated that the method is capable of operating in real time, with update rates compatible with standard control loops [123].

This controller does not incorporate any explicit mechanism for enforcing constraints. There is no built-in process to guarantee that the generated control actions or the resulting system states remain within predefined limits. Consequently, the approach relies on the assumption that both the reference trajectory and the network's output will inherently avoid producing actions that violate system constraints. If this assumption does not hold, the system may be exposed to physical or safety violations.

Like the previous proposals, this approach is suitable for two-degree-of-freedom systems, provided that the training dataset represents the full range of dynamics relevant to the task. Scalability to more complex systems is possible, but it requires new datasets that adequately capture the desired behavior of the system to be controlled. The neural network must be retrained or extended accordingly, which can be challenging if data is limited or difficult to obtain.

The robustness of the controller depends significantly on the quality of the training data and the tuning of the feedback gains, which ensure that deviations from expected behavior are quickly corrected. Although formal robustness guarantees are difficult to establish for the neural network component, empirical results from similar implementations suggest that the method can tolerate moderate discrepancies between training and operational conditions.

The method is fully compatible with standard industrial sensors, including position and

velocity feedback, which are typically sufficient for implementing both the neural network input and the classical feedback controller.

Finally, as with the previous proposals, the transferability from simulation to real systems depends entirely on the quality of the model used. In this case, the model is replaced by a neural network, which in turn depends critically on the training dataset. Therefore, successful real-world deployment relies on having a dataset that adequately captures the system's dynamics under realistic conditions. "An advantage of this proposal is the availability of multiple fine-tuning strategies to compensate for minor inaccuracies in the trained neural network.

Proposal 4: Reinforcement Learning: Online Policy

This proposal considers the implementation of an end-to-end control policy trained directly through reinforcement learning, using a recurrent neural network to process temporal information. The RL method is an online policy method such as PPO or TRPO. The policy receives as input a time-series consisting of past states, the current state, and a set of future goals, and directly outputs torque commands for the robot, repeating at each time step. The recurrent structure allows the policy to retain memory of past events, enabling more informed decision-making in dynamic environments. The reward function is manually designed to promote accurate trajectory tracking while addressing additional task-specific objectives.

This approach presents a moderate degree of innovation within the current state of the art. While reinforcement learning has gained considerable traction in robotics, its direct application to low-level control with temporal memory remains a developing area. Nonetheless, introducing methodological novelty in this domain would require a more extensive theoretical and experimental effort than is feasible within the timeframe of this project. Therefore, this proposal is best established as a baseline implementation of reinforcement learning using standard principles and reward design.

Artificial intelligence is the core component of this method, specifically through reinforcement learning, which enables the agent to learn control behavior by interacting with the environment rather than relying on a predefined model. The learned policy maps sequences of observations to control actions in a purely data-driven manner.

Regarding trajectory tracking, the performance of this method remains less documented in the context of joint-space control. Most available research focuses on task-space objectives, where the agent is free to explore and discover strategies that fulfill a task rather than being guided to follow a specific trajectory point by point [124] [125]. As a result, tracking accuracy may vary depending on the richness of the reward function and the agent's exposure to relevant training scenarios. Since this method is not inherently predictive and does not rely on a model of future system behavior, trajectory tracking performance may be lower compared to model-based alternatives.

The computational cost during execution is relatively low, as action selection involves a single forward pass through the trained neural network. While the complexity of this operation

depends on the network size and structure, real-time deployment is generally achievable with standard hardware resources.

A significant limitation of this method is the absence of explicit constraint enforcement. The network does not include built-in mechanisms to ensure that control signals or resulting states respect physical or safety constraints. Without careful reward shaping or external constraint-handling modules, violations are possible, especially in unfamiliar states or under unexpected disturbances.

This controller can be implemented for systems with at least two degrees of freedom, assuming the policy has been trained in a representative environment. Scalability to higher-dimensional systems is feasible, although it may require architectural adjustments and more extensive training to ensure generalization across a larger state-action space.

Robustness in reinforcement learning is a key concern, particularly when transferring policies from simulation to real-world systems. Since policies learn from limited training data, they may perform poorly under unseen conditions such as sensor noise, modeling errors, or environmental variability. To improve generalization, techniques like domain randomization, conservative exploration, and policy regularization are commonly used. While formal robustness guarantees are lacking, well-designed training procedures and safety mechanisms can yield reliable performance in practice.

The feasibility of sim-to-real transfer depends on the similarity between the training environment and the real system, as well as the variability captured during training. Techniques such as domain randomization, dynamics randomization, and real-world fine-tuning are often employed to improve transferability. When these strategies are applied, successful deployment on real hardware is achievable.

Proposal 5: Offline Imitation Learning with Trajectory Demonstrations

This proposal involves training a control policy through behavioral cloning using trajectory demonstrations obtained from expert sources, such as classical planners or human operators. The policy is represented by a neural network and is trained offline in a supervised learning setting. The objective is to learn a direct mapping from observed states to control actions, such as joint torques or target positions, by imitating the expert behavior. Once trained, the policy is expected to generalize the demonstrated skills to new but similar tasks.

This paradigm does not introduce a novel contribution to the state of the art, as it has been implemented in a wide range of environments and tasks across robotics and control, such as in [126]. Nevertheless, it provides a straightforward and well-established approach to incorporating artificial intelligence into robotic control systems, fulfilling the requirement for intelligent decision-making mechanisms in the context of this project.

The trajectory tracking capabilities of this method are highly dependent on the quantity and diversity of the training data. In order to ensure consistent performance, the dataset must comprehensively cover the relevant regions of the state space. A well-known limitation of behavioral cloning is the issue of distributional shift: once the learned policy starts to deviate slightly from the expert demonstrations, it may enter regions of the state space not

covered by the training data. Since it has never observed how to act in these regions, the policy may fail to recover, and errors can accumulate rapidly, degrading performance over time.

In terms of computational requirements, exactly as in Proposal 4, this method is highly efficient during execution. Only a forward pass through the trained neural network is required and real-time execution is generally feasible on typical robotic control hardware.

Constraint adherence is not explicitly handled in this method. There is no guarantee that the learned policy will respect system limitations such as torque bounds or joint limits. The best expectation is that the expert demonstrations themselves adhere to these constraints, and that the cloned policy simply inherits this behavior. However, in the absence of explicit constraint-aware training mechanisms, violations may occur in practice.

The method is suitable for systems with two degrees of freedom or more, depending on the structure of the expert policy and the training dataset. Scalability is possible, provided that the demonstrations cover the dynamics and complexity of higher-dimensional systems. In such cases, the policy must be trained on data reflecting the full range of motion and interactions expected during execution.

Robustness in imitation learning is often limited by the quality and diversity of the expert demonstrations. Since the policy is trained to mimic expert behavior through supervised learning, it may fail because of distributional shift,. To improve robustness, it is common to use techniques such as data augmentation, injecting noise into the inputs, and ensuring a wide range of scenarios are represented in the training dataset. However, because the policy does not receive corrective feedback once deployed, it may still struggle to recover from unexpected states during real-world operation.

The approach is fully compatible with common industrial sensing systems. As in the previous proposals, the policy can operate using standard feedback variables such as joint position and velocity, which are readily available through conventional encoders and estimation filters.

Finally, sim-to-real transfer follows similar considerations as those described in Proposal 4. Since the controller is represented by a neural network, its success in real-world operation depends heavily on how well the training data represents the real system behavior. Techniques such as domain randomization, data augmentation, or real-world fine-tuning can be applied to improve transferability and robustness when deploying the policy outside the training environment.

Proposal 6: GAIL Customized Approach

This approach builds upon Generative Adversarial Imitation Learning (GAIL) [66] and introduces several custom modifications to enhance stability and adherence to predefined motion trajectories. Inspired by the work of Peng et al. in A GAN-like Approach for Physics-Based Imitation Learning and Interactive Character Control [127], the policy network will integrate recurrent architectures such as GRUs to capture temporal dependencies inherent in robotic motion tasks. To mitigate the training instability typical of adversarial imitation learning, the standard single discriminator will be replaced by an ensemble of discriminators, which

improves reward consistency and robustness during training. The observation space will be carefully engineered to include current and past states, as well as a window of future reference states from a predefined trajectory, enabling the policy to follow specific instructions rather than general task-level objectives. The policy will be trained using Proximal Policy Optimization (PPO), where the reward signal is derived from the discriminator ensemble. Expert demonstrations used for training will come from a tuned Model Predictive Controller (MPC) that enforces both stability and constraint satisfaction.

This proposal represents a novel contribution to the field of robotic arm control; to the best of the author's knowledge, no existing work has combined GAIL with recurrent policies, discriminator ensembles, and trajectory-conditioned observations in this context. The approach is firmly grounded in artificial intelligence and aims to match or exceed state-of-the-art performance in trajectory tracking. Despite its complexity, the expected computational load remains low during deployment, as inference consists of a single forward pass through the trained network. As with other AI-based proposals, the computational burden is defined by the policy's architecture, but it remains suitable for real-time control loop execution.

Constraint adherence is further reinforced by leveraging expert trajectories from MPC, an inherently constraint-aware controller, and can be enhanced by including constraint violations as penalty signals within the learned reward function. The system is readily applicable to a 2-DoF robotic setup and is scalable to more complex arms, provided expert trajectories are available for training.

Robustness is enhanced through several key design choices. The use of an ensemble of discriminators provides a more stable and consistent reward signal, reducing the risk of mode collapse and training instability commonly observed in adversarial learning. Additionally, the inclusion of future trajectory references in the observation space helps the policy generalize better to unseen states by making its behavior dependent to structured guidance. The recurrent architecture further improves temporal consistency, making the policy more resilient to transient disturbances. While formal robustness guarantees remain difficult to establish, empirical performance and architectural design choices suggest that this approach offers improved generalization and reliability in dynamic control scenarios.

Like the other AI-based approaches, this method is compatible with common sensor used in robotic arms. Sim-to-real transfer is feasible through the use of domain randomization during training, and the structure of the discriminator ensemble provides robustness to discrepancies between simulation and real-world dynamics.

Table 3.6 presents a summary of all the evaluation criteria applied to each proposal. This table shows a systematic method for quantifying the overall quality of each option. The importance of each criterion is rated on a scale from 1 to 5, where 5 indicates the highest relevance. For each proposal, a rank from 1 to 6 is assigned per criterion, with 1 representing the most favorable outcome. A weighted sum is then calculated by multiplying each criterion's importance by the proposal's rank, and summing the results across all criteria. The proposal with the lowest total weighted score is considered the most suitable solution.

Criteria	Importance	Proposals					
		1	**2**	**3**	**4**	**5**	**6**
Innovation	5	4	4	2	2	4	1
AI Usage	5	5	5	3	1	3	1
Trajectory Tracking	5	1	2	2	5	5	2
Real Time Execution	4	6	1	1	1	1	1
Constraint Adherence	4	1	4	4	4	3	1
Implementable for 2 DoF	4	1	1	1	1	1	1
Scalability	4	2	2	4	4	4	1
Robustness	4	1	1	1	1	1	1
Compatibility with Common Sensors	5	4	4	3	2	3	1
Sim-to-Real Transferability	3	5	5	3	2	3	1
Weighted Total Score		129	126	103	100	124	**48**
Final Position (1=Best)		6	5	3	2	4	1

Table 3.6: Ranking of proposals across key criteria. Lower values indicate better performance. Importance values range from 3 (low) to 5 (high).

Among all the evaluated proposals, Proposal 6 stands out as the most robust and balanced solution. It introduces a novel combination of advanced reinforcement learning techniques, such as recurrent neural networks, discriminator ensembles, and trajectory-conditioned inputs, to address the limitations of standard imitation learning and reinforcement learning approaches. This method leverages the precision of MPC-generated expert demonstrations while incorporating the adaptability and learning capability of neural policies. The structured reward shaping and temporal modeling make it highly suited for complex robotic tasks that require fine control and compliance with dynamic constraints.

The ranking shown in table 3.6 confirms the strengths of Proposal 6. It achieved the best overall score with 98, reflecting its superior performance across criteria. While other proposals like classical MPC or neural network-based controllers perform well in specific areas, they fall short in key dimensions such as scalability or AI integration. In contrast, Proposal 6 offers a well-rounded architecture that meets both current implementation feasibility and future scalability needs.

Its adaptability to different scenarios, compatibility with standard sensors, and structured learning process position it as a strong candidate for real-world deployment. Furthermore, since the method builds on MPC-generated data, it aims to inherit the safety and robustness of traditional controllers while extending flexibility through learning.

3.2 Proof of Concept

Although the long-term goal of this research is to develop a learning-based control system applicable to complex manipulators, such as 6-DoF robotic arms operating in real environments, the proof of concept presented in this work has been intentionally limited down to a simplified and controlled scenario. This decision is grounded in both practical and methodological considerations.

First, a 2-DoF planar robot was selected instead of a higher-degree-of-freedom manipulator. This allows for a more tractable analysis, faster simulation cycles, and direct comparison with symbolic dynamic models. By using only two degrees of freedom, it becomes feasible to validate the learning framework, test the proposed reward structure, and assess the integration of physical constraints without the additional complexity introduced by higher-dimensional control problems.

In line with this, the physical model used for the robot is also intentionally idealized. The Unified Robot Description Format (URDF) file that defines the robot adopts simplified values; this ensures consistency between the URDF-based simulation and the symbolic dynamic model used in MPC formulations. Such equivalence is crucial to ensure that discrepancies in behavior are attributable to control design rather than model mismatches or poor modeling.

Furthermore, the entire validation is performed in simulation using the PyBullet engine, rather than on a physical robot. This decision stems from the lack of access to a dedicated 2-DoF physical platform and also avoids the complexities associated with hardware deployment such as sensor noise, latency, or calibration errors. Simulation allows for rapid prototyping, repeatable experiments, and precise control over environmental variables, which are essential during an early-stage development as this.

The project was therefore structured into a series of progressive stages, each focusing on one key component of the framework. This modular decomposition enables focused experimentation, controlled evaluation, and iterative refinement of the architecture. The stages are as follows:

- Simulation and reference setup
- Framework for unconstrained learning
- Hyperparameters tuning

These stages will be thoroughly discussed as follows.

3.2.1 Simulation and reference setup

In this stage, the expected outcome is the creation of multiple datasets that will serve as training material for the data-driven approach. Figure 3.3 illustrates the high-level pipeline designed for dataset generation. The process begins with the definition of the environment,

where several aspects are specified: the initial and goal positions of the manipulator, the positions, sizes, and shapes of obstacles, the detailed configuration of the manipulator, and the constraints associated with its joints. Subsequently, the trajectory generation phase produces a sequence of states, each consisting of joint angular positions and velocities. These generated trajectories are then used as desired references to be tracked by a MPC. The MPC is configured to operate under both constrained and unconstrained conditions, ensuring that the resulting behavior remains physically feasible.

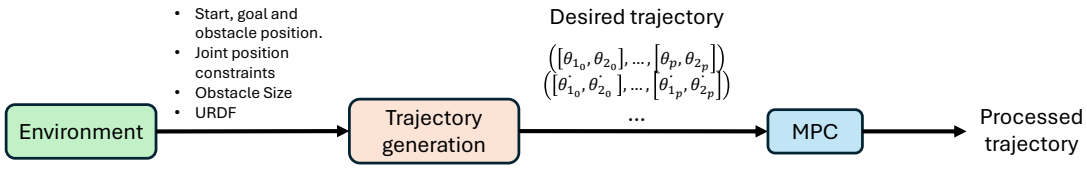


Figure 3.3: General block diagram of the trajectory generation pipeline.

A more detailed representation of the pipeline is shown in Figure 3.4. In this diagram, it can be observed that certain parameters, such as the start and goal positions of the manipulator and the obstacle positions, are randomly assigned within a valid range. This randomization is intended to ensure diversity across the training data. The start and goal positions are constrained to stay within a semicircular region on a single plane, motivated by the physical setup: the manipulator's base is positioned 7 cm above the ground, which limits its feasible workspace before colliding with the floor to a semicircular area in the Y-plane.

In contrast, some parameters are kept constant, such as the joint angular position range and the obstacle size. The joint position limits were set to $[-\pi, \pi]$ to provide maximum initial flexibility for trajectory planning. The obstacle size was selected carefully to challenge the planner to find alternative paths between start and goal points without making the task infeasible; for this reason, obstacles were modeled as cubes with an edge length of 10 cm. To verify that the entire state space is effectively covered by the generated trajectories, additional experiments were conducted, which will be discussed in the coverage test discussed in *Generation of reference geometrical trajectories*.

Finally, there are some parameters that need to be considered as design parameters. The number of segments P and the time differential dt define together how long the trajectories will be. dt is defined to be 0.004 s, since that is the frequency derived from the period of the control loop in the robotic arm (250 Hz) that the client intends to eventually implement the proposal on. And the number of segments is an estimation that allows the motion to be controlled and not so sudden, with enough range of control to compensate for errors and unpredicted behaviors. The parameter N represents the horizon window size, this was set to 10 since that is a commonly accepted value found in the literature [128] [129].

Finally, the objective function defines the focus of the optimization problem. Striking a balance between tracking precision and control effort is crucial. On one hand, prioritizing tracking precision could result in an extremely high control effort that may be unfeasible or even damage the physical hardware; on the other hand, it is necessary to tune how limited

the control effort should be, because a control effort that is too restricted could result in poorly tracked trajectories. It is also possible to enforce explicit constraints such as limits in the position or velocity of the joints or torque limits as well; this option is also used to generate constrained datasets in following stages.

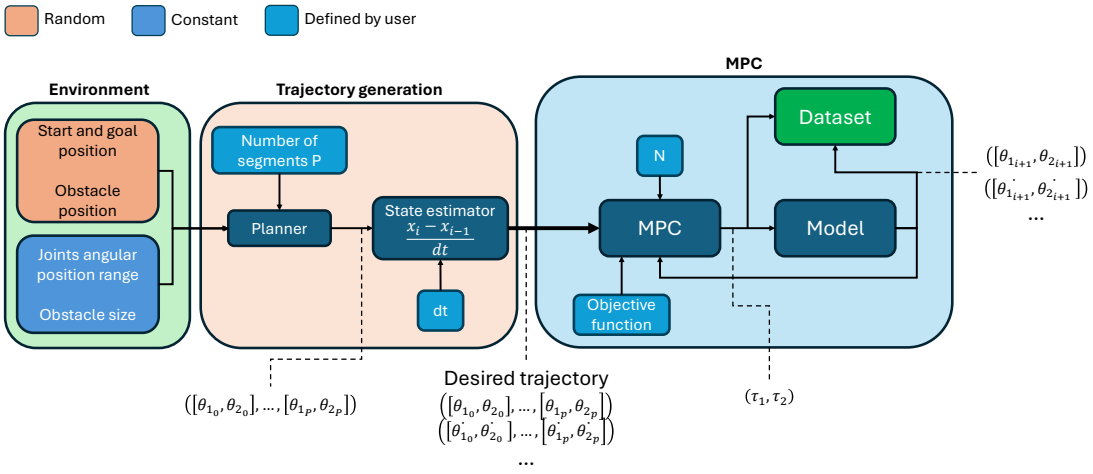


Figure 3.4: Detailed block diagram of the trajectory generation pipeline.

This stage comprises several essential components that form the basis for training and evaluating the proposed control approach. The process begins with the definition of the robot through an URDF file, which offers a standardized way to describe its physical and kinematic properties, including joint limits, link dimensions, and inertial characteristics. Following this, the simulation environment is configured by selecting an appropriate physics engine, and specifying parameters like gravity, time step resolution, and the method for handling collisions. With the robot and environment established, reference trajectories are generated using a motion planning algorithm like RRT* or BIT*, which computes a sequence of collision-free waypoints based on predefined start and goal positions, as well as obstacle information. These planned trajectories are then processed using a forward difference method to estimate joint velocities at each time step, resulting in full state sequences that include both positions and velocities. To improve motion quality and tracking performance, an unconstrained MPC algorithm is applied to the trajectories. Finally, a constrained MPC is used to refine the results further, ensuring that physical constraints such as torque and velocity limits are respected. The outcome is a dataset of dynamically feasible and physically consistent trajectories suitable for use in training and evaluation.

The mentioned steps are described below in detail.

Definition of the URDF

As previously stated, given the complexity of the proof of concept being developed, a 2-DoF robot with revolute joints is selected as the foundation for the simulation and control experiments. In addition, a simplified physical model is adopted; it idealizes several key properties such as the joint damping coefficient, neglects joint friction, and uses significantly low values for mass and inertia. These idealizations are intentional, as they facilitate the

use of symbolic dynamic models and allow for a more direct implementation of classical formulations commonly found in the control literature [14].

The robot consists of two rigid links (link1 and link2), connected by two revolute joints (joint1 and joint2). Both links have a simple rectangular geometry, and the entire configuration is constrained to move within a 2D plane, making it suitable for analytical derivations and simulation-based evaluations. What follows is a list of the main details of the model idealization mentioned before.

Center of Mass Location: For both links, the center of mass is located at the far end of each link, precisely where the next joint begins. This assumption simplifies the inertia terms in the motion equations and aligns the model with widely used formulations in planar robotics, such as the one proposed by [14].

Link Lengths: Both links are set to have a length of 1 meter. This symmetry simplifies the dynamics and improves the interpretability of the results during the control evaluation phase.

Mass and Inertia: The mass of each link is set to 1 kg, and the rotational inertia is deliberately kept low. These values are chosen to reduce the dynamic complexity of the system, making it more responsive and easier to simulate accurately. The low inertia ensures that the resulting accelerations and torques stay within realistic and manageable bounds.

Joint Damping: The damping coefficients associated with the joints are minimal, allowing the system to behave closer to an ideal mechanical structure.

Neglect of Friction: No frictional forces are included in the joint model. This simplification further aligns the physical robot definition with the assumptions made in most symbolic modeling approaches, where friction is generally neglected.

Fixed Base: The robot is mounted on a fixed base, meaning that it does not translate in space. Its only movements are the internal rotations at the joints, consistent with planar manipulators often used in academic control problems.

This idealized configuration supports the design of controllers that assume clean, noise-free dynamics and allows for a fair and focused comparison between model-based and learning-based control strategies. It ensures that any differences in performance are mainly due to the control methodology itself, rather than physical irregularities.

The URDF model used in this work was initially based on the reference implementation provided in [130]. This model was subsequently adapted and extended to conform to the specific kinematic and structural requirements outlined in the previous sections.

Setup of the simulation environment

To model the physical environment and perform collision checking during both trajectory generation and controller evaluation, the PyBullet physics simulation engine is employed [131]. PyBullet is an open-source physics engine that offers real-time dynamics simulation and support for rigid-body systems, making it a suitable choice for simulating robotic manipulators. It provides accurate modeling of contacts, gravity, torque-based actuation, and

joint dynamics, which are essential for validating control policies.

PyBullet was selected over more complex alternatives such as NVIDIA Isaac Gym or ROS-based platforms like Gazebo for several reasons [132] [133]. While these alternatives provide features such as GPU-accelerated simulation, advanced rendering capabilities, or tight integration with robot middleware, such features were not necessary for the scope of this work. The objective here is not high-fidelity sensor simulation or multi-agent environments, but rather a controlled and efficient environment for simulating robot dynamics and evaluating controller performance with respect to trajectory tracking and collision avoidance.

PyBullet's main advantages lie in its ease of use, lightweight architecture, and Python-based Application Programming Interface API, which allows seamless integration with path planning frameworks, learning frameworks, and symbolic modeling tools such as CasADi [134]. Its fast simulation step time and reliable support for torque-controlled joints make it particularly well-suited for iterative control testing and training cycles. Furthermore, PyBullet supports direct loading of URDF files, enabling quick deployment of robot models without additional parsing or preprocessing.

The simulation environment is configured with fixed gravity, a constant simulation time step of 1/250 seconds, and a planar workspace. Each episode is initialized with a fixed base and involves no external disturbances or sensor noise. These idealized conditions are intentionally chosen to allow fair comparisons between control methods and to ensure that discrepancies in performance are caused by the controller design rather than environmental variability.

Overall, PyBullet strikes a balance between physical realism and computational simplicity, making it an appropriate platform for the goals of this project: validating the feasibility of reference trajectories, testing control responses in physically plausible conditions, and ensuring that collision constraints are respected during policy evaluation.

Generation of reference geometrical trajectories

This step in the data generation process involves computing geometrical trajectories that serve as the foundation for reference motion. Each trajectory is defined by a start point, a goal point, and the presence of an obstacle that must be avoided. A motion planner is then employed to compute a collision-free path that connects the initial and final configurations while respecting these spatial constraints. The resulting paths provide the geometrical structure from which full system states (positions and velocities) will later be derived, initially only positions are available.

For this task, two planning libraries were evaluated: PyBullet Planning [135] and the Open Motion Planning Library (OMPL) [136]. OMPL was ultimately selected due to its active development, broad support for sampling-based algorithms, and its established integration into robotic frameworks such as MoveIt [137]. In contrast, PyBullet Planning exhibited limited reliability and frequent inaccuracies, occasionally producing paths that intersected with obstacles or included unfeasible movements, which disqualified it for consistent data generation.

Other alternatives, including CHOMP, and TrajOpt, were also reviewed [138] [139]. While

these offer advanced optimization-based planning capabilities, they are tightly coupled with ROS-based infrastructures and demand more substantial integration effort. Additionally, their extended feature sets, such as continuous-time optimization and constraint handling, exceed the requirements of this preliminary stage.

Given that this planning phase is intended solely to provide reasonable and physically plausible trajectories for subsequent refinement, perfect path optimality is not critical. What is required is a planning framework that can reliably produce feasible collision-free motions to support the generation of training data. OMPL meets this need effectively while maintaining compatibility with the intended workflow and keeping the implementation effort reasonable. The BIT* algorithm was chosen since it was a proven method from the documentation of the library, also this algorithm is not particularly relevant to the work.

To evaluate the spatial coverage of the generated dataset, heatmaps of both the end-effector position and the joint configuration space were analyzed at various sample sizes. As shown in Figures 3.5 and 3.6, the dataset with 200 samples is clearly insufficient to cover the full workspace of the robot. Both the end-effector positions and joint angles display sparse and uneven distributions, which would limit the diversity and reliability of any learning-based models trained on this data.

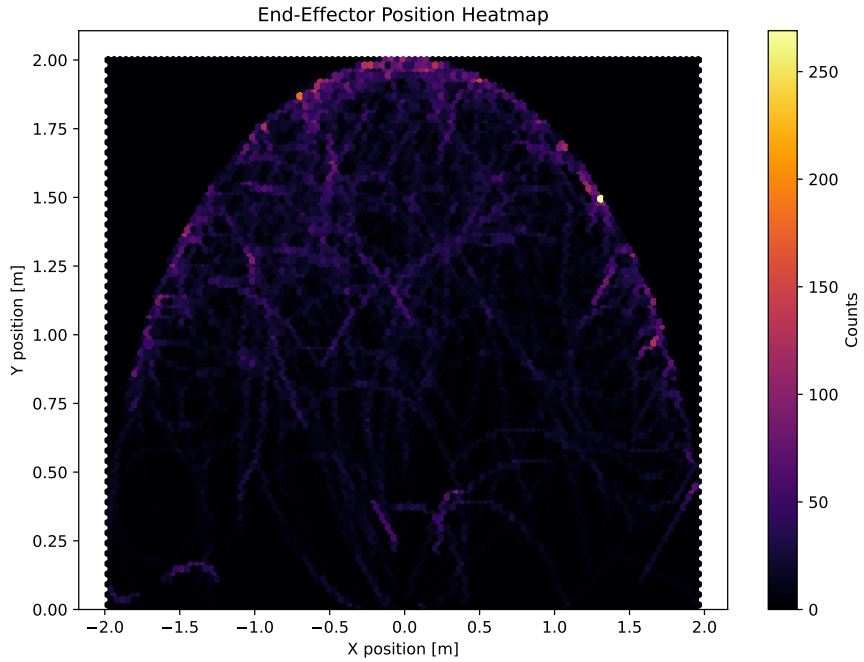


Figure 3.5: Heatmap of the end-effector position in the generated dataset (200 samples).

For this reason, the sample count was increased to 5000, as seen in Figures 3.7 and 3.8. At this size, the spatial coverage is significantly improved, with both the Cartesian and joint spaces appearing densely and uniformly represented.

However, to further improve the dataset's quality and generalization capacity, the number of samples was doubled to 10,000. This decision was not made purely based on spatial coverage, since 5000 samples already provided sufficient representation. Instead, it was driven

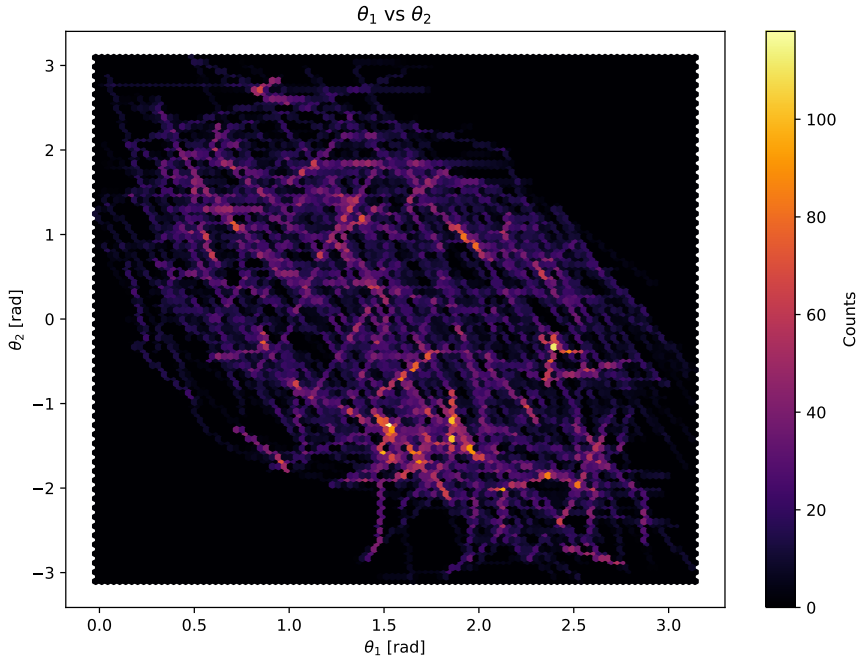


Figure 3.6: Heatmap of the joint positions in the generated dataset (200 samples).

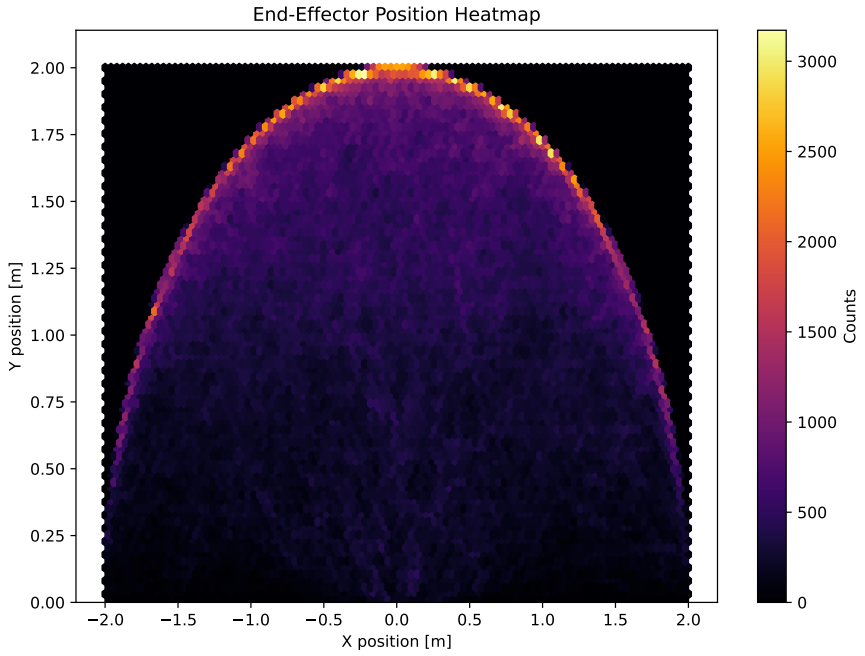


Figure 3.7: Heatmap of the end-effector position in the generated dataset (5000 samples).

by several key factors. First, increasing the sample size improves the resolution of density estimates, especially in regions that appear less frequently but may still be important for learning. Second, a larger dataset offers more statistically reliable data, which is particularly useful when working with learning-based methods. Finally, having better coverage near the edges of the joint space and workspace helps to clearly define the system's operational boundaries, which supports more effective and safer training. Additionally, the computational cost of generating more samples is not a major concern, since this step only needs to

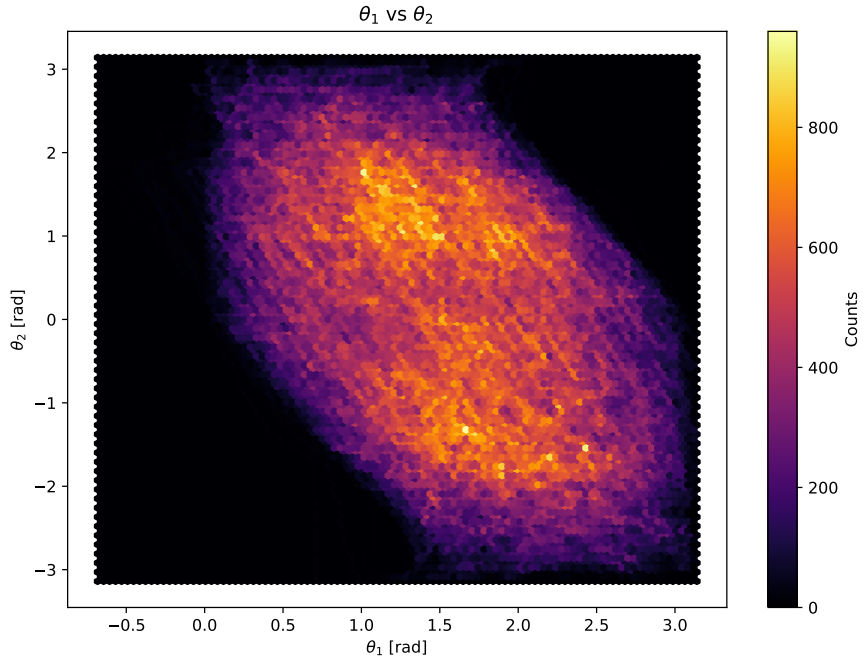


Figure 3.8: Heatmap of the joint positions in the generated dataset (5000 samples).

be done once and does not have to be repeated throughout the development process.

Post-processing using finite difference

Finite differences are commonly used for numerical differentiation. Forward, backward, and central finite difference are basic methods commonly used to approximate derivatives based on discrete data points. The forward method estimates the derivative using the current point and a point ahead (as shown in 3.2), while the backward method uses a point behind; both are first-order accurate with an error of $\mathcal{O}(h)$. The central difference method, which uses points symmetrically around the target and achieves second-order accuracy $\mathcal{O}(h^2)$, is generally more precise. However, in practice, the difference in accuracy may not be significant, especially when working with large datasets where noise and resolution already limit precision [140]. In this work, the central method was discarded due to its higher computational cost, since it requires two evaluations per point, making the forward or backward approaches more suitable for efficient processing.

$$\Delta_h[f](x) = f(x+h) - f(x) \tag{3.2}$$

Post-processing using unconstrained MPC

This section details the formulation of the cost function used in the MPC controller. The cost function defines the desired behavior of the system and is embedded within an optimization problem, which is solved using the Interior Point Optimizer (IPOPT) solver. IPOPT is a numerical solver for large-scale nonlinear optimization problems, particularly those with continuous variables and nonlinear equality and inequality constraints. It implements a primal-dual interior-point method, which transforms the constrained problem into a series of barrier subproblems. These are solved iteratively by applying a Newton-based approach to

the Karush-Kuhn-Tucker (KKT) conditions, making it suitable for problems where gradients are available or can be approximated [141].

One possible approach is to define the cost function based solely on trajectory tracking. In this case, the system prioritizes minimizing the deviation from the desired trajectory, aiming for convergence above all else. While this configuration can give favorable results in terms of trajectory tracking metrics such as RMSE, it often leads to unstable, jerky, or discontinuous velocity, acceleration, and control signal profiles. Such behavior not only increases mechanical wear in real systems, but also complicates the training of the data-driven controller that learns from the trajectories generated by the MPC, while also converging in more unrealistic physical behaviors.

To achieve more realistic behavior and smoother, more consistent velocity profiles across different trajectories, a weight study was conducted to balance tracking accuracy and control effort. Equation 3.3 defines the cost function chosen, where the tracking error is calculated as the squared norm of the difference between the actual and reference or desired states, and the control effort is computed as the squared norm of the control signals u scaled by a weighting matrix R.

$$\text{Cost} = \|x - x_{\text{ref}}\|^2 + \|Ru\|^2 \tag{3.3}$$

The weighting matrix is defined in 3.4, where w_1 is a tunable parameter and w_0 is derived from it. The two weights differ because, due to the system's physical configuration, the first joint experiences a significantly higher load. It must support both links as well as the second joint, and its distance from the center of mass introduces a larger moment. As a result, the control signals for the two joints operate on different scales. The scaling factor of 5 is used to reflect this difference, taking into account that the cost function includes squared terms. This ratio was estimated based on the average magnitude of control signals from a small sample of trajectories, where the difference was observed to be approximately a factor of 5.

$$R = \begin{bmatrix} w_0 & 0 \\ 0 & w_1 \end{bmatrix}, \quad w_0 = \frac{w_1}{\sqrt{5}} \tag{3.4}$$

Figure 3.9 shows histograms of RMSE values obtained from a sample of 50 trajectories using different weight values: 0.005, 0.01, 0.02, and 0.03. As expected, smaller weights favor tracking accuracy, while larger weights prioritize minimizing and stabilizing control signals, sometimes at the expense of tracking performance. This trade-off is reflected in the increasingly left-skewed distributions as the weight decreases. Ideally, both objectives would be optimized simultaneously; however, in practice, a balance must be seek to achieve an effective compromise between tracking performance and control effort.

Table 3.7 presents the numerical statistics for the RMSE across the tested weight values. As shown, tracking performance varies significantly depending on the weight. The mean RMSE remains relatively low (approximately 0.025–0.06) for weights 0.005 and 0.01. However, a noticeable increase in RMSE is observed between weights 0.01 and 0.02, with a difference of

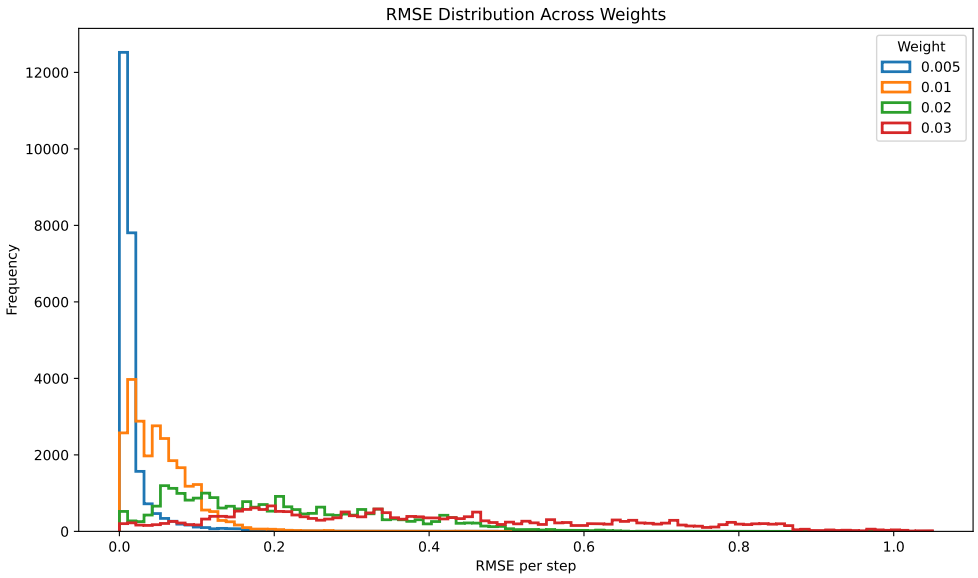


Figure 3.9: Histograms of RMSE of different weights for the cost function of the MPC implementation.

approximately 0.15. Given that low RMSE is desirable, weights of 0.005 and 0.01 present the most favorable results in terms of tracking accuracy.

Weight	Mean RMSE	Standard Deviation RMSE
0.005	0.025074	0.103012
0.01	0.059427	0.100376
0.02	0.210358	0.154032
0.03	0.400317	0.249016

Table 3.7: Mean and standard deviation of RMSE for different weight values in the MPC cost function.

Figure 3.10 shows boxplots of control effort for the same set of weights. A clear and expected trend emerges: as the weight increases, the control effort decreases. The reduction between weights 0.005 and 0.01 is particularly notable, especially in the upper quartile. However, the differences in control effort between weights 0.01, 0.02, and 0.03 are less pronounced.

Figure 3.11 presents a histogram of the control actions for a weight of 0.01. Most trajectories result in control signals within a reasonable range, generally around 100, with only a few outlier cases requiring significantly higher torques.

Figure 3.12 shows the distribution for a weight of 0.02. The overall control effort is slightly lower, with most values staying below 80. As with the previous case, the primary differences are seen in a small number of outliers.

Based on these results, a weight of 0.01 is selected as it offers a balanced compromise. It achieves RMSE values close to those of the lowest-weight configuration (0.005), while sig-

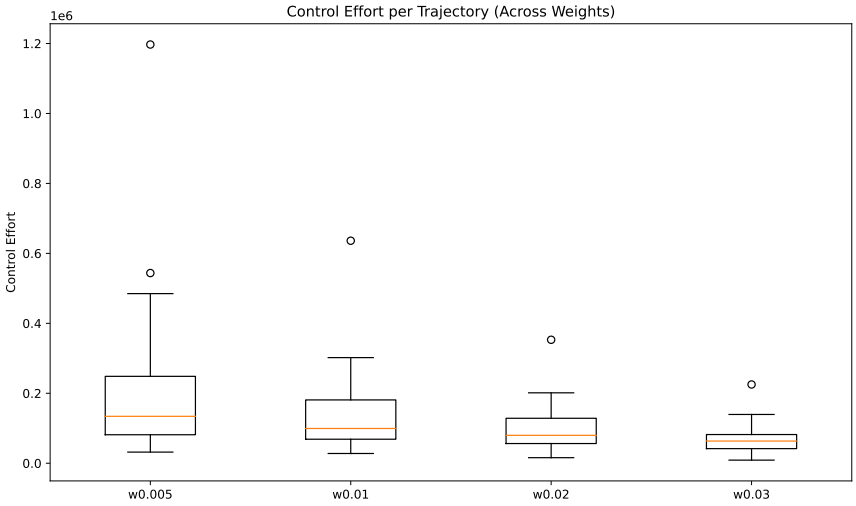


Figure 3.10: Boxplots of the control effort of different weights for the cost function of the MPC implementation.

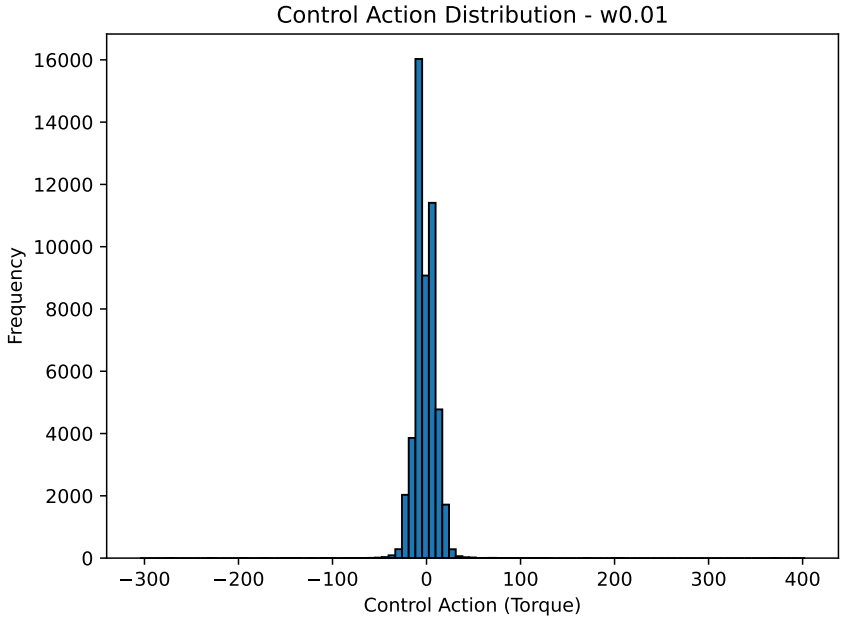


Figure 3.11: Histograms of the control actions for a cost function with weight of 0.01.

nificantly reducing control effort. This weight demonstrates an effective trade-off between trajectory tracking and control smoothness, making it a practical choice for the MPC implementation.

Post-processing using constrained MPC

This section addresses the generation of trajectories that explicitly comply with system constraints. The process began with an analysis of the value ranges in the original dataset to understand their distribution, including the minimum and maximum values and the regions where most trajectories are concentrated. A coverage test was then conducted to examine the statistical behavior of the applied torques. Based on this analysis, two specific datasets were

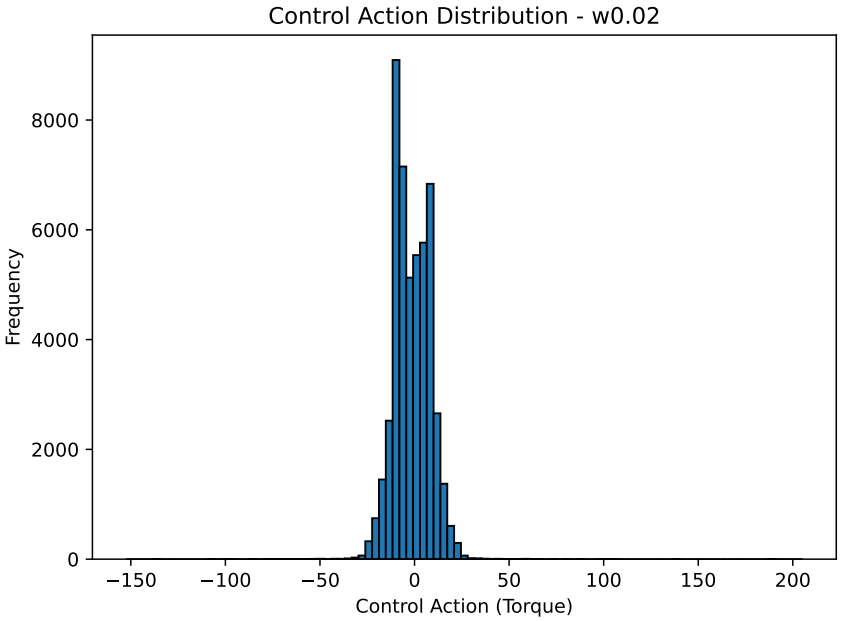


Figure 3.12: Histograms of the control actions for a cost function with weight of 0.02.

generated for further evaluation and for use in MPC training. These datasets were created using constrained torque values, with each dataset corresponding to a different torque limit: one more restrictive than the other.

To better understand the distribution of applied control signals, Figure 3.13 presents a rescaled version of the torque histogram previously shown in Figure 3.12. The plot is divided into two histograms, one for each joint. Joint 1 displays a broader range of torque values, consistent with its higher mechanical load, while joint 2 exhibits a narrower but more irregular distribution.

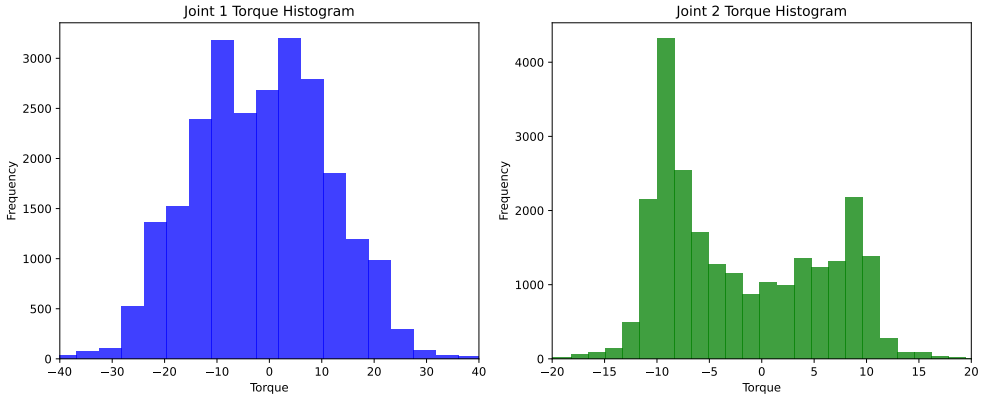


Figure 3.13: Detailed histograms of the control actions for a cost function with weight of 0.02.

Table 3.8 summarizes the key statistics for applied torques across the dataset. These statistics are: mean, standard deviation, minimum, and maximum. These values reflect the behavior of the original, unconstrained dataset where velocities were estimated using forward differences.

Variable	Mean	Std	Min	Max
τ_1 (Nm)	-1.5523	14.3503	-272.6692	370.5749
τ_2 (Nm)	-1.9140	8.3124	-152.4585	93.1549

Table 3.8: Statistical summary of position and velocity variables across the dataset.

Since this work is presented as a proof of concept, the primary goal of introducing constraints is to explore the system's response under different scenarios, whether constraints are inherently present in the dataset or externally enforced. Based on the analysis summarized in table 3.8, a set of practical constraint values for torque are shown, as listed in Table 3.9. These values are chosen based on the key statistics and also on the distributions shown on the corresponding histograms. They are chosen to potentially influence the behavior of the existing dataset while ensuring that it remains feasible. The constraints are used in the subsequent sections to evaluate the system's ability to operate safely and efficiently within realistic limits.

Variable	Min	Max
τ_1 (Nm)	-38 and -20	33 and 18
τ_2 (Nm)	-18 and -12	17 and 12

Table 3.9: Summary of chosen position, velocity, and torque constraints.

3.2.2 Framework for Unconstrained Learning

This section provides a detailed description of the operation and implementation of the unconstrained learning framework. An overview of the system is shown in Figure 3.14. The framework is built upon an adversarial learning structure combined with reinforcement learning. In this setup, the discriminator assigns rewards based on the similarity between generated trajectories and expert demonstrations. The policy is trained to maximize this reward by adapting its behavior to imitate expert trajectories, effectively attempting to deceive the discriminator. Concurrently, the discriminator is updated to improve its ability to distinguish between expert and generated trajectories, creating a competitive dynamic that drives both components to improve over time.

The process begins with a policy rollout, which refers to using the current policy to generate trajectories by applying predicted control actions to the agent (robot) in certain environment. The resulting observations are then used to train both the policy and the discriminator. This forms a loop that is repeated for a number of training iterations, which depends on the current stage of the implementation.

If the system is in the stage where the objective is to tune hyperparameters, meaning that several configurations are being evaluated, it is not efficient to train each configuration for a large number of iterations. However, once the best-performing configurations are identified,

they can be trained for a significantly greater number of iterations to fully exploit their performance. These aspects are discussed in more detail in 3.2.4 and the results are presented in 4.2.

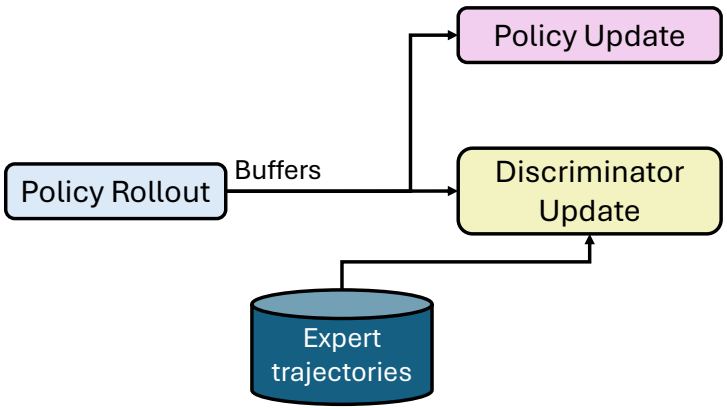


Figure 3.14: High-level block diagram of the unconstrained learning framework.

Figure 3.15 presents a second-level block diagram of the policy rollout process. It begins with an initialization block, followed by multiple environments executing the current policy under different conditions and storing the resulting data in dedicated buffers. This architecture is designed to exploit the advantages of parallel computing, enabling multiple environments to run simultaneously and collect data efficiently. The stored buffers include: log-probabilities, actions, done flags, observations, rewards, and value estimates. All of these components contribute to the subsequent policy update, while the collected observations are also used to update the discriminator.

Based on Section \ref{ch:marco}, the role of each buffer used in the PPO algorithm is discussed as follows. The log-probabilities buffer stores the log-probabilities of the actions taken by the old policy, which are later used to compute the probability ratio r that indicates how much more or less likely the new policy is to take the same action compared to the old one. The actions and observations buffers store the actions executed and the corresponding observations at each timestep. These are necessary for evaluating the probability of the new policy selecting the same actions based on the stored observations, which is essential for calculating the surrogate objective. The dones buffer records whether a given timestep corresponds to the end of an episode. This information is used to reset the internal state when needed and to determine the episode lengths. The rewards buffer stores the immediate rewards received at each step, and these values are used to compute the returns and advantages, which measure how well an action performed given a particular observation. Finally, the values buffer contains the value function predictions from the critic, which are used along with the rewards to compute the advantage estimates and to calculate the loss associated with the value function during the update phase.

Next, Figure 3.16 presents a third-level block diagram that provides a detailed view of the internal process of the policy rollout loop. The process begins with an initialization phase, in which a reference trajectory is randomly selected and the robot is configured to match the initial pose of that trajectory. This initialization occurs only at the beginning of each

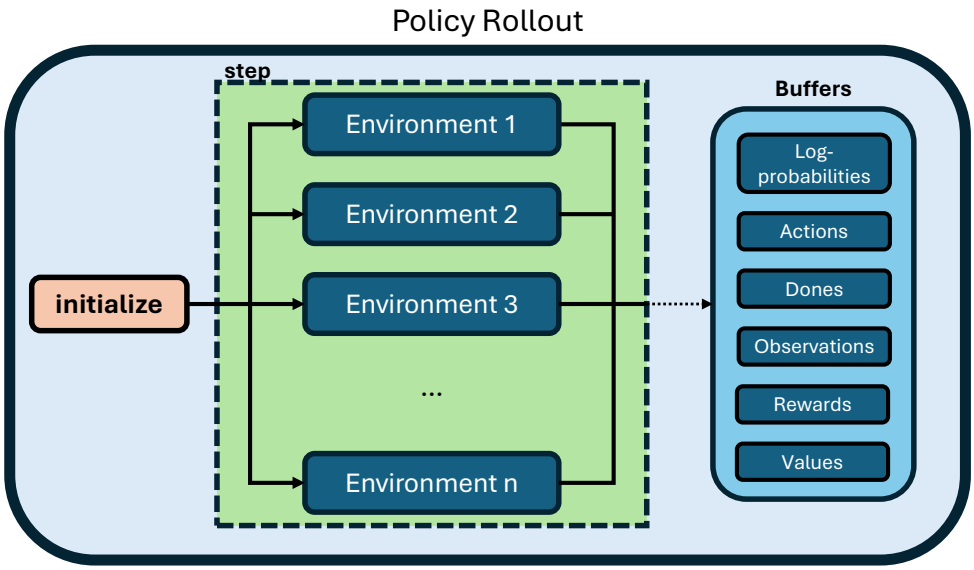


Figure 3.15: Second-level block diagram of the policy rollout block.

episode. A new episode can be triggered under three conditions: (1) at the start of a new rollout iteration, (2) when an episode completes all steps of the assigned trajectory but more observations still need to be collected, or (3) when the robot collides, causing the episode to terminate and restart.

The sub-loop labeled step represents the sequential execution of control signals for each step of the assigned trajectory. The first addition operator in the diagram refers to the reorganization of the observation, which is explained in more detail later using Figure 3.17. Once the observation is structured in the required format, the current policy is executed. This policy outputs the control signals to be applied in the simulation environment in order to compute the next joint state.

In addition to the control signals, the policy also outputs the selected action and its associated log-probability, both of which are stored in the corresponding buffers. Next, a collision checker is executed to determine whether the robot has collided with itself or the ground, which would immediately end the episode.

After obtaining the new state, the addition operator is applied again to structure the new observation. This observation is stored in the buffer and used as input for both the discriminator and the value network. These modules generate the reward and the predicted value, respectively, which are also stored. The cycle is then repeated, this time using the final state from the previous step as the starting state, rather than the one from the initial trajectory configuration.

As described in the explanation of the third-level block diagram, each observation is composed of both the actual states obtained from executing the control actions and the desired future states taken from the assigned trajectory during initialization. Figure 3.17 illustrates the structure of an observation more intuitively. Specifically, an observation consists of M past states, the current state, and N future desired states. The current and past states

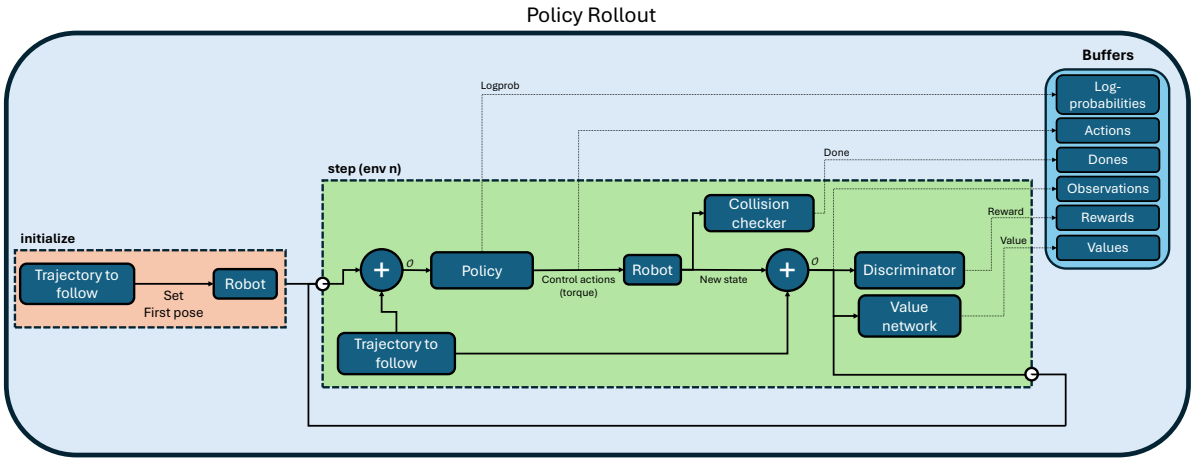


Figure 3.16: Third-level block diagram of the rollout block.

are collected by executing the predicted control actions from the policy, while the future desired states are taken directly from the reference trajectory defined at the beginning of the episode.

The reason for including both actual and desired states is to provide the policy with sufficient information about the current condition of the system and the target condition it should aim to reach. This allows the policy to generate control actions that reduce the discrepancy between the current state and the desired future states. Simultaneously, the discriminator receives both the executed behavior and the expected behavior, enabling it to detect inconsistencies between generated and expert trajectories. Expert trajectories, by design, show smooth and coherent transitions, which the discriminator can use as a basis for comparison.

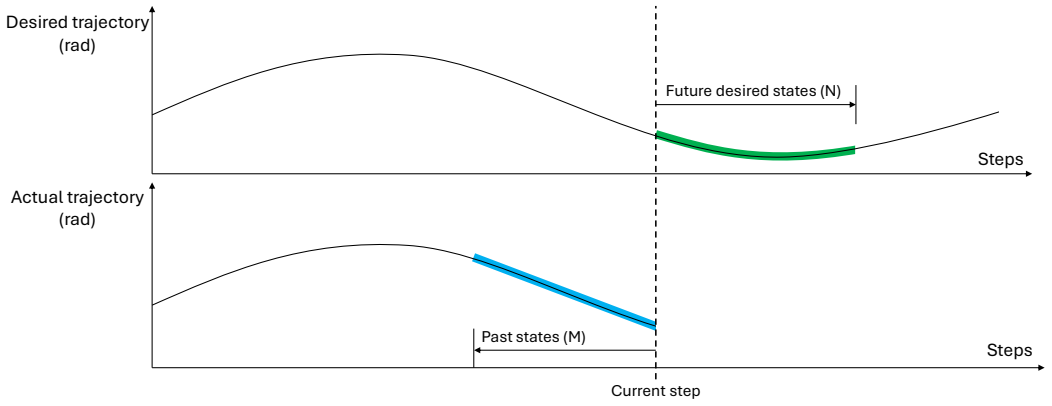


Figure 3.17: Representation of the observation space.

The action space correspond to torques. In the case of this proof of concept with 2 DoF, the shape of the output is 2 different torque values that will be input as the torque values for both joints such as (τ_1, τ_2).

The action space corresponds to joint torques. In this proof-of-concept implementation with two degrees of freedom (2 DoF), the policy outputs a vector of two torque values, one for

each joint. The resulting control action is represented as (τ_1, τ_2), where (τ_1 and τ_2 denote the torques applied to joint 1 and joint 2, respectively.

A more concise and generic implementation is shown in the algorithmic description 1.

Algorithm 1 PPO with Structured Observations and Discriminator-Based Reward

Initialize:

- Policy π_θ and value network V_θ (GRU-based)
- Discriminator network D_ϕ
- Buffers: $\mathcal{O}$ (observations), $\mathcal{A}$ (actions), $\mathcal{L}$ (log-probabilities), $\mathcal{R}$ (rewards), $\mathcal{V}$ (values), $\mathcal{D}$ (dones)
- Simulation environment

Load expert trajectories as structured observation windows into buffer $\mathcal{D}_{\text{expert}}$
for each PPO iteration **do**
  for $t = 1$ to T **do**
    Sample $a_t, \log \pi(a_t|\mathcal{O}_{t-N:t+M}), v_t \leftarrow \pi_\theta(\mathcal{O}_{t-N:t+M})$
    Execute action a_t in the environment.
    Store $\mathcal{O}_{t-N:t+M}, a_t, \log \pi(a_t), v_t, r_t, done_t$ in respective buffers
    Store $\mathcal{O}_{t-N:t+M}$ in generated buffer $\mathcal{D}_{\text{gen}}$
  end for
  for each discriminator training step **do**
    Sample mini-batch of structured observations from $\mathcal{D}_{\text{expert}}$ and $\mathcal{D}_{\text{gen}}$
    Update D_ϕ using cross-entropy discriminator loss
  end for
  Compute advantages $\hat{A}_t$ using GAE over $\mathcal{R}$ and $\mathcal{V}$
  Compute returns $R_t = \hat{A}_t + V_t$
  for each PPO epoch **do**
    Shuffle and split the rollout data into minibatches
    for each minibatch **do**
      Recompute $\log \pi_\theta(a|\mathcal{O})$ and $V_\theta(\mathcal{O})$
      Compute importance ratio r_t
      Compute PPO clipped surrogate loss and value loss
      Update policy and value networks via gradient descent
    end for
  end for
end for

To implement said algorithm, an existing software framework was chosen as a baseline. The frameworks are evaluated based on how well they balance efficiency, flexibility, and compatibility with simulation environments. Stable Baselines3, CleanRL, and RLlib are among the most widely used libraries, each offering unique strengths [142] [143] [144]. Stable Baselines3 emphasizes modularity and ease of use, providing well-tested implementations of popular algorithms like PPO and A2C, along with utilities for logging, normalization, and

recurrent policies via the sb3-contrib package. CleanRL, in contrast, focuses on simplicity and transparency, offering single-file, easy-to-read implementations ideal for research and education, though it lacks some advanced features such as recurrent support and multi-environment training.

RLlib stands out for its scalability, offering distributed training capabilities and a unified interface for a variety of algorithms within the Ray ecosystem. It is suitable for large-scale, industrial applications but can be complex to set up and overkill for smaller projects. Ultimately, selecting an RL library for robotic control depends on specific needs, such as the importance of recurrent architectures, ease of customization, simulation compatibility, and training scalability. In the specific case of this work in which a customized algorithm is implemented, resulting in several debugging situations, where complete control of the code is crucial, the best option is cleanrl.

Architectures

This section describes the neural network architectures used for both the agent and the discriminator ensemble. Both components are implemented using the PyTorch deep learning framework [145], which provides the flexibility and performance necessary for training and deploying reinforcement learning models.

The agent follows an actor-critic architecture with a shared temporal encoding stage. As shown in Figure 3.18, the input to the model is a sequence of observed states S_t, composed of N past states, the current state, and M future desired states. Each state is a 4-dimensional vector, resulting in an input of shape (4×(N+M+1)). This sequence is processed by a GRU layer, which acts as a temporal feature extractor. The final hidden state of the GRU serves as a compact representation of the sequence and is shared between both the actor and critic networks.

The actor network, shown on the left side, comprises two fully connected (FC) layers with 128 and 64 units, respectively, each followed by ReLU activations. The final output layer generates the mean vector μ of a Gaussian distribution over the two-dimensional action space. In contrast, the standard deviation is not computed directly from the network. Instead, it is modeled as a separate, learnable parameter vector representing $\log \sigma$, which is shared across both the batch and action dimensions during policy sampling. This design ensures a positive standard deviation, as required by the distribution, and enhances numerical stability. Optimizing the logarithm of the standard deviation is also preferred because it avoids issues with extreme floating-point values and typically results in a smoother learning process compared to directly learning the variance or standard deviation.

The critic network, shown on the right side of Figure 3.18, mirrors the structure of the actor with two FC layers of sizes 128 and 64, each followed by ReLU activations. It outputs a single scalar value V, which estimates the expected return from the given input sequence.

All linear layers in both the actor and critic branches use orthogonal initialization to enhance training stability. The clear modular structure ensures efficient representation sharing while

enabling separate optimization of the policy and value functions.

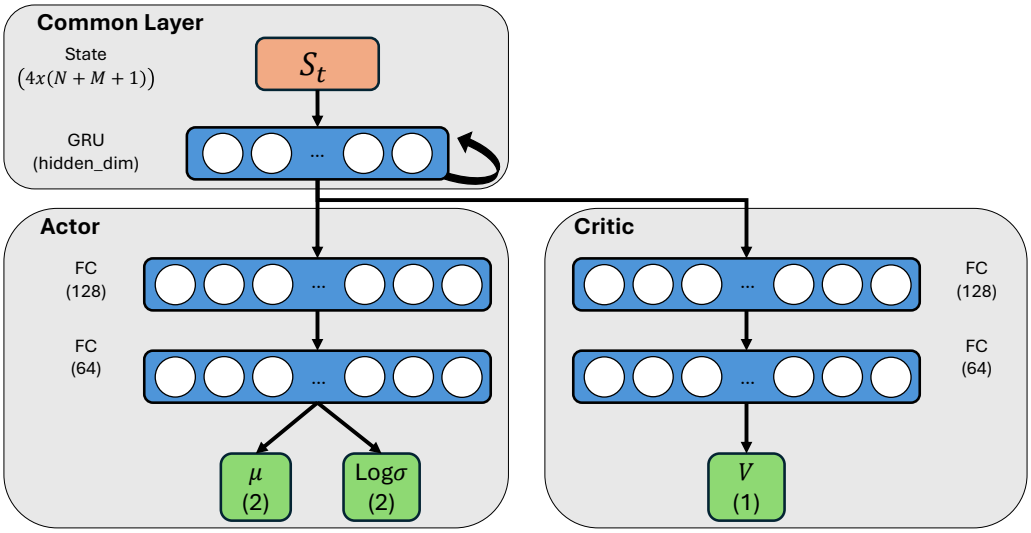


Figure 3.18: Architecture of the PPO agent, showing the shared GRU encoder and separate actor-critic branches

The reward signal in this framework is learned using an ensemble of discriminator networks shown in 3.19. The ensemble is composed of multiple independent output heads that share a common GRU encoder and feedforward backbone. This structure is motivated by ensemble learning techniques, which reduce variance and improve robustness when learning from noisy data. Each head produces a scalar output indicating whether the input trajectory window resembles expert or policy behavior. The shared GRU processes the input sequence identically to the agent's encoder, and the final hidden state is passed through a shared two-layer MLP, which then feeds into the independent linear heads.

Training the discriminator ensemble follows a Wasserstein GAN with Gradient Penalty (WGAN-GP) formulation, in which each head is trained to assign high scores to expert trajectories and low scores to generated ones. A gradient penalty term is applied to interpolated samples between expert and generated sequences to enforce Lipschitz continuity. The ensemble average is used during reward shaping, and the gradient penalty coefficient (λ_{GP}) is treated as a tunable hyperparameter.

3.2.3 Framework for Constrained Learning

This framework directly builds on the previous section by using the same algorithm with a single key modification: the integration of constraint handling. Constraints may be introduced through explicit penalty terms, inferred implicitly from the training dataset, or a combination of both. The objectives of this section are twofold: first, to evaluate the system's ability to infer constraints implicitly followed by the expert without explicit specification; and second, to assess how effectively the system can explicitly enforce such constraints once inferred.

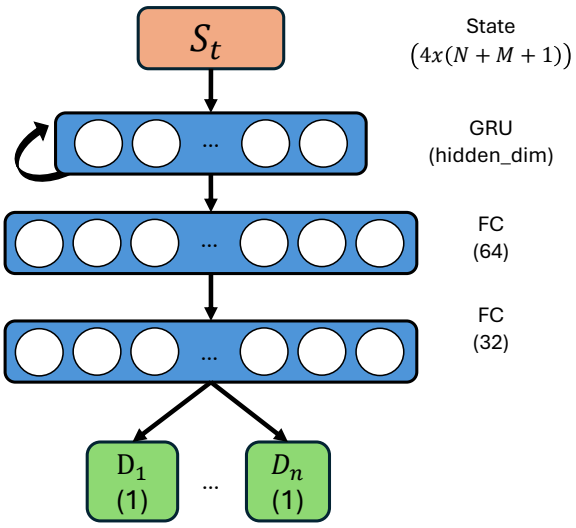


Figure 3.19: Architecture of the discriminator ensemble, featuring a shared GRU and MLP, followed by multiple independent output heads.

To analyze the effectiveness of each approach to constraint handling, four experimental configurations are proposed:

- Learn solely from a constrained dataset.
- Learn from a constrained dataset and additionally assign a penalty to torque limit violations.
- Apply clipping to control actions that exceed predefined limits.
- Combine all previous methods: learn from a constrained dataset, apply clipping, and penalize violations.

Penalty Funtion

The penalty function is a mechanism introduced to discourage the policy from producing control actions that violate torque limits. In the context of reinforcement learning, especially when using learned reward functions, the agent might exploit areas of the action space that obtain high rewards but are physically unsafe or unrealistic. The penalty function addresses this issue by assigning a negative cost to actions that exceed safe operational boundaries, thus guiding the learning process toward feasible and safe behaviors. By integrating this penalization into the reward signal, the agent is incentivized not only to imitate expert behavior but also to remain within safe actuation ranges, improving both safety and transferability to real-world systems. A detailed definition of this function is provided below.

First, the action output is normalized to the range [-1,1] to facilitate its use in penalty computation. It is then clipped, allowing for aggressive penalization of control actions that

significantly exceed torque limits, while maintaining reasonable sensitivity. This avoids disregarding the primary objective of trajectory tracking. Penalties are applied only when actions exceed the range $[u_{min} + \text{safety\_factor}, u_{max} - \text{safety\_factor}]$, ensuring that actions violating a stricter margin (but still within limits) are also penalized.

A multiplicative scaling factor is applied to align the penalty with the reward magnitude produced by the discriminator. This factor was empirically determined based on experiments conducted using the unconstrained configuration. A quick calculation reveals that the worst-case penalty in this framework is -0.225, which remains within the typical reward scale observed in previous iterations.

In future iterations of this framework, the primary constraint of interest will be the force at the tool center point (TCP). Since the application involves cobots, monitoring and limiting the contact force at the end-effector is critical for ensuring safe interaction with humans. Controlling this force allows the system to mitigate potentially harmful impacts and comply with safety standards in human-robot collaboration.

However, accurately estimating and constraining TCP forces introduces significant complexity, as it requires dynamic modeling, force propagation through the kinematic chain, and potentially external sensing or robust estimation techniques that are difficult to validate. Due to these challenges, the current implementation focuses on joint torque constraints as a preliminary step. Joint torques are directly accessible from the simulation and low-level control interface and provide a viable method for regulating excessive actuation, which indirectly contributes to limiting unsafe behavior at the end-effector.

This simplification allows us to first validate the system's ability to handle constraints effectively in a controlled and computationally manageable setting before progressing to more sophisticated constraints such as end-effector force limits.

3.2.4 Hyperparameter Tuning

As in most artificial intelligence implementations, hyperparameter tuning is a critical but often heuristic-driven process that significantly impacts the performance and stability of the final model [146]. Traditionally, this process involves manual experimentation or grid search, both of which are time-consuming and computationally expensive. In this work, to ensure efficiency and reproducibility, an automated hyperparameter optimization strategy is adopted.

The hyperparameters used in this framework are detailed in Table 3.10, which includes the name of each hyperparameter, a brief description, the search method, and the range or discrete values considered for tuning.

Automated Machine Learning (AutoML) refers to a collection of tools and methods designed to automate the end-to-end process of applying machine learning, including hyperparameter tuning, model selection, and sometimes feature engineering [147]. In the context of RL, hyperparameter tuning plays a vital role, as performance can vary widely depending on the

combination of hyperparameters, and the implementations tend to have a large amount.

AutoML frameworks typically use optimization techniques such as Bayesian optimization, bandit algorithms, or evolutionary strategies to search the hyperparameter space more intelligently than exhaustive or random search [148]. These methods aim to find near-optimal configurations with fewer training runs, which is crucial given the high computational cost of training the proposed algorithm.

In this work, the policy training pipeline integrates PPO with an ensemble of discriminator networks to generate a reward signal. The combined architecture involves a large number of interdependent hyperparameters across policy learning, reward modeling, and environment configuration. Manual tuning would be highly inefficient and error-prone in this setting.

By using AutoML, the search is guided toward promising regions of the hyperparameter space, improving sample efficiency and model performance while significantly reducing the number of training runs required. Furthermore, AutoML allows for more principled comparisons across configurations.

Several AutoML libraries were evaluated for their potential to be integrated into the existing reinforcement learning framework. Some of the considered libraries were: Optuna, Hyperopt, Scikit-Optimize, Nevergrad and FLAML [149] [150] [151] [152] [153].

After comparing usability, integration capabilities, documentation, FLAML (Fast Lightweight AutoML) was selected for this work. FLAML was chosen due to its low-overhead design, ease of integration with the RL training pipeline, and strong support for budget-constrained optimization. One of its key advantages is that it minimizes tuning overhead by using cost-effective search strategies like low-cost initial sampling and adaptive resource allocation. Additionally, FLAML supports integration with Ray Tune [154], which provides scalable, distributed hyperparameter search across CPUs and GPUs, fitting the requirements of the training setup used in this work. The decision to use FLAML was further supported by the recommendation of the thesis client, who had prior experience with the library and confirmed its effectiveness in reinforcement learning workflows.

3.3 Economic Feasibility Study

It is important to clarify that this project is being developed within a research laboratory, and for that reason, its economic analysis is not as straightforward as it would be in an industrial context. The implementation and innovation presented here do not lead to an immediate commercial solution. As a result, quantifying its economic value is more complex, since potential improvements in productivity or safety must go through additional stages before they can be compared to current state-of-the-art industrial solutions.

This work can be considered an initial step in exploring a novel concept which, to the best of the author's knowledge, has not yet been tested. Based on its theoretical and practical potential as identified in the literature, this project proposes and implements a proof of concept.

The economic feasibility study will be structured starting first with a small-scale analysis, followed by a large-scale projection. The small-scale analysis is an estimate of the research-level investment required to develop the project in a laboratory setting, followed by the potential funding that could be obtained through research grants. Meanwhile, the large-scale projection is an estimate of the investment needed for full industrial deployment, and an analysis of the possible long-term economic returns.

3.3.1 Estimated Costs Incurred During the Proof of Concept

Although the proof of concept was conducted within a university research environment, it involved several direct and indirect expenditures. This section outlines the main cost components associated with the development of the project, including travel, accommodation, computing resources, and access to digital infrastructure. While not all of these incurred out-of-pocket expenses, they reflect the economic value of the resources used.

Administrative and Living Expenses: Conducting the project at TUM involved several location-specific expenses related to international mobility, residency, and administrative requirements. These include international travel from Costa Rica to Munich, estimated at approximately 700 euros for a one-way flight. During the six-month research stay, accommodation in a shared one-bedroom apartment amounted to roughly 4,500 euros. In accordance with German visa regulations, it was mandatory to obtain health insurance, which costs approximately 150 euros per month, totaling 900 euros for the duration of the stay. Additional administrative costs included a visa application fee of approximately 90 euros, as well as TUM enrollment fees for both the summer and winter semesters, totaling around 160 euros. These costs represent essential non-research expenditures directly associated with the ability to conduct the project on-site at TUM.

Use of Personal Equipment: The development and testing phases were primarily carried out on a personal laptop that is over five years old. While no direct purchase was made, the equipment's age and performance constraints justify an estimated depreciation and maintenance cost of 100 euros over the duration of the project.

Access to Academic Databases: Access to academic literature was facilitated through the TUM library services, which provide institutional subscriptions to databases such as JSTOR and EBSCO. While this access was granted without individual charge, its commercial value is significant. A proportional estimate of usage for this project is valued at 500 euros.

Access to High-Performance Computing (HPC) Resources: HPC clusters provided by the host laboratory were used for model training and simulation. Although no fee was charged to the researcher, the fair-use value of such access is estimated at 500 euros, based on standard academic HPC service rates.

Table 3.11 summarizes the estimated value of resources used during the project.

Table 3.11: Estimated Costs Incurred During the Proof of Concept

Category	Estimated Cost (€)
Administrative and Living Expenses	6350
Use of Personal Equipment	100
Academic Database Access	500
HPC Access	500
Total Estimated Cost	7200

3.3.2 Available Funding and Investment Opportunities

This section discusses potential funding opportunities that the laboratory could pursue as a result of developing this project. Specifically, it refers to research grants that align with the technical focus and objectives of the work. It is important to note that simply completing the project may not be sufficient to secure such funding. Rather, the outcomes of this project could serve as the foundation for a more targeted and well-defined proposal, highlighting specific research directions and justifying their relevance based on the preliminary results.

Horizon Europe

Horizon Europe, the European Union's flagship research and innovation programme, offers substantial funding opportunities for collaborative R&D projects [155]. Notably, Cluster 4: Digital, Industry and Space includes calls focusing on robotics, artificial intelligence (AI), and advanced manufacturing [156]. Recent calls have allocated a budget of around 20 million to topics that aim to provide industry with more autonomous, easy to operate, and trustworthy AI and robotics technologies [157].

Consortia comprising universities, industries, and other European partners are eligible to apply for Innovation Action grants supporting multi-year projects. This project aligns well with themes like enhancing human-robot collaboration, advancing industrial automation, and integrating AI into manufacturing processes. Participation in Horizon Europe projects also facilitates cross-border collaboration and increased visibility [155].

European Research Council (ERC)

The ERC offers prestigious grants to individual principal investigators pursuing ambitious scientific research. Given the project's foundation in fundamental research, an ERC grant could be an appropriate funding mechanism [158].

ERC grants typically provide funding up to 2.5 million euros over a five-year period, sufficient to support a small research team [158]. The emphasis is on scientific excellence; for instance, an ERC grant could enable a TUM professor to dive into the theoretical aspects of AI-based

control and safe learning in robotics, directly benefiting the project's applied feasibility.

German Research Foundation (DFG)

The DFG (Deutsche Forschungsgemeinschaft) serves as Germany's primary agency for funding fundamental research [159]. It offers individual research grants, typically through 2 to 3 years, which could support a PhD student or postdoctoral researcher [159]. DFG also administers coordinated programmes such as Priority Programmes (SPP) and Collaborative Research Centres (SFB) [160]. [161]

For example, DFG has funded clusters focusing on topics like "Soft Material Robotic Systems" (SPP 2100), emphasizing the development of adaptable robotic systems using compliant materials [162]. Applying for a DFG grant would highlight the project's scientific innovation, such as new control theories or learning algorithms, rather than immediate industrial deployment.

Federal Ministry of Education and Research (BMBF)

BMBF in Germany supports applied research and development through thematic programmes and calls. Recently, BMBF launched the "Aktionsplan Robotikforschung" (Robotics Research Action Plan) to strategically enhance AI-based robotics innovations and facilitate the practical application of intelligent robotics. Under this initiative, specific funding calls are anticipated in domains such as manufacturing, healthcare, and environmental robotics [163].

BMBF grants often involve consortia comprising academia and industry, focusing on translating research into real-world impact. For instance, BMBF has funded programmes in human-robot collaboration, robot learning, and AI in industrial production. These projects typically receive funding in the range of a few million euros over three-year durations, shared among partners [164].

Additionally, initiatives like the Robotics Institute Germany (RIG), are backed by BMBF to strengthen national collaboration [165]. Participation in RIG or related networks can open further opportunities for funding and support. BMBF funding could be accessed either through direct project grants or by aligning the project with national innovation agendas, such as ÄI in manufacturingör "Future of Work with robots" [165].

Other Funding Sources

Apart from the big four options detailed above, there are other notable funding avenues.

Bavarian State Programmes: Bavaria offers its own innovation funds, such as the "Bayerische Forschungsstiftung" and high-tech agendas supporting AI and robotics. The Bavarian Research Foundation has funded projects like FORinFPRO with approximately 2 million euros for the development of AI-based and self-adapting manufacturing processes. Being located

in Munich, the project could potentially secure state co-funding or industrial sponsorship from the region's prominent robotics and automation industry [166].

EU Horizon 2020/Horizon Europe Cascade Funding: Smaller grants via EU networks offer funding in the range of 50,000–200,000 euros to experiment with new robotics technologies. This option is particularly suitable for a quick prototype integration [167].

Industry Partnerships: Collaborations with companies such as robot manufacturers (e.g., KUKA, ABB) or automation firms can provide sponsorship for university research, especially if it aligns with their strategic roadmaps. For example, the Marie Skłodowska-Curie Industrial Doctorate programme facilitates partnerships between academia and industry, offering funding for doctoral candidates engaged in industry-relevant research projects [168].

3.3.3 Projected Costs for Industrial-Scale Implementation

1. High-Performance Computing (HPC) for Training

Training a data-driven controller demands substantial computational resources. Utilizing GPU-accelerated HPC clusters, the project is expected to consume between hundreds to a few thousand GPU-hours for neural network training and extensive simulations.

As discussed in 3.1.4, commercial cloud platforms offer NVIDIA A100 GPUs priced at approximately $3.35 or $1.35 (preemptible) per hour. Consequently, 1,000 GPU-hours would equate to an estimated $1350 - $3350 (1190-2945 euros). Additionally, CPU hours for simulations, especially when employing physics engines, should be considered. Accumulating a few thousand core-hours might incur costs in the range of a few hundred euros.

Data storage expenses for large simulation datasets are also pertinent and should be factored into the budget.

2. Simulation Environment and Data Generation

Developing simulation scenarios and generating training data is a significant task. This process encompasses setting up simulations (robot models, environment models, scenario scripting) and allocating compute time to execute them.

Advanced simulators like NVIDIA Isaac Sim or Gazebo. While many are open-source like Gazebo [133], some, such as RoboDK, have licensing fees; RoboDK's professional license is priced at $3995 [169]. Or NVIDIA Isaac Sim is free to use but in case of wanting to distribute a developed application, then a license is required which costs from $4500 for enterprise usage [170].

Computationally, generating 10,000 trajectories of simulated robot motions might necessitate several days of runtime on a robust workstation or server. For example, running simulations on an 8-core machine for 100 hours results in 800 CPU-hours. Given that CPU costs are considerably lower than GPU costs, they are relatively negligible in comparison.

Overall, simulation is a cost-effective method for producing training data relative to physical trials, which may involve wear-and-tear or require increased human supervision. It is

reasonable to budget 500 to 3000 euros for specialized simulation tools, computing overhead beyond HPC, and data storage management.

3. Hardware for Deployment (Prototyping & Future Deployment)

While the project currently focuses on simulations, planning for real-world deployment necessitates budgeting for prototype hardware.

At a minimum, a collaborative robot is required for controller testing. The applied mechanics chair at TUM posseses a GoFa 15000 cobot from ABB. Howeverm if it was to be considered as an adquisition in order to test on it, acquiring it would cost around 34,000 euros [171].

New costs pertain to computing platforms for real-time controller execution. The chosen option, discussed in 3.1.4, is Coral Edge TPU, priced at approximately $125 (110 euros). This would be required both for the development and as part of the serial production of the robot, in case of full adoption of the controller.

4. Personnel and Research Labor

Personnel expenses constitute a significant portion of research project budgets, often accounting for approximately 70–80% of total costs . In Germany, PhD students are typically employed under the collective agreement for public service employees of the federal states (TV-L), specifically at the E13 pay grade. The exact salary depends on factors such as experience level (Stufe) and the percentage of a full-time position [172] [173]

For a full-time (100%) position at E13, the gross monthly salary ranges from 4,253 euros to 5,951 euros, depending on the experience level . Assuming an entry-level PhD student at Stufe 1, the annual gross salary would be approximately 51,036 euros. Over a typical three-year PhD duration, this amounts to 153,108 euros per student. Therefore, employing two PhD students over three years would result in personnel costs ranging from approximately 306,000 euros to 428,000 euros, depending on their experience levels and salary progression [173] [174].

It's important to note that these figures represent gross salaries. Net salaries are subject to deductions for income tax, health insurance, pension contributions, and other social security contributions. For instance, a single PhD student at 100% E13, Stufe 1, would have a net monthly salary of approximately 2,935 euros [175].

In addition to PhD salaries, budgets should account for other personnel-related expenses, such as:

- Student Research Assistants: Part-time assistants who support research activities.
- Postdoctoral Researchers: Postdocs may be employed at higher than E13 pay grades.
- Faculty Supervision Time: A portion of faculty members' time dedicated to supervising PhD students and overseeing research activities.
- Travel and Dissemination: Costs associated with attending conferences, workshops, and other dissemination activities, which are often required by funding agencies.

5. Miscellaneous

Miscellaneous costs encompass publishing fees (e.g., open-access journal charges), patent application expenses if pursuing intellectual property protection, and contingency funds for equipment repairs or additional data requirements. These costs are relatively minor compared to the categories above but should be included at approximately 5–15% of the direct costs.

Table 3.12 provides a summary of all implementation-related cost components discussed. The robotic manipulator is marked with an asterisk (*) to indicate that its inclusion depends on the scope of the analysis. It is excluded from the lower-bound cost estimate but included in the upper-bound estimate to reflect a scenario where hardware acquisition is required.

3.3.4 Economic Appeal of Collaborative Robots in Industry

The cobot industry is experiencing rapid growth, driven by increasing demand for flexible automation across sectors. The global cobot market was estimated at roughly $2.1 billion in 2024 and is projected to grow at an annual rate of about 30–35%, reaching $11–12 billion by 2030 [176] [177] .

This high growth rate reflects strong adoption of cobots in manufacturing, healthcare, automotive, logistics, and other industries seeking cost-effective automation [176].

Europe represents a significant portion of this market, around 29.6% of global cobot revenues in 2024. European cobot sales were around $635 million in 2024 and are expected to climb to approximately $3.3 billion by 2030 (around 30% compound annual growth rate (CAGR)) [178].

Germany in particular is a leading adopter in Europe, projected to exhibit the highest regional growth through 2030 [178] . Key application areas for cobots include assembly, pick-and-place, quality inspection, and collaborative operations in close proximity to humans. The appeal of cobots lies in their higher return on investment (ROI) compared to traditional industrial robots. Robots have easier programming, and built-in safety features that allow deployment without extensive safety fencing [176] .

This makes cobots especially attractive to small and medium-sized enterprises and new use-cases (from factory floors to healthcare and services) that were previously infeasible for automation. Notably, while cobots still account for a minority (∼10%) of total industrial robot installations as of 2023, their share is quickly rising [179].

Economic Potential of AI-Based Control for Cobots

Reduced Engineering and Modelling Costs

Designing conventional robot controllers often demands extensive upfront modeling and manual tuning. Engineers must derive accurate physical models and iteratively adjust parameters (e.g. PID gains), which consumes significant time and expertise. For example, achieving a

well-tuned manipulator controller usually involves system identification of the robot's dynamics, a tedious procedure in terms of experimental trials and data processing time [180] . In multi-robot industrial cells, the programming and planning phase alone can account for over 40% of the total deployment cost [181]. Such modeling and engineering overhead translates to many person-hours of skilled labor and prolonged development cycles. This evidence supports the claim that AI-based or learning controllers, which can learn control policies directly from data, have potential to reduce this overhead. By forgoing explicit physics models and auto-tuning control parameters, learning-driven approaches can save considerable modeling effort and engineer time [180]. In short, traditional control design is resource-intensive, and reducing reliance on manual modeling (as AI controllers do) can cut both the time and cost needed to commission robotic systems.

Programming and Integration Timelines for Cobots

Deploying a collaborative robot (cobot) for a new task using conventional programming methods typically takes several days or even weeks, depending on task complexity. Industry case studies report a wide range of integration times. For simple applications, cobots can be set up and programmed within a few hours to a couple of days using techniques like teach-by-demonstration or logic-based programming [182]. However, more complex deployments take significantly longer. For example, one manufacturer reported that integrating a UR10 cobot into a CNC machine-tending line required approximately three weeks [183]. In advanced multi-robot systems, the timeline can extend to "weeks to months" due to the need for extensive programming, safety logic, and validation—where even small changes require reprogramming and testing [181].

These traditional integration workflows contrast sharply with modern approaches. Some manufacturers now report that standardized interfaces and improved software tools have reduced integration times from several days to just a few hours [184].

The advantage of learning-based methods is that cobots can be trained to perform tasks using data, rather than manually coded logic. This greatly reduces downtime and allows for faster re-tasking. Multiple control policies can be trained for different tasks, turning the cobot into a plug-and-play system that requires minimal or no programming. Since conventional programming can be tedious and time-consuming, eliminating the need to manually write task-specific logic results in much faster deployment.

Adoption Barriers in Industry and the Role of Adaptability

Although cobots offer many advantages, adoption remains limited in several industries due to challenges like integration complexity, limited flexibility, and safety concerns. SMEs, a key target for cobots, often lack the technical expertise for programming and deployment. A European study identified three main barriers in manufacturing SMEs: complex programming and safety requirements, limited integration knowledge, and cultural resistance such as fear of accidents or job loss [185].

Surveys confirm these issues: 42% of SMEs cite high upfront costs, while 37% struggle with integration due to the need for specialized programming [186]. About one-third are unsure

if automation will deliver sufficient ROI [187]. In healthcare, 44% of providers report cost as a barrier, and 43% face integration issues with existing systems [186]. Similar hesitations exist in agriculture and food processing, where tasks are unstructured and robotics expertise is scarce [188].

AI-based controllers could reduce these barriers by enabling cobots to learn and adapt to new tasks, turning them into plug-and-play systems. Over 60% of modern cobots now support rapid deployment without deep programming [186], and many integrate AI features like machine vision and adaptive control [186]. This flexibility allows robots to be re-used across tasks and lowers the need for custom integration.

Greater autonomy and improved safety features also reduce user concerns. As analysts note, easier deployment and flexibility "make robotic automation a more viable and cost-effective solution," particularly for organizations with limited technical resources [181]. Overall, AI-based control can help expand cobot adoption in sectors like healthcare, agriculture, and food processing.

3.3.5 Discussion

Proof of Concept

The implementation of the proof of concept within a research setting involves a modest yet meaningful allocation of resources. With an estimated total value of approximately 7,200 euros, the project leveraged a mix of direct expenditures and in-kind institutional support, including access to HPC clusters, academic databases, and research infrastructure. Although these costs are relatively low by industrial standards, they reflect the essential baseline investment required to evaluate the viability of a novel AI-based control concept.

The core value of this early-stage work lies not in immediate return on investment, but in its potential to unlock significant funding opportunities for future development. By demonstrating preliminary results and technical feasibility, the project positions itself to compete for European research grants. Opportunities such as Horizon Europe, ERC Advanced Grants, and national programs offer substantial financial support; ranging from 50,000 euros cascade grants to multi-million consortia funding. These grants could support the continuation and expansion of the research through the recruitment of PhD students, acquisition of new hardware, and broader experimentation.

Thus, the modest upfront investment in the proof of concept could translate into substantial financial leverage if used strategically as a foundation for high-quality proposals. The opportunity to secure funding not only validates the research but also ensures the project's continuity and potential to mature into applied industrial solutions.

Industrial-Scale Implementation

Scaling the project from laboratory proof of concept to full industrial deployment involves considerably higher investment. As outlined in the feasibility study, projected costs range from approximately 323,000 to 538,000 euros, depending on how conservative the prediction is. The majority of this cost is attributed to research labor, which alone accounts for more than 300,000 euros across three years.

Despite the substantial costs, the potential return on investment is promising, especially when considering the explosive growth of the collaborative robot (cobot) market. With a projected (CAGR) of 30–35%, the global cobot industry is expected to expand from $2.1 billion in 2024 to $11–12 billion by 2030. Germany, and Europe more broadly, are at the forefront of this trend, with significant participation from sectors such as manufacturing, healthcare, and logistics. Within this context, a product that reduces engineering time, accelerates integration, and enhances flexibility offers a clear competitive advantage.

Traditional robot controllers require substantial time and expertise for physical modeling, parameter tuning, and integration, often making deployment slow and costly. AI-based controllers, on the other hand, can learn control behaviors directly from data, drastically reducing both setup time and technical complexity. This is especially beneficial for sectors with low automation adoption, such as healthcare, agriculture, and SMEs, which often lack the specialized staff needed for traditional integration. As modern cobots increasingly shift toward plug and play functionality, AI-based control stands out as a critical enabler, removing barriers such as programming difficulty, high engineering costs, and limited system adaptability.

Considering the projected deployment costs, the economic potential becomes compelling when targeting a market that is growing at over 30 percent annually. If proven effective, the proposed controller could be scaled across different platforms or licensed to industry partners, unlocking long-term economic value. However, implementing a system at this scale would require a comprehensive market study, involving expert consulting and the use of current industry data and trends. Such a study would be essential to validate commercial viability and guide strategic decisions before full-scale investment.

Table 3.10: Search space for hyperparameter tuning.

Name	Description	Iteration Type	Values or Range
anneal_lr	Flag for annealing or not the learning rate	Choice	True or False
ent_coef	Weight for promoting entropy	LogUniform	[1e-9, 1e-3]
batch_size	Number of environment steps per PPO update	Choice	2048, 4096 or 8192
num_minibatches	Number of mini-batches used in PPO optimization	Choice	16, 32, 64 or 128
num_envs	Parallel environments for data collection	Choice	4, 8 or 16
update_epochs	Optimization passes per PPO update	Integer	[4, 10]
hidden_dim	Size of hidden layers in networks	Choice	48, 96, 128 or 256
num_heads	Number of ensemble discriminators used	Choice	8, 16 or 32
s lambda_gp	Regularization term to enforce Lipschitz continuity	Integer	[6, 15]
batch_disc	Number of samples for training the discriminator	Choice	4096, 8192 or 16384
minibatch_disc	Mini-batch size for discriminator	Choice	256, 512 or 1024
std_init	Initial standard deviation for policy output	Uniform	[0.001, 3.0]
lr_pol	Learning rate for policy network	LogUniform	[5e-5, 5e-3]
lr_val	Learning rate for value function network	LogUniform	[5e-5, 1e-3]
lr_disc	Learning rate for discriminator network	LogUniform	[5e-5, 1e-3]
gamma	Weight for future rewards in return calculation	Choice	0.95, 0.975 or 0.99
N	Number of past states	Choice	3 or 5
M	Number of future desired states	Choice	11 or 12

Table 3.12: Summary Costs of Industrial-Scale Implementation

Category	Description	Cost (EUR)
High-Performance Computing (HPC)	GPU-hours, CPU-hours, and storage	1400 – 3200
Simulation Environment	Data Generation, Simulation tool licenses, runtime on workstations, and storage	500 – 2775
Hardware for Deployment	Coral Edge TPU	130
Robotic Manipulator *	ABB GoFa 15000	34000
Personnel	Research Labor and Two full-time PhD students over 3 years	306000 – 428000
Miscellaneous	Publishing, IP, repairs, and contingencies	5–15%
Total		323431-538320

Chapter 4

Results and Analysis

The results are structured as follows. The first section presents the baseline performance, represented by the MPC controller. This is divided into two categories: unconstrained and constrained implementations. The unconstrained MPC includes a generalized cost that penalizes control effort, but it does not enforce strict torque limits. In contrast, the constrained MPC explicitly enforces action limits and is evaluated under two different torque boundaries: nominal and aggressive. For each configuration, metrics are reported on computation time, constraint violations, and trajectory tracking accuracy.

Next, the hyperparameter tuning process is discussed. This section highlights intermediate experiments and justifies several design choices made during development, such as adjustments to the discriminator learning rate that yielded better training stability and performance.

Following this, the unconstrained implementation of the proposed learning-based controller is evaluated. Although no constraints are enforced during training or inference, the system's behavior is assessed using the same set of metrics: inference time, trajectory tracking accuracy, and constraint adherence. In this section, snapshots of the simulation environment are shown.

The final sections analyze the constrained versions of the proposed controller in both nominal and aggressive scenarios. Each scenario is evaluated across all four constraint-handling strategies, with detailed comparisons of inference time, constraint satisfaction, and tracking performance. A representative trajectory is analyzed in depth to illustrate qualitative differences between methods. The section concludes with a comparison between the best-performing configurations in the nominal and aggressive settings, highlighting how constraint feasibility changes under more restrictive action bounds.

4.1 Baseline

As previously discussed, this project introduces an innovative approach that aims to compete with the current state of the art or at least offer a path toward future improvements. For

	Average time per step (ms)	Average standard deviation (ms)	Average max duration (ms)
Trial 1	27.2970	24.4338	66.4617
Trial 2	27.9836	24.5404	67.2348
Trial 3	28.2285	25.6858	68.8325
Average	27.8364	24.8866	67.5097

Table 4.1: Time metrics for the unconstrained MPC implementation based on three repeated trials.

this reason, the evaluation metrics used in this chapter are defined based on a comparison with a method widely recognized as a reference: MPC. This method was implemented under constrained and unconstrained definitions; both are discussed ahead.

4.1.1 Unconstrained MPC

The baseline configuration uses an MPC controller with a control effort penalty weight of 0.01. This value was selected to provide a balanced trade-off between tracking accuracy and control effort. The average computation time required to generate control action is evaluated. Table 4.1 reports the time-related performance metrics, including the average time per step, the standard deviation, and the maximum duration observed. These results are based on three independent trials to capture the variability inherent in desktop computing environments. All tests were conducted on the same hardware configuration: an Intel Core i7-9750H with 12 logical cores. This setup was used to ensure consistency in measuring inference time, while accounting for variability caused by factors such as operating system scheduling and background processes.

Although the MPC configuration used here does not include explicit enforcement of joint torque constraints, it is still analyzed how often the generated control actions exceed the predefined safety margins. This analysis serves as a reference point for evaluating how well the proposed learning-based approaches are able to satisfy physical constraints.

Figures 4.1 and 4.2 show histograms of the control actions produced by the unconstrained MPC controller under two different sets of torque bounds. These histograms help assess how frequently the control signals fall outside the designated safe operating range.

Figure 4.1 corresponds to a moderately permissive constraint scenario, with bounds defined as [-38, 33] for joint τ_1 and [-20, 18] for joint τ_2. In this case, only 1.5% of the actions fall outside the specified bounds, indicating that most actions naturally comply with these nominal constraints. These insights provide a useful benchmark for comparing the constraint-handling performance of learning-based controllers discussed in subsequent sections.

In contrast, Figure 4.2 presents a more restrictive case with bounds set to [-18, 17] for τ_1 and [-12, 12] for τ_2. Under these stricter conditions, approximately 23.86% of the control actions

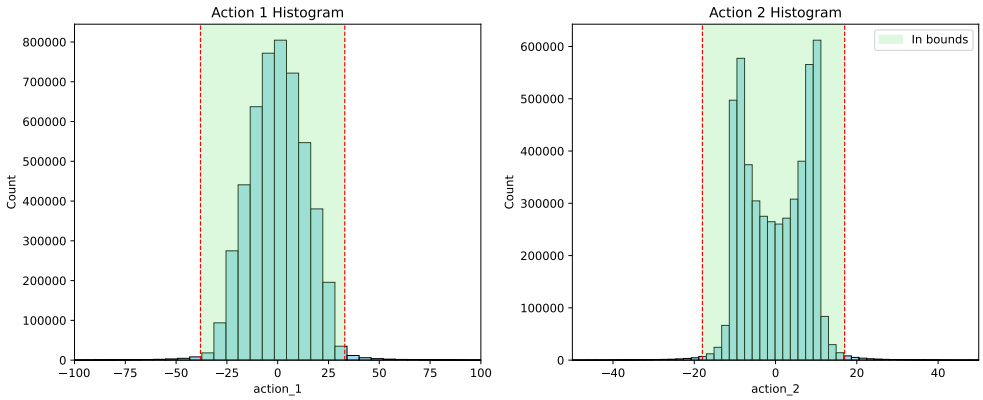


Figure 4.1: Histogram of control actions from the unconstrained MPC under nominal torque limits.

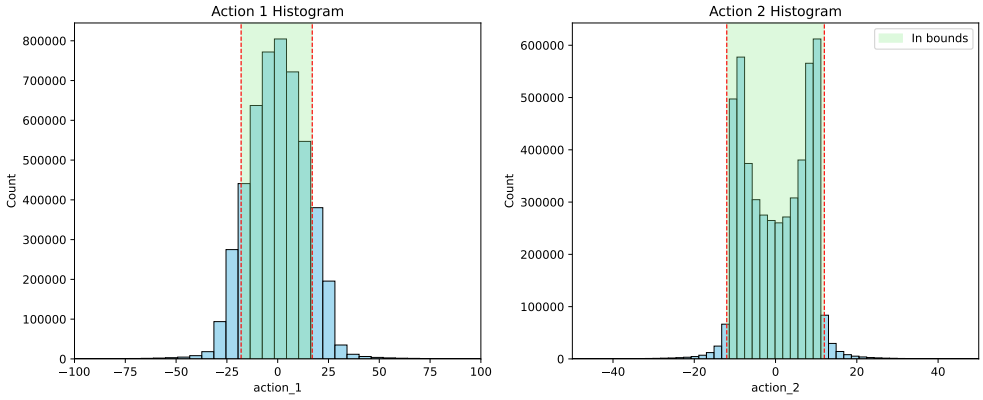


Figure 4.2: Histogram of control actions from the unconstrained MPC under aggressive torque limits.

violate the limits. This result highlights that, while MPC tends to produce smooth control signals, it cannot be assumed to inherently satisfy stricter safety margins unless explicitly enforced.

Under this configuration, the system achieved an average total RMSE of 0.059427 and a standard deviation of 0.100376. This RMSE value combines position and velocity errors equally, without distinction, as described in section 3.2.1.

Table 4.2 presents a breakdown of the tracking performance, comparing the original dataset of 10000 trajectories with the same trajectories generated by the MPC controller. The corresponding histogram that shows the error distribution is included in Figure 3.9.

	Mean	Standard deviation	Maximum
Position (rad)	0.001535	0.003086	0.202552
Velocity (rad/s)	0.036800	0.024172	0.249736

Table 4.2: Trajectory tracking metrics for MPC with control effort consideration (weight = 0.01).

	Mean	Standard deviation	Maximum
Position (rad)	0.001514	0.002133	0.087372
Velocity (rad/s)	0.045088	0.051105	1.638840

Table 4.3: Tracking metrics for MPC with joint torque constraints set to [-38, 33] and [-20, 18] for τ_1 and τ_2 respectively.

	Average time per step (ms)	Average standard deviation (ms)	Average max duration (ms)
Trial 1	34.28258	5.15896	75.58930
Trial 2	34.20248	5.19936	74.47812
Trial 3	33.86116	4.52418	71.12338
Average	34.11541	4.96083	73.73027

Table 4.4: Time metrics for the MPC constrained implementation

4.1.2 Constrained MPC

This section evaluates system performance when explicit joint torque constraints are enforced within the MPC formulation. Unlike the unconstrained case, these limits are implemented as hard constraints within the optimization problem. As a consequence of the nature of the controller, the joint torques are guaranteed to stay within the specified bounds.

Table 4.3 shows the trajectory tracking performance for the first constrained configuration, where torque limits are set to [-38, 33] for τ_1 and [-20, 18] for τ_2. The tracking performance remains close to the unconstrained case shown in Table 4.2, indicating that these bounds are lightly restrictive.

Execution time metrics for this configuration are shown in Table 4.4. The results were obtained under the same hardware and software conditions as in the unconstrained case. The time per step is $\sim$26% higher, which reflects the increased complexity of solving the optimization problem with constraints.

The histogram in Figure 4.3 shows the distribution of control actions for this scenario, in which a distribution with no apparent saturation is shown.

Table 4.5 presents the trajectory tracking results for a more demanding configuration. The torque constraints in this case are reduced to [-18, 17] for τ_1 and [-12, 12] for τ_2. As expected, the overall tracking performance decreases, especially for velocity, for which the error increased by more than twice the value in the previous scenario presented in Table 4.3 due to the more restrictive actuation range.

Execution time results for this stricter setup are reported in Table 4.6. The increase in computational time of 60% with respect to the unconstrained implementation is consistent with the reduced feasible space in the optimization problem. The solver takes more time to

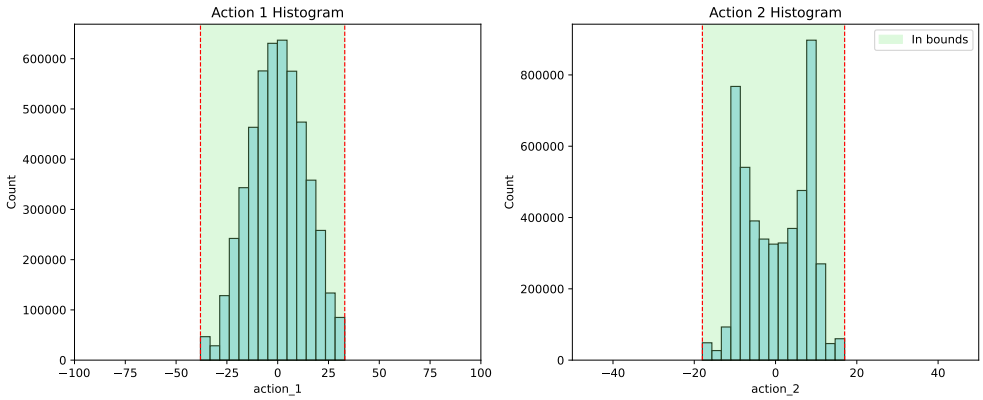


Figure 4.3: Distribution of control actions with nominal constrained MPC

	Mean	Standard deviation	Maximum
Position (rad)	0.001517	0.002063	0.09507
Velocity (rad/s)	0.094733	0.117302	3.181856

Table 4.5: Tracking performance of MPC with stricter torque constraints [-18, 17] for τ_1 and [-12, 12] for τ_2.

converge, as expected under more demanding constraints.

Figure 4.4 shows the distribution of control actions. Compared to the previous case, there is even more accumulation near the limits. This behavior confirms that under tighter constraints, the controller frequently reaches the actuation boundaries in order to meet trajectory tracking requirements.

4.2 Hyperparameter Tuning

This section presents the tuning process used to identify effective hyperparameter configurations for the learning-based controller. The tuning procedure was carried out in two stages. The initial stage aimed to explore a broad set of configurations, while the second stage refined the search around the most promising regions identified in the first iteration. Detailed

	Average time per step (ms)	Average standard deviation (ms)	Average max duration (ms)
Trial 1	42.88414	7.24112	85.55038
Trial 2	44.79244	8.0031	91.77184
Trial 3	43.47158	7.15542	84.59952
Average	43.71605	7.46654	87.30725

Table 4.6: Execution time metrics for MPC with strict joint torque limits [-18, 17] and [-12, 12].

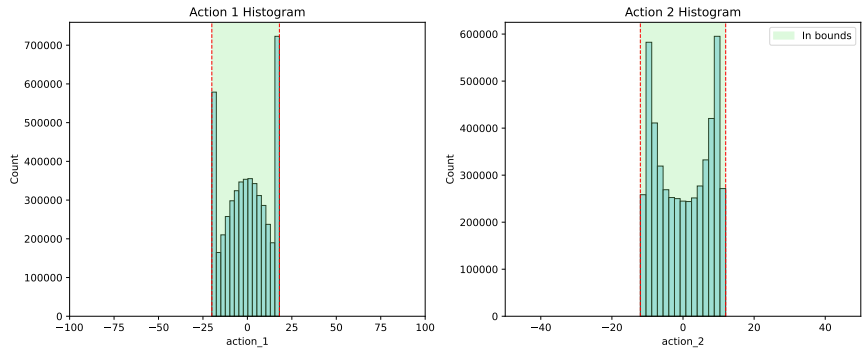


Figure 4.4: Distribution of control actions with aggressively constrained MPC

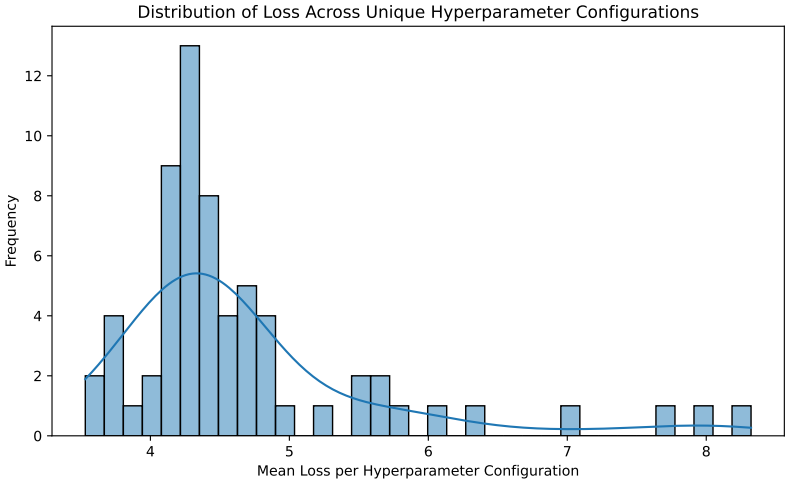


Figure 4.5: Distribution of final losses across different configurations during the first hyperparameter tuning stage.

results from the first tuning phase are included in appendix A.1 .

In the second tuning phase, the search ranges is reduced, while the number of tuned hyperparameters is expanded. Table 3.10 lists all hyperparameters included in the final stage.

Figure 4.5 presents the distribution of final loss values across different configurations evaluated during the first tuning stage. Most trials converged to loss values around 4.2, with the best configurations reaching approximately 3.5. This distribution served as a useful reference to narrow the search space for the second stage.

The second stage of tuning, guided by the observations from the first phase, did not reveal any new configurations with significantly improved performance. Figure 4.6 shows a similar distribution to that observed in the best performing configurations of the first tuning stage, suggesting that the most performant regions had already been covered in the earlier trials.

Trials in both stages have a computational budget ranging from 300 000 to 2 million environment steps. The AutoML framework used includes an early stopping mechanism, so not all configurations reached the upper bound. This limitation was introduced due to the high computational cost associated with training and the practical constraints of time and

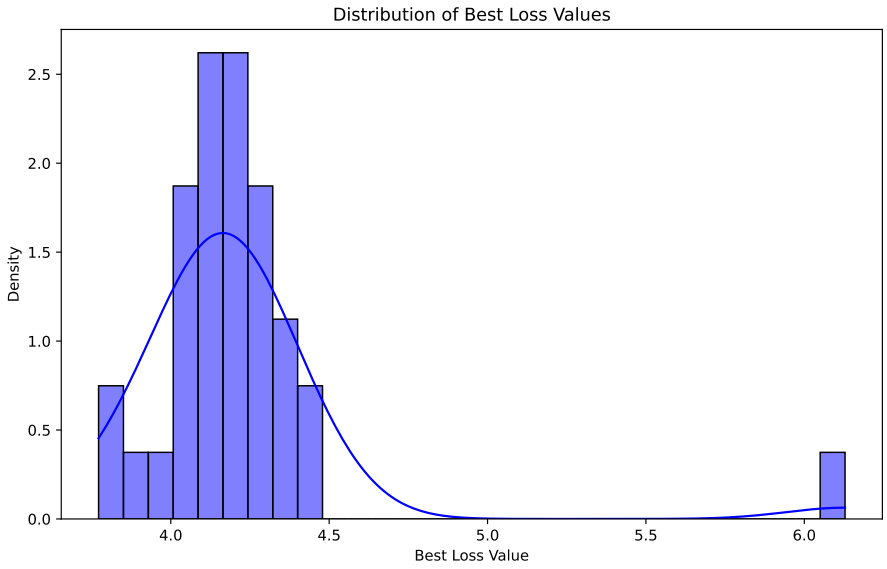


Figure 4.6: Loss distribution for the second hyperparameter tuning.

available resources.

The best-performing configuration identified during tuning was selected for extended training. An initial run of 10 million steps was conducted, followed by a longer 30 million-step training session once promising improvements were observed. This extension aimed to ensure convergence and to further exploit the generalization capabilities of the learned policy.

Table 4.7 summarizes the optimal configuration obtained through the tuning process.

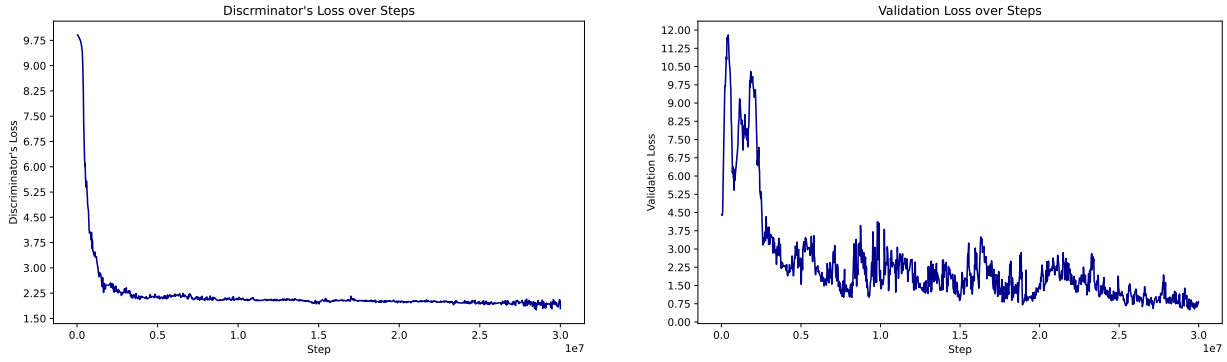
A comparison between the 10 million and 30 million-step models revealed significant improvements after extended training. Specifically, a 63.33% reduction in position error and a 66.55% reduction in velocity error were observed relative to the 10M version. The specific obtained results are discussed in 4.3.

An important training behavior was identified while analyzing the evolution of losses during learning. Figure 4.7 shows a steep decline in the discriminator ensemble's loss, indicating that the ensemble initially struggles to differentiate between expert and agent-generated trajectories. The system goes through approximately 2 million steps to reach a competitive point. Figure 4.7 also shows that the policy begins to improve only after the discriminator loss stabilizes around a value of 2. As a result of this dynamic, the discriminator causes the policy to also require around 2 million steps to converge, which delays meaningful policy improvement.

To address this issue and reduce the required training time, the discriminator learning rate was increased from 4.12×10^{-5} to 5.12×10^{-4}. This represents a twelvefold increase over the initially tuned value. Additionally, learning rate annealing for the discriminator ensemble was introduced. This approach accelerates early convergence of the discriminator while preserving long-term stability and avoiding vanishing gradients, which was the primary motivation for introducing the ensemble and its tailored loss formulation. Figure 4.8 confirms

Hyperparameter	Optimal Value
Anneal Learning Rate	False
Entropy Coefficient	0.00018
Batch Size	2048
Number of Minibatches	64
Number of Environments	16
Update Epochs	7
Hidden Dimension	256
Number of Discriminator Heads	8
λ_{GP} (Gradient Penalty)	8
Discriminator Batch Size	4096
Discriminator Minibatch Size	1024
Initial Std. Deviation	3.0
Policy Learning Rate	5×10^{-5}
Value Function Learning Rate	4.43×10^{-4}
Discriminator Learning Rate	4.12×10^{-5}
Discount Factor (γ)	0.975
Past Horizon (N)	5
Future Horizon (M)	11

Table 4.7: Best hyperparameter configuration selected after hyperparameter tuning.

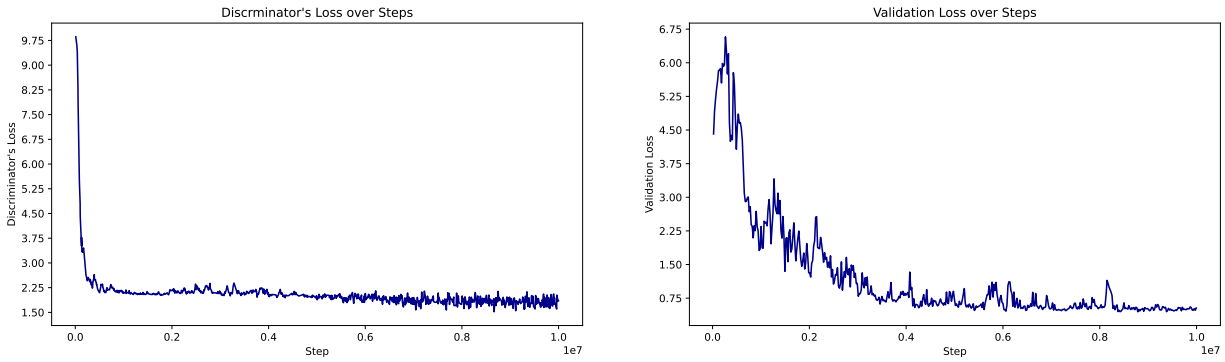


(a) Discriminator loss with LR of 4.12×10^{-5}.

(b) Validation loss during training.

Figure 4.7: Loss evolution using the original discriminator learning rate (4.12×10^{-5}), showing delayed policy improvement due to slow discriminator convergence.

that the discriminator now stabilizes around a loss of 2 in fewer than 500,000 steps, significantly faster than before. Figure 4.8 further supports the benefit of this change, showing that the policy converges to comparable or better performance in substantially less time.



(a) Discriminator loss with LR of 5.12×10^{-4}.

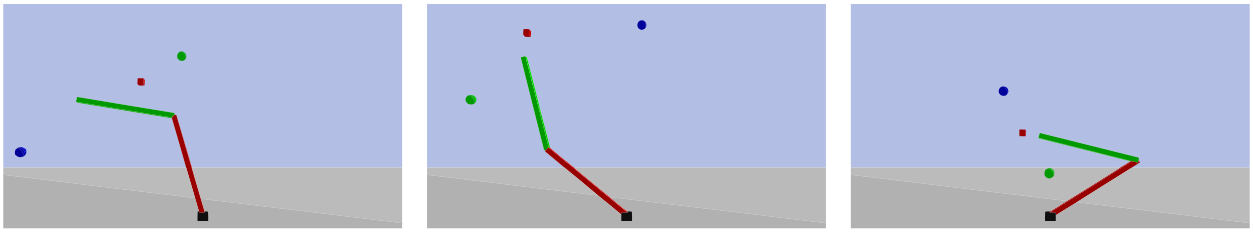
(b) Validation loss during training.

Figure 4.8: Loss evolution using an increased discriminator learning rate (5.12×10^{-4}), leading to faster policy convergence.

4.3 Unconstrained Implementation

This section addresses the second specific objective of this work: the development of a neural network capable of generating control signals for a planar robot with at least two degrees of freedom. The system was evaluated on a dataset consisting of 1010 trajectories, each containing 500 time steps.

Figures in reference 4.9 show three representative scenarios selected from the 1010 trajectories used to evaluate the system. The green sphere marks the initial position, the blue sphere indicates the goal, and the red box represents an obstacle that increases the complexity of the task by requiring obstacle-aware trajectory execution.



(a) Trajectory 10

(b) Trajectory 18

(c) Trajectory 19

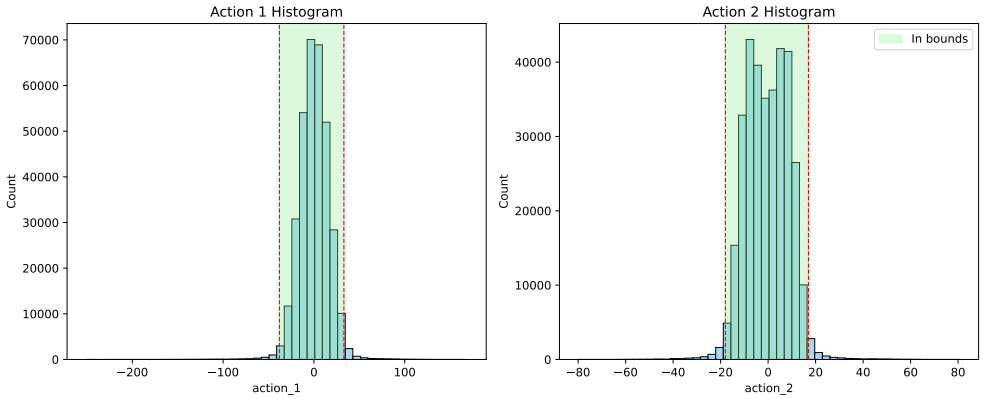
Figure 4.9: Rendered snapshots from different episodes of the simulation, showing distinct steps in the execution of trajectory-following tasks.

Although the policy is not explicitly constrained in any way during training, it is still relevant to evaluate how well the generated control actions remain within predefined safe torque bounds. This analysis reveals the degree to which the learned policy has developed implicit awareness of physical limitations, and it serves as a reference for assessing the benefits of different explicitly constrained training setups.

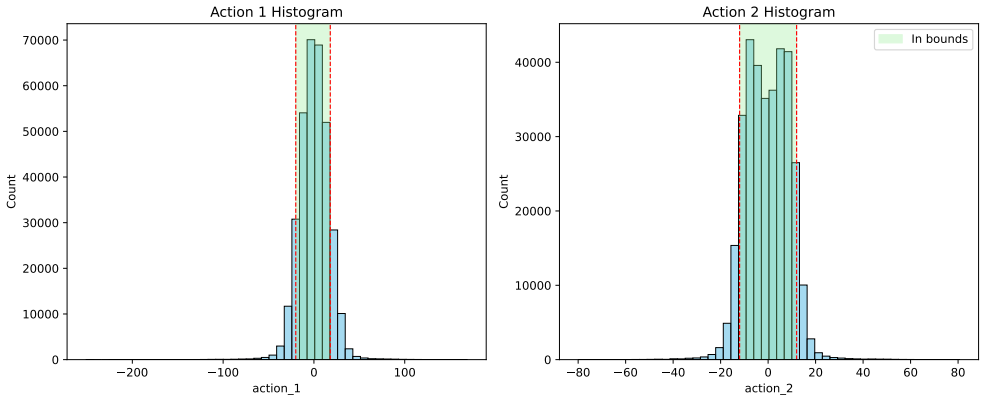
Figures 4.10a and 4.10b show the distribution of the generated control actions under two different constraint regimes. The first scenario uses nominal bounds of $[-38, 33]$ for τ_1 and $[-20, 18]$ for τ_2, while the second scenario applies more aggressive limits of $[-18, 17]$ and

$[-12, 12]$ respectively.

In the nominal setting, only 4.39 percent of actions violated the torque limits. Under the aggressive regime, the violation rate increased to 31.2 percent. These results represent a considerable improvement over the 10 million-step model, which exhibited violation rates of 80.07 percent and 91.63 percent, respectively. This improvement demonstrates that extended training allowed the policy to transition from exploratory behavior to a more stable and physically feasible control strategy. Figure 4.10 compares the control action distributions under nominal and aggressive constraints, showing a clear increase in violations in the latter.



(a) Nominal constraint bounds



(b) Aggressive constraint bounds

Figure 4.10: Histograms of control actions for the unconstrained policy under (a) nominal and (b) aggressive torque constraint bounds.

Inference time metrics for the 30 million-step model were obtained using the same hardware configuration as the MPC experiments: an Intel Core i7-9750H CPU with 12 logical cores. On average, the neural policy achieved an inference time of 2.1 ms per control step, with a standard deviation of 0.6 ms. However, occasional latency spikes were observed, reaching up to 220.5 ms. These outliers are likely attributed to operating system-level interruptions such as background processes and the absence of real-time execution guarantees, rather than to inefficiencies in the model itself.

It is important to note that the neural policy was executed entirely on the CPU, without employing inference-time optimizations such as model tracing, scripting, or GPU acceler-

	Mean	Standard deviation	Maximum
Position (rad)	0.2136	0.1563	1.5273
Velocity (rad/s)	0.4080	0.3129	48.8857

Table 4.8: Trajectory tracking metrics for the unconstrained policy trained for 30 million steps.

Method	Mean (ms)	Std (ms)	Max (ms)
Constrained Expert	2.3	0.7	208.8
Penalty	2.1	0.7	217.1
Clipping	2.1	0.6	204.0
Penalty + Clipping	2.0	0.6	198.7
Average	2.125	2.6	207.15

Table 4.9: Average, standard deviation, and maximum inference time per step for the nominal constraints configurations.

ation. Despite these constraints, the system demonstrated an inference speed that is well within the requirements for real-time operation, indicating that with appropriate deployment strategies, the controller can be used in practical settings.

Table 4.8 presents the trajectory tracking performance of the final model, which was trained for 30 million steps. The results show the average, standard deviation, and maximum observed values for both position and velocity tracking errors.

4.4 Constrained Implementation: Nominal Scenario

This section evaluates the constrained implementation under nominal torque limits and addresses the third and fourth specific objectives of this work. The third objective consists in incorporating mechanisms to handle physical constraints and variations in the system's dynamic behavior. The fourth aims to validate the system's performance under realistic task conditions. The evaluation focuses on inference time, constraint enforcement, and trajectory tracking accuracy.

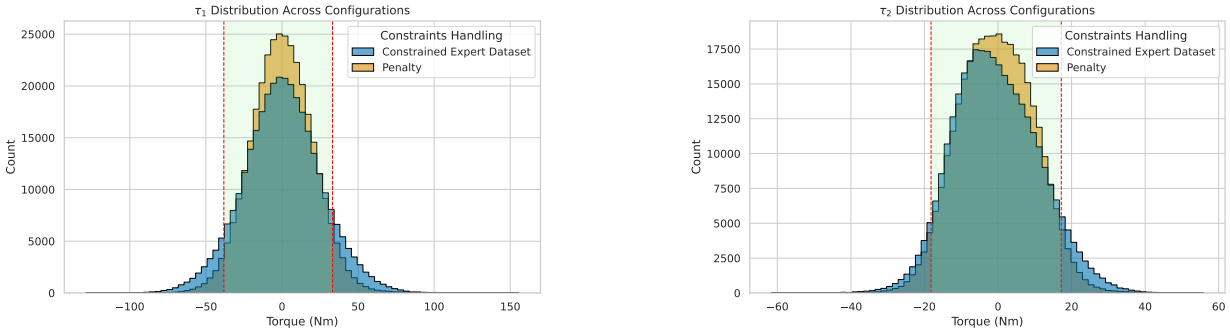
All variants were trained under identical architectural assumptions, differing only in the constraint handling strategies employed. As a result, their inference times are consistently low, remaining below 2.5 milliseconds per step on average. The configuration that combines penalty, clipping, and constrained expert data achieved the best overall timing, with an average of 2.0 milliseconds and a peak of 198.7 milliseconds. The rest remained close, with variations in maximum latency caused by infrequent spikes. Table 4.9 presents a unified view of inference timing metrics for all four constrained configurations.

Compared to the previously developed MPC-based controller, which required an average of 34.115 milliseconds per step, this represents a more than thirteenfold reduction in compu-

tational cost. This gain reinforces the suitability of learning-based methods for real-time execution.

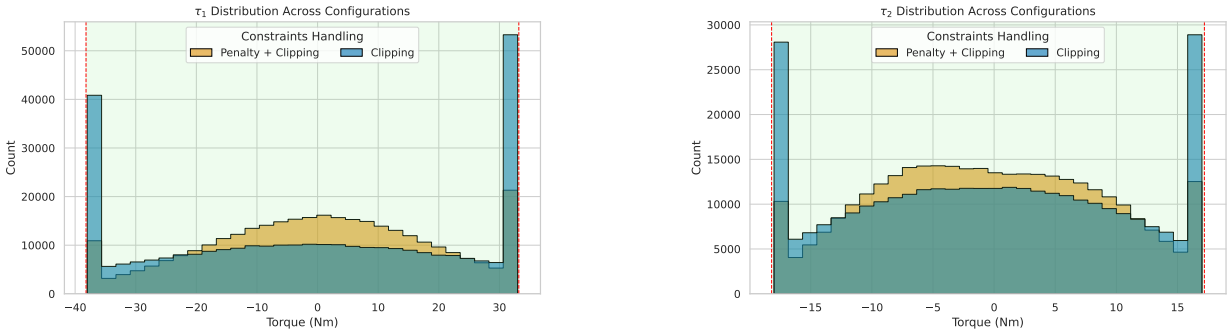
Control compliance varied significantly depending on the enforcement strategy. The configuration trained exclusively with constrained expert data yielded 25.4% of control actions exceeding the nominal torque bounds. Introducing a penalty term into the reward function reduced this rate to 14.35%, showing that soft penalties can guide the policy toward safer actuation. However, neither approach could guarantee strict constraint satisfaction.

Figures 4.11 and 4.12 show histograms of the control actions for τ_1 and τ_2 under constrained implementations without and with action clipping, respectively.



(a) τ_1 **(b)** τ_2

Figure 4.11: Histograms of τ_1 and τ_2 for the constrained implementations without clipping.



(a) τ_1 **(b)** τ_2

Figure 4.12: Histograms of τ_1 and τ_2 for the constrained implementations with clipping.

Nevertheless, their behavior differed considerably; the version relying only on clipping exhibited clear saturation at the torque boundaries, as shown in the action histograms. In contrast, the combined method, which also incorporated a penalty term and expert demonstrations, showed a more uniform distribution of torques, indicating that the policy learned to operate within safe regions without consistently relying on hard limits. This distinction is important, as it suggests that combining soft and hard constraint mechanisms results in safer and more controlled behavior than using either alone.

A summary of all results is shown in Table 4.10, only the methods that include action

clipping achieved full compliance. Without clipping, violations ranged from 14 to over 25 percent of the control actions.

Method	Torque Violations (%)
Constrained Expert	25.40
Penalty	14.35
Clipping	0.00
Penalty + Clipping	0.00

Table 4.10: Percentage of control actions exceeding nominal torque boundaries.

In terms of tracking performance, the method using only expert data already achieved reasonable results, but with notably higher violation rates. The penalty-only approach improved constraint satisfaction and slightly reduced the mean position error. The pure clipping configuration, while fully safe, resulted in a clear deterioration of tracking quality, with the highest position error (0.1622 rad). Finally, the best performance was obtained by the configuration that combined constrained expert data, a penalty term, and clipping. It achieved the lowest mean position error (0.0790 rad) and the lowest variance (0.0626), while maintaining very competitive velocity tracking.

Table 4.11 summarizes the trajectory tracking results across all constrained approaches.

	Position (rad)		Velocity (rad/s)	
Method	Average RMSE	Std. RMSE	Average RMSE	Std. RMSE
Expert Data	0.1013	0.0651	0.3543	0.2747
Penalty	0.0942	0.0657	0.3624	0.3131
Clipping	0.1622	0.1132	0.4052	0.3138
Penalty-Clipping	0.079	0.0626	0.3649	0.3286

Table 4.11: Trajectory tracking performance for each configuration.

Clipping ensures safety; however, it significantly degrades control quality in this case. The pure clipping strategy resulted in the worst accuracy and the highest velocity overshoot. In contrast, the combined approach maintained perfect constraint compliance while improving both average and worst-case tracking metrics. This suggests that hybrid strategies combining soft and hard constraint mechanisms enable more reliable and efficient control.

Although all constrained methods outperformed the unconstrained baseline in terms of position RMSE (0.2136 rad), none of them matched the level of precision achieved by the MPC controller, which reached an average RMSE of 0.001517 rad . However, this level of precision comes at a high computational cost and limited flexibility. The combined approach presented here offers a more balanced and practical alternative.

Figure 4.13 shows the same trajectory (ID 280) executed using the four different controllers trained with distinct constraint-handling strategies. In Figure 4.13a, the expert dataset

configuration achieves good trajectory tracking performance, although some control actions exceed 100 Nm, significantly surpassing the nominal torque limits.

In Figure 4.13b, corresponding to the configuration with a penalty term, the position profiles more closely match the reference, and the velocity signal is less noisy. This improved behavior is reflected in the torque profiles, which remain within a smaller range, though still reaching values near 60 Nm; well above the defined constraints.

Figure 4.13c illustrates the results obtained using a controller trained with action clipping only. In this case, tracking performance noticeably deteriorates, which can be attributed to the heavy saturation observed in the torque profiles. The system failed to learn compensatory behaviors under the imposed limits, resulting in degraded performance.

Finally, Figure 4.13d shows the controller trained with both a penalty term and action clipping. This configuration achieved the best overall performance, even surpassing the unconstrained expert in tracking quality. The controller not only respected the action bounds but also demonstrated a learned behavior that effectively compensates for the limited control range. Additionally, since the actions are clipped, any isolated poor control signal remains bounded, reducing its negative impact on trajectory deviation.

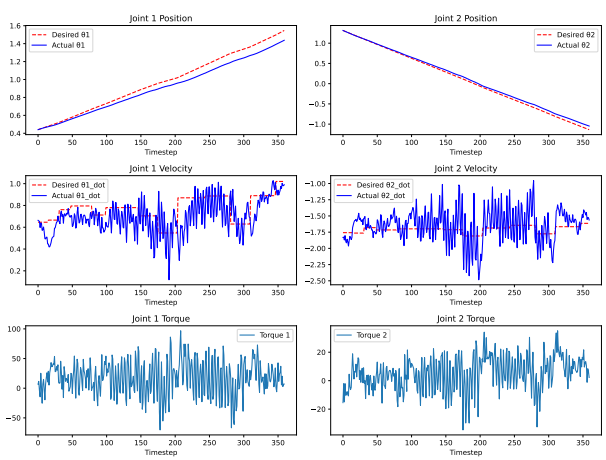
4.5 Constrained Implementation: Aggressive Scenario

This section extends the evaluation of the third and fourth specific objectives to a more demanding setup. The aggressive scenario introduces significantly tighter torque constraints during training: joint 1 is limited to [-18, 14] and joint 2 to [-10, 9]. These limits emulate strict operational boundaries, typical of safety-critical environments or hardware with limited actuation capacity, and are intended to test the control system near its feasible limits. All four previously discussed constraint-handling strategies were assessed under these more restrictive conditions.

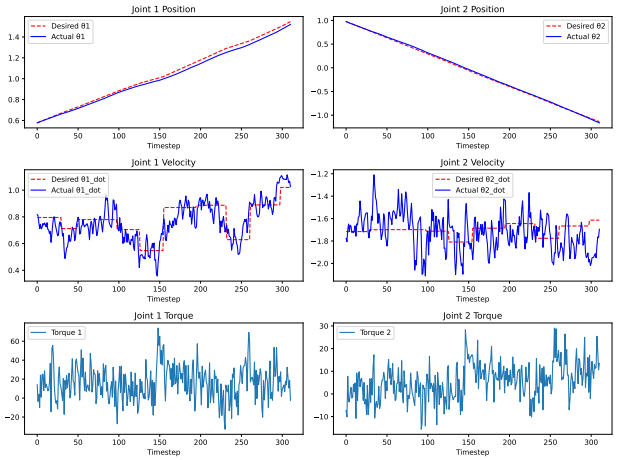
Inference time metrics are summarized in Table 4.12. All methods demonstrated real-time viability, with average per-step durations between 2.2 ms and 2.4 ms. As in prior scenarios, rare outliers led to maximum durations exceeding 200 ms, but these are not representative of typical performance. Crucially, the enforcement strategy had negligible impact on average computational cost.

Constraint adherence under aggressive torque limits is illustrated in Figures 4.14 and 4.15, which show the torque action distributions for τ_1 and τ_2 without and with action clipping, respectively. In the non-clipping configurations, a significant portion of control actions exceeded the torque bounds, particularly in the expert-only model. The penalty-based strategy mitigated these violations to some extent, highlighting the benefit of constraint-aware reward shaping. In contrast, both clipping-based methods achieved full constraint compliance, with all actions staying within limits. However, the pure clipping approach exhibited saturation at the torque boundaries, indicating reduced policy flexibility.

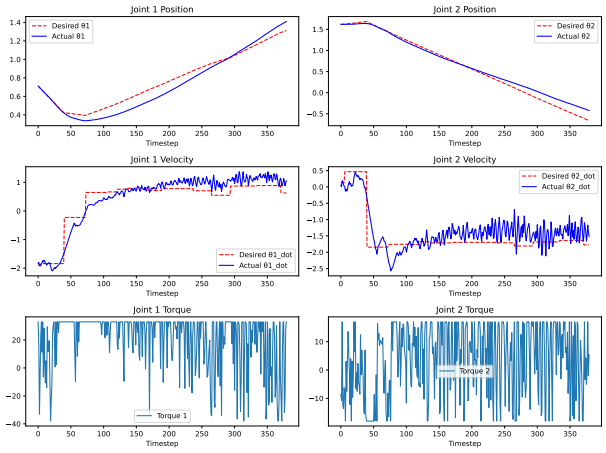
Constraint violations across the four configurations are summarized in Table 4.13. The re-



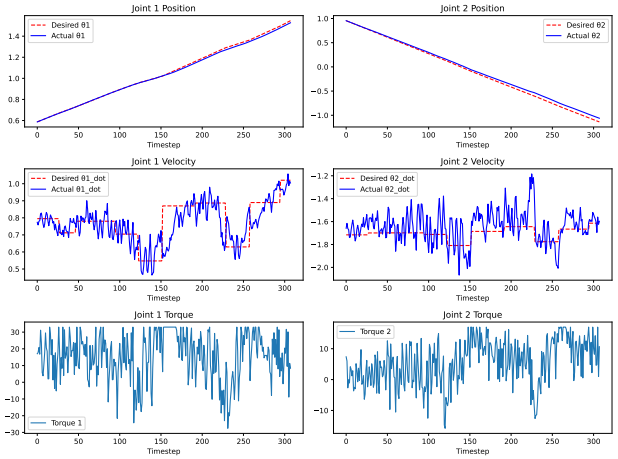
(a) Expert Dataset



(b) Penalty Term



(c) Clipping



(d) Penalty Term + Clipping

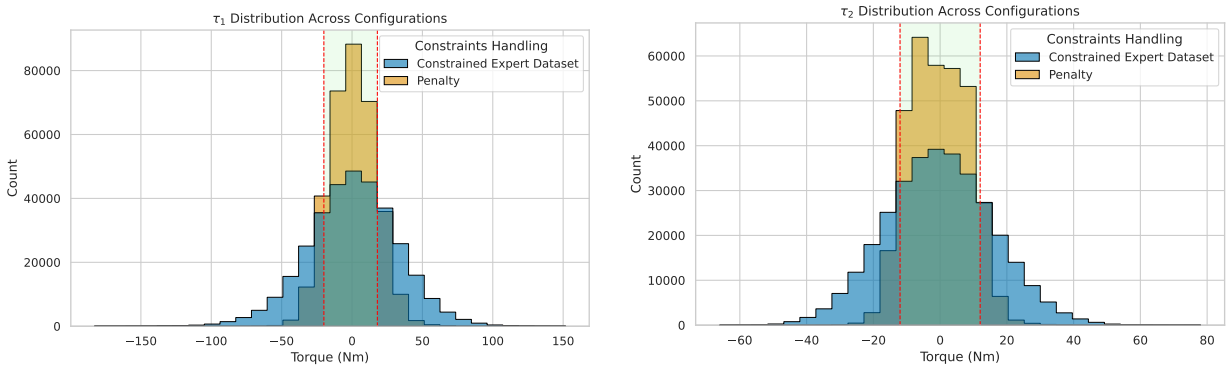
Figure 4.13: Results of the same trajectory for the four different methods of constraint handling.

Method	Mean (ms)	Std (ms)	Max (ms)
Constrained Expert	2.4	0.7	198.4
Penalty	2.3	0.9	261.7
Clipping	2.2	0.8	220.1
Penalty + Clipping	2.2	0.8	234.3
Average	2.275	0.8	228.625

Table 4.12: Average, standard deviation, and maximum inference time per step for the nominal constrained configurations.

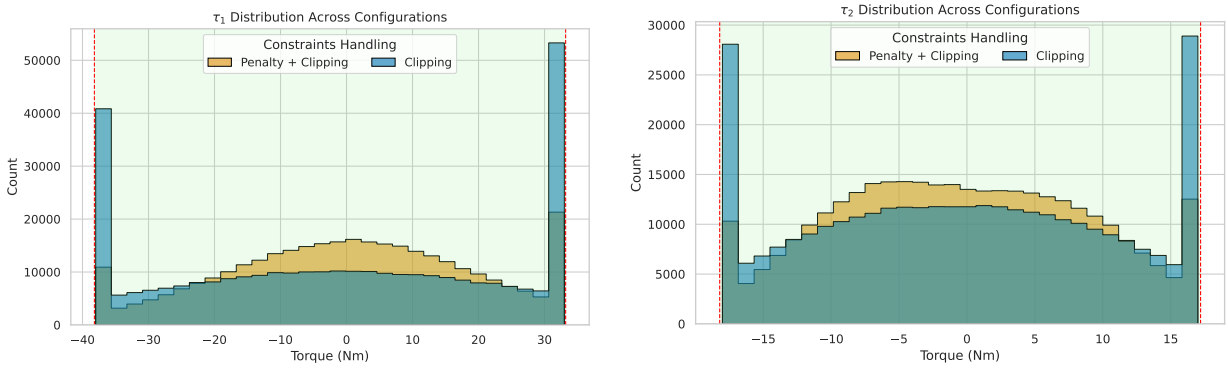
sults show a clear progression in constraint compliance, with the penalty-based approach significantly reducing the violation rate compared to the baseline. Both clipping-based strategies achieved complete enforcement, confirming the effectiveness of explicit control saturation mechanisms in aggressive settings.

Trajectory tracking metrics are presented in Table 4.14. The constrained expert config-



(a) τ_1 **(b)** τ_2

Figure 4.14: Histograms of τ_1 and τ_2 for the aggressively constrained implementations without clipping.



(a) τ_1 **(b)** τ_2

Figure 4.15: Histograms of τ_1 and τ_2 for the aggressively constrained implementations with clipping.

Method	Torque Violations (%)
Constrained Expert	74.01
Penalty	36.23
Clipping	0.00
Penalty + Clipping	0.00

Table 4.13: Percentage of control actions exceeding aggressive torque boundaries.

uration resulted in the lowest average position error (0.8001 rad) and moderate velocity accuracy, but it also produced the highest rate of torque violations. This indicates that the torque limits defined for this scenario exclude feasible solutions for many of the reference trajectories. The penalty-only configuration maintained similar velocity performance and introduced a slight increase in position error (0.1207 rad), achieving a better balance between feasibility and constraint adherence. In contrast, the clipping-only model showed the worst tracking results across both position (0.1993 rad) and velocity (0.5838 rad/s) metrics. This reinforces the idea that applying hard limits without additional guide for the learning process severely restricts the policy's ability to explore effective solutions. The configuration that combined penalty and clipping achieved slightly better performance than clipping alone, particularly in velocity tracking. This improvement is attributed to the role of the penalty term, which encourages lower-magnitude actions during training and helps the policy operate more effectively within the reduced torque range.

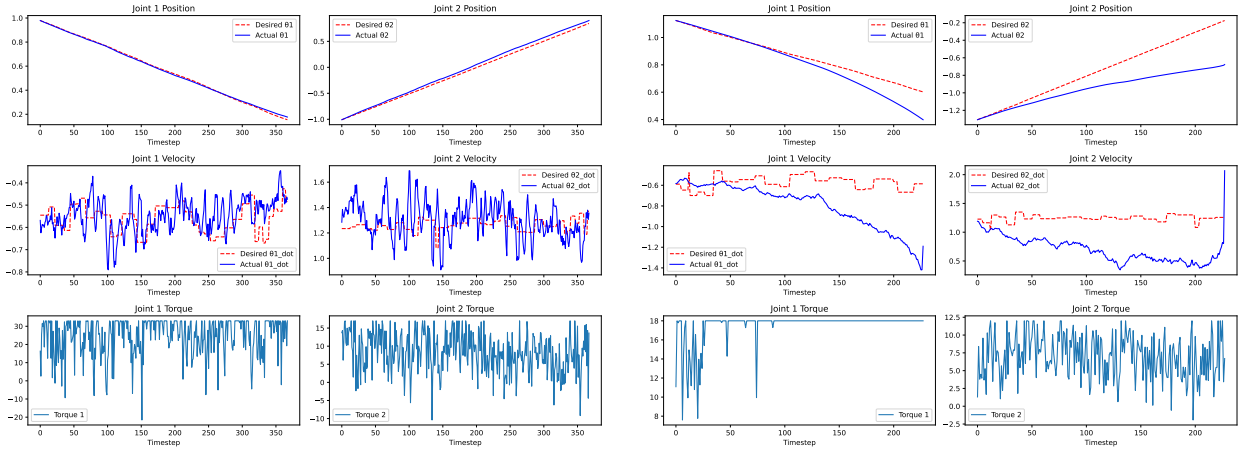
	Position (rad)		Velocity (rad/s)	
Method	Average RMSE	Std. RMSE	Average RMSE	Std. RMSE
Expert Data	0.0801	0.0569	0.4091	0.3464
Penalty	0.1207	0.1004	0.4146	0.381
Clipping	0.1993	0.1499	0.5838	0.4873
Penalty-Clipping	0.1990	0.1638	0.5322	0.4600

Table 4.14: Trajectory tracking performance for each configuration for aggressive torque boundaries.

Figures 4.16a and 4.16b present the position, velocity, and control action profiles for the same trajectory executed by two different controllers. All quantities follow standard units: position in radians, velocity in radians per second, and torque in Nm. In each plot, the red curves represent the reference trajectory generated by the planner, while the blue curves correspond to the actual trajectory followed by the robot under the learned control policy.

Figure 4.16 compares the behavior of the same trajectory under two controllers trained with different torque constraint configurations. The left plot illustrates the performance of the controller trained with nominal torque limits. Despite moderate torque saturation, the position and velocity profiles closely follow the desired references, resulting in an accurate and safe trajectory execution. In contrast, the right plot shows the result of using a controller trained under more aggressive torque constraints. In this case, the velocity profile begins

to deviate significantly from the reference, particularly in dynamic segments of the motion. This degradation is caused by severe torque saturation, suggesting that the reduced torque range was insufficient to generate the required motion, ultimately compromising trajectory tracking.



(a) Nominal torque limits

(b) Aggressive torque limits

Figure 4.16: Comparison of tracking performance for the same trajectory using controllers trained with different torque limits.

Chapter 5

Conclusions and Recommendations

This work presented a learning-based control system for torque-driven robotic manipulators and evaluated its effectiveness across unconstrained and constrained scenarios. The system was benchmarked against a classical MPC baseline and assessed using quantitative performance metrics aligned with the second, third, and fourth specific objectives of this work.

The first objective was fulfilled by establishing a solid analytical foundation for the project. This involved identifying key challenges in generating control signals that are both safe and efficient in environments with static obstacles, particularly under physical constraints. The study critically examined the limitations of classical approaches , specially from, in terms of computational cost and scalability. Based on this analysis, a clear set of performance metrics was defined, including position and velocity RMSE, constraint violation rates, and inference time. These metrics guided the design and evaluation of the proposed system throughout the process.

Regarding the second objective, which aimed to develop a controller for a planar robot with at least two degrees of freedom, the results demonstrate clear success. The unconstrained policy trained for 30 million steps achieved an average position RMSE of 0.2136 radians and a velocity RMSE of 0.408 rad/s. While the position error slightly exceeded the marginal threshold ($\leq$0.2 rad), the velocity metric met the ideal criterion ($\leq$0.4 rad/s). More importantly, the average inference time of 2.1 ms per control step, with a standard deviation of 0.6 ms, satisfied both the ideal and marginal requirements. These outcomes confirm the system's suitability for real-time execution and validate that a learning-based controller can serve as a viable alternative to traditional MPC under comparable computational budgets. Furthermore, one of the constrained configurations, specifically the one that combined expert data, penalty shaping, and control action clipping, met all performance targets, including both position and velocity RMSE. In this configuration, occasional control actions that could otherwise degrade trajectory accuracy were effectively limited by the clipping mechanism. This ensured that when suboptimal decisions occurred, their impact on the position or velocity remained bounded. As a result, the system maintained accurate behavior while respecting physical constraints.

The third objective involved extending the controller to enforce physical constraints, such as joint torque limits. Several constraint-handling strategies were explored, including expert data conditioning, reward penalties, and hard clipping. The hybrid approach combining all three mechanisms achieved the best overall performance, ensuring 100 percent compliance with torque limits in both the nominal and aggressive scenarios. In the nominal case, this same configuration also met all other system-level specifications, including the average and standard deviation of inference time, the average RMSE for both position and velocity, and full constraint adherence. All values reached either the marginal or ideal target. This confirms that the proposed learning-based system, when properly trained with combined constraint mechanisms, is not only viable but optimal for real-time robotic control under nominal conditions.

In contrast, the aggressive constraint scenario was designed to assess how the system performs when the feasible actuation space is significantly reduced. Although the controller trained under these more demanding conditions did not fully meet the original tracking performance thresholds, this outcome was expected. The purpose of this scenario was not to enforce strict success but to investigate how much performance can be retained when operating under tight physical limitations. In the best-performing configuration, the average position RMSE reached 0.1207 radians, which remains within the marginal specification ($\leq$0.2 rad), while the velocity RMSE increased to 0.4146 rad/s, slightly exceeding the ideal target of 0.4 rad/s by just 3.65 percent. Despite this small deviation, the system was still able to guarantee full constraint compliance using hybrid strategies. These results highlight the flexibility of the proposed architecture and its ability to operate near the boundary of feasibility while preserving safety and stability.

The fourth objective centered on validating the controller's robustness in realistic, obstacle-rich environments. Across all constrained configurations, the hybrid method consistently achieved the lowest standard deviation in position RMSE. In the nominal scenario, it reached 0.0626 radians, which is well below the ideal target of 0.15. This demonstrates both precision and consistency. Even under aggressive constraints, the same configuration remained competitive. This indicates that the controller is capable of generalizing across diverse operating regimes while maintaining controlled behavior, even when tracking becomes increasingly difficult due to actuation limits.

Finally, the training process itself yielded important insights. Stable learning was only achieved after increasing the discriminator ensemble's learning rate and introducing annealing, which accelerated early convergence and produced a more informative reward signal. This change allowed the policy to reach competent behavior in fewer steps and contributed to achieving the system's performance targets. The interdependence between the policy and discriminator dynamics was essential to successful training, confirming that proper tuning of reward generation is a determining factor in constrained learning-based control.

In addition to the results obtained, several recommendations have been identified that could strengthen and extend the proposed system. A first line of work involves validating the controller directly on physical hardware. Deploying the system on real robots would make it possible to verify its behavior under real-world conditions, where factors such as sensor

noise, actuator discretization, communication latency, and mechanical disturbances directly impact control stability and precision. This validation would not only confirm whether inference times remain within acceptable limits on real platforms but also help identify practical limitations that cannot be fully captured in simulation.

Separately, although the current system meets real-time requirements when running on a CPU, it would be recommended to explore optimized versions of the model using hardware accelerators. These improvements could be particularly valuable when scaling the architecture to manipulators with higher degrees of freedom, as the computational cost per control step is expected to increase with observation dimensionality and system complexity.

As for future work, a natural extension involves increasing the number of degrees of freedom of the manipulator to assess the system's applicability to current industrial platforms, where six or even seven degrees of freedom are standard. This extension would allow for the study of scalability in terms of training duration, inference speed, and generalization capability when facing more complex and redundant dynamics.

Another immediate direction involves explicitly incorporating joint-level position and velocity constraints. While the present work focused primarily on torque limits as the main physical restriction, adding limits on joint positions and velocities would allow the system to exclude unsafe or undesirable regions of the workspace. This is especially relevant in collaborative settings or constrained environments, where certain areas must be avoided to ensure safety or prevent mechanical damage.

Additionally, a promising research involves shifting from joint-level constraints to force-based control at the tool center point (TCP). Rather than defining limits at the joint level, one could formulate restrictions based directly on the magnitude and direction of the end-effector forces. This aligns better with safety requirements in physical human-robot interaction, where regulating contact forces is one of the most critical components for safe collaboration.

It is also expected that mechanisms will be implemented to allow the system to perform automatic calibration based on data collected in real time. This capability would enable the controller to adapt its internal parameters or behavior dynamics without manual intervention, supporting its deployment under changing conditions such as varying friction, or mechanical wear. Such adaptability is essential to ensure robust performance in realistic scenarios where conditions cannot remain constant.

Finally, although the results achieved in this work were satisfactory and, in many cases, exceeded the defined thresholds, there is still room to improve trajectory tracking accuracy. Narrowing the performance gap with state-of-the-art model-based approaches remains a valid and challenging objective. Bridging this gap would further justify the use of deep learning-based methods and support their adoption in applications where tracking precision is critical.

.

Bibliography

[1] DIN – German Institute for Standardization, "Recommendations for implementing the strategic initiative industrie 4.0: Securing the future of german manufacturing industry," https://www.din.de/resource/blob/76902/e8cac883f42bf28536e7e8165993f1fd/recommendations-for-implementing-industry-4-0-data.pdf, 2016, accessed: May 2025.

[2] International Federation of Robotics, "How robots work alongside humans," https://ifr.org/ifr-press-releases/news/how-robots-work-alongside-humans, 2023, accessed: May 2025.

[3] H. Alabool, M. H. Jamaludin, R. A. Rahman, F. Khan *et al.*, "The expanding role of artificial intelligence in collaborative robots for industrial applications: A systematic review of recent works," *Machines*, vol. 11, DOI 10.3390/machines11010111, no. 1, p. 111, 2023. https://www.mdpi.com/2075-1702/11/1/111

[4] N. Kazantsev, S. Ostanin, and M. Krasheninnikov, "Artificial intelligence in robot control systems," *IOP Conference Series: Materials Science and Engineering*, vol. 363, DOI 10.1088/1757-899X/363/1/012013, no. 1, p. 012013, 2018. https://ui.adsabs.harvard.edu/abs/2018MS%26E..363a2013K

[5] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robotic Modeling and Control*, 1st ed. JOHN WILEY & SONS, INC., 2005.

[6] SHENCHONG. Robotic manipulator: Definition, parts, applications. (2025) Accessed: Mar. 2025. https://www.shenchong.com/robotic-manipulator-definition-parts-applications.html

[7] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modeling, Planning and Control*. London: Springer London, 2009.

[8] J. J. Craig, *Introduction to robotics: Mechanics and control*, 4th ed. NY NY: Pearson, 2018.

[9] K. Idrissi. Degrees of freedom of a robot. (2020) Accessed: Mar. 2025. https://medium.com/@khalil_idrissi/degrees-of-freedom-of-a-robot-c21624060d25

[10] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Cham: Springer International Publishing, 2016.

[11] MathWorks. What is inverse kinematics? (2025) Accessed: Mar. 2025. https://de.mathworks.com/discovery/inverse-kinematics.html

[12] Akshit Lunia, Ashley Stevens, Cavender Holt, Rhyan Morgan, Joshua Norris, Shailendran Poyyamozhi, and Yehua Zhong, "Modeling, motion planning, and control of manipulators and mobile robots."

[13] R. N. Jazar, *Theory of Applied Robotics*. Boston, MA: Springer US, 2010.

[14] A. A. Okubanjo, O. K. Oyetola, M. O. Osifeko, O. O. Olaluwoye, and P. O. Alao, "Modeling of 2 dof robot arm and control," *Futo Journal Series*, vol. 3, no. 2, 2017. https://www.researchgate.net/profile/Ayodeji-Okubanjo/publication/324531716_Modeling_of_2-DOF_robot_Arm_and_Control/links/5ae029430f7e9b285945fcf9/Modeling-of-2-DOF-robot-Arm-and-Control.pdf

[15] S. C. Chapra and R. P. Canale, *Numerical methods for engineers*, 7th ed. New York NY: McGraw-Hill Education, 2015.

[16] L. Biagiotti and C. Melchiorri, *Trajectory planning for automatic machines and robots*. Berlin and Heidelberg: Springer, 2008.

[17] B. Andreas. Robotics for software engineers: Chapter 6 - kinematics. (2025) Accessed: Mar. 2025. https://livebook.manning.com/book/robotics-for-software-engineers/chapter-6/v-8/13

[18] S. Castro, "Trajectory planing for robotic manipulators," 1 May 2019. https://www.mathworks.com/videos/trajectory-planning-for-robot-manipulators-1556705635398.html

[19] S. M. Lavalle, *Planning Algorithms*. Cambridge University Press, 2006.

[20] P. W. Code. Trajectory planning. (2025) Accessed: Mar. 2025. https://paperswithcode.com/task/trajectory-planning

[21] S. Lavalle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," *Algorithmic and computational robotics: New directions*, 01 2000.

[22] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, DOI 10.1177/0278364911406761, no. 7, pp. 846–894, 2011.

[23] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, DOI 10.1016/j.ijforecast.2006.03.001, no. 4, pp. 679–688, 2006. https://doi.org/10.1016/j.ijforecast.2006.03.001

[24] M. Müller, *Information Retrieval for Music and Motion*. Springer, 2007.

[25] K. Ogata, *Modern control engineering*, 5th ed., ser. Prentice-Hall electrical engineering series. Instrumentation and controls series. Boston: Prentice-Hall, 2010.

[26] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini, *Feedback Control of Dynamic Systems*, 8th ed. Harlow: Pearson, 2020.

[27] Stanislaw H. Zak, "An introduction to model-based predictive control (mpc)," *ECE*, no. 680, 2017.

[28] Z. Zhang *et al.*, "Collision-free trajectory planning of mobile robots by integrating deep reinforcement learning and model predictive control," in *Proceedings of the 2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, DOI 10.1109/CASE56687.2023.10260515, pp. 1–7. IEEE, 2023.

[29] Z. Wu *et al.*, "A tutorial review of machine learning-based model predictive control methods," *Reviews in Chemical Engineering*, DOI 10.1515/revce-2024-0055, 2024.

[30] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2021.

[31] P. Corke, *Robotics, Vision & Control: Fundamental Algorithms in MATLAB®*, 2nd ed., ser. Springer Tracts in Advanced Robotics, vol. 118. Cham, Switzerland: Springer, 2017.

[32] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[33] S. S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1999.

[34] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[35] M. Liaichi. Neural networks and deep learning: A comprehensive introduction. (2020) Accessed: Mar. 2025. https://medium.com/@mustaphaliaichi/neural-networks-and-deep-learning-a-comprehensive-introduction-092449336c1f

[36] F. Chollet, *Deep Learning with Python, Second Edition*. Shelter Island, NY, USA: Manning, 2021.

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[38] C. Olah, "Understanding lstm networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015, accessed: Apr. 2025.

[39] N. Yehia. Understanding long short-term memory networks. (2025) Accessed: Mar. 2025. https://mlarchive.com/deep-learning/understanding-long-short-term-memory-networks/

[40] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "The performance of lstm and bilstm in forecasting time series," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 3285–3292, 2019. https://api.semanticscholar.org/CorpusID:211297310

[41] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, DOI 10.3115/v1/D14-1179, pp. 1724–1734. Doha, Qatar: Association for Computational Linguistics, Oct. 2014. https://aclanthology.org/D14-1179/

[42] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[43] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, DOI 10.1109/ICNN.1996.548916, pp. 347–352 vol.1, 1996.

[44] ResearchGate. Gated recurrent unit (gru). (2025) Accessed: Mar. 2025. https://www.researchgate.net/figure/Gated-Recurrent-Unit-GRU_fig4_328462205

[45] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014. https://arxiv.org/abs/1406.2661

[46] K. S. Egambaram Thirumagal. Generative adversarial network architecture. (2025) Accessed: Mar. 2025. https://www.researchgate.net/figure/Generative-adversarial-network-architecture_fig2_359092115

[47] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. http://incompleteideas.net/book/the-book-2nd.html

[48] J. Murel. What is reinforcement learning? (2025) Accessed: Mar. 2025. https://www.ibm.com/think/topics/reinforcement-learning

[49] M.-A. Chadi and H. Mousannif, "Understanding reinforcement learning algorithms: The progress from basic q-learning to proximal policy optimization," 2023. https://arxiv.org/abs/2304.00026

[50] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2017. https://arxiv.org/abs/1502.05477

[51] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. https://arxiv.org/abs/1707.06347

[52] C. Liu and C. G. Atkeson, "Standing balance control using a trajectory library," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, DOI 10.1109/IROS.2009.5354018, pp. 3031–3036. St. Louis, MO: IEEE, 2009.

[53] C. Vallon and F. Borrelli, "Task decomposition for iterative learning model predictive control," in *2020 American Control Conference (ACC)*, DOI

10.23919/ACC45564.2020.9147625, pp. 2024–2029. Denver, CO, USA: IEEE, 2020.

[54] A. U. Levin and K. S. Narendra, "Control of nonlinear dynamical systems using neural networks: controllability and stabilization," *IEEE transactions on neural networks*, vol. 4, DOI 10.1109/72.207608, no. 2, pp. 192–206, 1993.

[55] Y. Jiang, C. Yang, J. Na, G. Li, Y. Li, and J. Zhong, "A brief review of neural networks based learning and control and their applications for robots," *Complexity*, vol. 2017, DOI 10.1155/2017/1895897, pp. 1–14, 2017.

[56] J. Fei and C. Lu, "Adaptive sliding mode control of dynamic systems using double loop recurrent neural network structure," *IEEE transactions on neural networks and learning systems*, vol. 29, DOI 10.1109/TNNLS.2017.2672998, no. 4, pp. 1275–1286, 2018.

[57] Y.-J. Liu, S. Lu, and S. Tong, "Neural network controller design for an uncertain robot with time-varying output constraint," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, DOI 10.1109/TSMC.2016.2606159, no. 8, pp. 2060–2068, 2017.

[58] M. Wang, H. Ye, and Z. Chen, "Neural learning control of flexible joint manipulator with predefined tracking performance and application to baxter robot," *Complexity*, vol. 2017, DOI 10.1155/2017/7683785, pp. 1–14, 2017.

[59] W. He, Z. Yan, Y. Sun, Y. Ou, and C. Sun, "Neural-learning-based control for a constrained robotic manipulator with flexible joints," *IEEE transactions on neural networks and learning systems*, vol. 29, DOI 10.1109/TNNLS.2018.2803167, no. 12, pp. 5993–6003, 2018.

[60] T. Sun, H. Pei, Y. Pan, H. Zhou, and C. Zhang, "Neural network-based sliding mode adaptive control for robot manipulators," *Neurocomputing*, vol. 74, DOI 10.1016/j.neucom.2011.03.015, no. 14-15, pp. 2377–2384, 2011.

[61] F. L. Lewis, A. Yegildirek, and K. Liu, "Multilayer neural-net robot controller with guaranteed tracking performance," *IEEE transactions on neural networks*, vol. 7, DOI 10.1109/72.485674, no. 2, pp. 388–399, 1996.

[62] I. Ranatunga, S. Cremer, F. L. Lewis, and D. O. Popa, "Neuroadaptive control for safe robots in human environments: A case study," in *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, DOI 10.1109/CoASE.2015.7294099, pp. 322–327. IEEE, 2015.

[63] J. Nicodemus, J. Kneifl, J. Fehr, and B. Unger, "Physics-informed neural networks-based model predictive control for multi-link manipulators," 2021. https://arxiv.org/abs/2109.10793

[64] W. He, D. Ofosu Amoateng, C. Yang, and D. Gong, "Adaptive neural network control of a robotic manipulator with unknown backlash-like hysteresis," *IET Control Theory & Applications*, vol. 11, DOI https://doi.org/10.1049/iet-cta.2016.1058, no. 4, pp. 567–575, 2017. https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cta.2016.1058

[65] K. El Hamidi, M. Mjahed, A. El Kari, and H. Ayad, "Adaptive control using neural networks and approximate models for nonlinear dynamic systems," *Modelling and Simulation in Engineering*, vol. 2020, DOI https://doi.org/10.1155/2020/8642915, no. 1, p. 8642915, 2020. https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/8642915

[66] J. Ho and S. Ermon, "Generative adversarial imitation learning," 2016. https://arxiv.org/abs/1606.03476

[67] Z. Zhou, Y. Lu, S. Kokubu, P. E. Tortós, and W. Yu, "A gan based pid controller for highly adaptive control of a pneumatic-artificial-muscle driven antagonistic joint," *Complex & Intelligent Systems*, vol. 10, DOI 10.1007/s40747-024-01488-y, no. 5, pp. 6231–6248, 2024.

[68] P. Xu and I. Karamouzas, "A gan-like approach for physics-based imitation learning and interactive character control," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 4, DOI 10.1145/3480148, no. 3, pp. 1–22, 2021.

[69] M. Jegorova, J. Smith, M. Mistry, and T. Hospedales, "Adversarial generation of informative trajectories for dynamics system identification," 2020. https://arxiv.org/abs/2003.01190

[70] A. Rostamijavanani, S. Li, and Y. Yang, "Data-driven modeling of parameterized nonlinear fluid dynamical systems with a dynamics-embedded conditional generative adversarial network," 2024. https://arxiv.org/abs/2412.17978

[71] K. Ulrich, S. Eppinger, and M. C. Yang, *Product Design and Development*, 7th ed. McGraw-Hill Education, 2019.

[72] TUM, "Tanmay goyal – scientific employee at the chair of applied mechanics," https://www.mec.ed.tum.de/en/am/staff/employees/scientific-employees/tanmay-goyal/, 2025, accessed: Apr. 2025.

[73] Chair of Applied Mechanics, "Chair of applied mechanics – technical university of munich," https://www.mec.ed.tum.de/am/home/, 2025, accessed: Feb. 2025.

[74] ABB, "Abb robot controller portfolio," https://search.abb.com/library/Download.aspx?DocumentID=9AKK108470A1173&LanguageCode=en&DocumentPartId=&Action=Launch, accessed: Apr. 2025.

[75] Cloudflare, "What's the difference between ai training and inference?" https://www.cloudflare.com/learning/ai/inference-vs-training/, accessed: Apr. 2025.

[76] NVIDIA Blog, "What's the difference between deep learning training and inference?" https://blogs.nvidia.com/blog/difference-deep-learning-training-inference-ai/, accessed: Apr. 2025.

[77] HPCwire, "Infrastructure requirements for ai inference vs. training," https://www.hpcwire.com/2022/06/13/infrastructure-requirements-for-ai-inference-vs-training/, accessed: Apr. 2025.

[78] ByteBridge. Gpu and tpu comparative analysis report. (2025) Accessed: Apr. 2025. https://bytebridge.medium.com/gpu-and-tpu-comparative-analysis-report-a5268e4f0d2a

[79] Google Cloud, "Cloud tpu v4 overview," https://cloud.google.com/tpu/docs/v4, 2024, accessed: Apr. 2025.

[80] T. P. Morgan, "Google stands up exascale tpuv4 pods on the cloud," https://www.nextplatform.com/2022/05/11/google-stands-up-exascale-tpuv4-pods-on-the-cloud/, 2022, accessed: Apr. 2025.

[81] Google Cloud, "Cloud tpu pods break ai training records," https://cloud.google.com/blog/products/ai-machine-learning/cloud-tpu-pods-break-ai-training-records, 2021, accessed: Apr. 2025.

[82] CloudExpat. Aws trainium vs google tpu v5e vs azure nd h100. (2025) Accessed: Apr. 2025. https://www.cloudexpat.com/blog/comparison-aws-trainium-google-tpu-v5e-azure-nd-h100-nvidia/

[83] Google Cloud, "Cloud tpu pricing," https://cloud.google.com/tpu/pricing?hl=en, 2025, accessed: Apr. 2025.

[84] Google Cloud, "Better scalability with cloud tpu pods and tensorflow 2.1," https://cloud.google.com/blog/products/ai-machine-learning/better-scalability-with-cloud-tpu-pods-and-tensorflow-2-1, 2020, accessed: Apr. 2025.

[85] Google Cloud, "Tensor processing units (tpus)," https://cloud.google.com/tpu, 2025, accessed: Apr. 2025.

[86] P. Team, "Pytorch/xla github repository," https://github.com/pytorch/xla, 2025, accessed: Apr. 2025.

[87] NVIDIA Corporation, "Nvidia a100 tensor core gpu," https://www.nvidia.com/en-us/data-center/a100/, 2025, accessed: May 2025.

[88] NVIDIA Developer Blog, "Accelerating inference with sparsity using ampere and tensorrt," https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/, 2020, accessed: May 2025.

[89] S. Ward-Foxton, "Mlperf training scores: Microsoft demonstrates fastest cloud ai," https://www.eetimes.com/mlperf-training-scores-microsoft-demonstrates-fastest-cloud-ai/, 2023, accessed: Apr. 2025.

[90] NVIDIA Developer Blog, "Mlperf v1.0 training benchmarks: Insights into a record-setting performance," https://developer.nvidia.com/blog/mlperf-v1-0-training-benchmarks-insights-into-a-record-setting-performance/, 2021, accessed: Apr. 2025.

[91] DataCamp. Understanding tpus vs gpus in ai: A comprehensive guide. (2024) Accessed: Apr. 2025. https://www.datacamp.com/blog/tpu-vs-gpu-ai

[92] DataCrunch.io, "Cloud gpu pricing comparison: Aws vs gcp vs azure," https://datacrunch.io/blog/cloud-gpu-pricing-comparison, 2025, accessed: Apr. 2025.

[93] Google Cloud, "Spot vms pricing," https://cloud.google.com/spot-vms/pricing?hl=en, 2025, accessed: May 2025.

[94] NVIDIA Corporation, "Deep learning frameworks support matrix," https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html, 2025, accessed: May 2025.

[95] NVIDIA Corporation, "Cuda toolkit documentation," https://developer.nvidia.com/cuda-toolkit, 2025, accessed: May 2025.

[96] NVIDIA Corporation, "Nvidia h100 tensor core gpu overview," https://www.nvidia.com/en-us/data-center/h100/, 2025, accessed: May 2025.

[97] K. Freund, "New mlperf benchmarks show why nvidia has reworked its product roadmap," https://www.forbes.com/sites/karlfreund/2023/11/08/new-mlperf-benchmarks-show-why-nvidia-has-reworked-its-product-roadmap, 2023, accessed: May 2025.

[98] NVIDIA Corporation, "Transformer engine user guide," https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html, 2025, accessed: May 2025.

[99] NVIDIA Corporation, "Fp8 primer: Transformer engine examples," https://docs.nvidia.com, 2025.

[100] Intel Corporation, "App metrics for intel® microprocessors," Intel Corporation, Tech. Rep. Revision 7, Mar. 2025, accessed: Apr. 2025. https://cdrdv2-public.intel.com/841556/APP-for-Intel-Core-Processors.pdf

[101] eunomia.dev, "Os-level challenges in llm inference and optimizations," Feb. 2025, accessed: Apr. 2025. https://eunomia.dev/blog/2025/02/18/os-level-challenges-in-llm-inference-and-optimizations/

[102] Lenovo, "What is thermal design power (tdp)?" https://www.lenovo.com/us/en/glossary/what-is-thermal-design-power/, 2023, accessed: May 2025.

[103] Intel Corporation, "Thermal design power (tdp) in intel® processors," Apr. 2023, accessed: Apr. 2025. https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html

[104] Lanner Inc., "Eai-i730 edge ai appliance," 2024, accessed: Apr. 2025. https://www.lannerinc.com/products/edge-ai-appliance/deep-learning-inference-appliances/eai-i730

[105] Lanner Inc., "Lec-2290b/c/d embedded platform," 2024, accessed: Apr. 2025. https://lannerinc.com/products/intelligent-edge-appliances/embedded-platform/lec-2290b-c-d

[106] OpenBLAS Contributors, "Openblas: An optimized blas library," 2025, accessed: Apr. 2025. https://www.openblas.net/

[107] Open Zeka, "Jetson orin nx module series datasheet," 2022, accessed: Apr. 2025. https://openzeka.com/wp-content/uploads/2022/03/Jetson-Orin-NX-Module-Series-Datasheet.pdf

[108] NVIDIA Corporation, "Jetson orin nx series modules datasheet v1.5," 2023, accessed: Apr. 2025. https://developer.download.nvidia.com/assets/embedded/secure/jetson/orin_nx/docs/Jetson-Orin-NX-Series-Modules-Datasheet_DS-10712-001_v1.5.pdf

[109] M. Amit, "Pytorch for jetson: A comprehensive practical guide for data scientists," 2022, accessed: Apr. 2025. https://mr-amit.medium.com/pytorch-for-jetson-a-comprehensive-practical-guide-for-data-scientists-7d6b2964b97a

[110] Seeed Studio, "Mastering video analytics on nvidia jetson: Your best bet for concurrent and high-throughput solution," 2024, accessed: Apr. 2025. https://www.seeedstudio.com/blog/2024/06/06/mastering-video-analytics-on-nvidia-jetson-your-best-bet-for-concurrent-and-high-throughput-solution/

[111] NVIDIA Corporation, "Jetpack 6.2 brings super mode to jetson orin nano and orin nx modules," 2024, accessed: Apr. 2025. https://developer.nvidia.com/blog/nvidia-jetpack-6-2-brings-super-mode-to-nvidia-jetson-orin-nano-and-jetson-orin-nx-modules/

[112] NVIDIA Corporation, "Installing pytorch for jetson platform," 2024, accessed: Apr. 2025. https://docs.nvidia.com/deeplearning/frameworks/install-pytorch-jetson-platform/index.html

[113] Coral, "Products — coral," 2025, accessed: May 2025. https://coral.ai/products/

[114] Google Coral, "Edge tpu benchmarks," 2023, accessed: Apr. 2025. https://coral.ai/docs/edgetpu/benchmarks/

[115] Arrow Electronics, "Google coral edge tpu accelerator vs intel neural compute stick 2," 2023, accessed: Apr. 2025. https://www.arrow.com/en/research-and-events/articles/google-coral-edge-tpu-accelerator-vs-intel-neural-compute-stick-2

[116] Coral, "Edge tpu performance benchmarks," 2025, accessed: Apr. 2025. https://coral.ai/docs/edgetpu/benchmarks/

[117] Coral, "Model-driven cluster resource management for ai workloads in kubernetes," *ACM Transactions on Computer Systems*, vol. 41, DOI 10.1145/3582080, no. 1, pp. 1–27, 2023.

[118] Coral, "Edge tpu compiler," 2025, accessed: Apr. 2025. https://coral.ai/docs/edgetpu/compiler/

[119] Google Coral, "Edge tpu github issue #156," 2023, accessed: Apr. 2025. https://github.com/google-coral/edgetpu/issues/156

[120] Intel, "Intel core i7-9700te processor," https://www.intel.com/content/www/us/en/products/sku/195329/intel-core-i79700te-processor-12m-cache-up-to-3-80-ghz/specifications.html, accessed: May 2025.

[121] Amazon, "Jetson orin nx 8gb super kit development board kit," https://www.amazon.com/Jetson-Orin-Development-Board-Basis/dp/B0CCV5SW44, accessed: May 2025.

[122] Amazon, "Google coral usb edge tpu ml accelerator," https://www.amazon.com/Google-Coral-Accelerator-coprocessor-Raspberry/dp/B07R53D12W, accessed: May 2025.

[123] D. Zhang and B. Wei, "A review on model reference adaptive control of robotic manipulators," *Annual Reviews in Control*, vol. 43, DOI 10.1016/j.arcontrol.2017.02.002, pp. 188–198, 2017.

[124] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," 2016. https://arxiv.org/abs/1610.00633

[125] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine, "Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation," 2018. https://arxiv.org/abs/1806.10293

[126] E. Johns, "Coarse-to-fine imitation learning: Robot manipulation from a single demonstration," 2021. https://arxiv.org/abs/2105.06411

[127] P. Xu and I. Karamouzas, "A gan-like approach for physics-based imitation learning and interactive character control," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 4, DOI 10.1145/3480148, no. 3, p. 1–22, Sep. 2021. http://dx.doi.org/10.1145/3480148

[128] E. Bøhn, S. Gros, S. Moe, and T. A. Johansen, "Reinforcement learning of the prediction horizon in model predictive control," 2021. https://arxiv.org/abs/2102.11122

[129] MathWorks, "What is model predictive control (mpc)?" https://www.mathworks.com/help/mpc/gs/what-is-mpc.html, 2024, accessed: Mar. 2025.

[130] P. G. A. S. o. C. S. University of Washington and Engineering, "Csep 590a homework 2: Motion planning," https://courses.cs.washington.edu/courses/csep590a/23sp/homeworks/HW2.pdf, 2023, accessed: Mar. 2025.

[131] E. Coumans, "Bullet physics simulation," in *ACM SIGGRAPH 2015 Courses*, 2015.

[132] NVIDIA Corporation, "Nvidia isaac sdk," https://developer.nvidia.com/isaac, 2025, accessed: Apr. 2025.

[133] Open Source Robotics Foundation, "Gazebo sim — open source robotics simulator," https://gazebosim.org/home, accessed: Apr. 2025.

[134] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "Casadi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, DOI 10.1007/s12532-018-0139-4, no. 1, pp. 1–36, 2019.

[135] C. Garrett, "pybullet-planning: Python motion planning toolbox built on pybullet," https://github.com/caelan/pybullet-planning, 2025, accessed: May 2025.

[136] I. A. Şucan, M. Moll, and L. E. Kavraki, "The open motion planning library," in *IEEE International Conference on Robotics and Automation (ICRA)*, DOI 10.1109/ICRA.2012.6225337, pp. 4453–4458, 2012.

[137] S. Chitta, I. Sucan, and S. Cousins, "Moveit! [ros topics]," in *IEEE Robotics & Automation Magazine*, vol. 19, DOI 10.1109/MRA.2011.2181749, no. 1, pp. 18–19, 2012.

[138] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *2009 IEEE International Conference on Robotics and Automation (ICRA)*, DOI 10.1109/ROBOT.2009.5152817, pp. 489–494. IEEE, 2009.

[139] ROS-Industrial Consortium, "Optimization motion planning with tesseract and trajopt for industrial applications," https://rosindustrial.org/news/2018/7/5/optimization-motion-planning-with-tesseract-and-trajopt-for-industrial-applications, 2018, accessed: Apr. 2025.

[140] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge, UK: Cambridge University Press, 1995.

[141] A. Wächter, "An interior point algorithm for large-scale nonlinear optimization with applications in process engineering," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, Jan. 2002. https://users.iems.northwestern.edu/~andreasw/pubs/waechter_thesis.pdf

[142] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. http://jmlr.org/papers/v22/20-1364.html

[143] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo, "Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022. http://jmlr.org/papers/v23/21-1342.html

[144] Z. Wu, E. Liang, M. Luo, S. Mika, J. E. Gonzalez, and I. Stoica, "RLlib flow: Distributed reinforcement learning is a dataflow problem," in *Conference on Neural Information Processing Systems (NeurIPS)*, 2021. https://proceedings.neurips.cc/paper/2021/file/2bce32ed409f5ebcee2a7b417ad9beed-Paper.pdf

[145] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[146] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *Neural networks: Tricks of the trade*, pp. 437–478, 2012.

[147] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021.

[148] M. Feurer and F. Hutter, "Hyperparameter optimization," in *Automated Machine Learning*, pp. 3–33. Springer, 2019.

[149] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

[150] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2013.

[151] T. Head *et al.*, "Scikit-optimize: Sequential model-based optimization in python," https://scikit-optimize.github.io/, 2018.

[152] J. Rapin and O. Teytaud, "Nevergrad: An optimization platform," https://facebookresearch.github.io/nevergrad/, 2018.

[153] C. Wang, X. Shi, Q. Fu, D. Wang, Y. Zhu, L. Zhang, J. Chen, S. Liu, Y. Chen, M. Liu *et al.*, "Flaml: A fast and lightweight automl library," in *Proceedings of the 2021 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 3468–3476, 2021.

[154] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," in *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[155] European Commission, "Industrial leadership in ai, data and robotics – advanced human-robot interaction (ai data and robotics partnership) (ia)," https://www.horizon-europe.gouv.fr/industrial-leadership-ai-data-and-robotics-advanced-human-robot-interaction-ai-data-and-robotics, accessed: Apr. 2025.

[156] European Commission, "Cluster 4: Digital, industry and space – horizon europe," https://research-and-innovation.ec.europa.eu/funding/funding-opportunities/funding-programmes-and-open-calls/horizon-europe/cluster-4-digital-industry-and-space_en, accessed: Apr. 2025.

[157] European Commission, "Digital technologies and research," https://digital-strategy.ec.europa.eu/en/activities/digital-technologies-and-research, accessed: Apr. 2025.

[158] European Research Council, "Apply for an erc advanced grant," https://erc.europa.eu/apply-grant/advanced-grant, accessed: Apr. 2025.

[159] German Research Foundation (DFG), "What is the dfg?" https://www.dfg.de/en/about-us/about-the-dfg/what-is-the-dfg, accessed: Apr. 2025.

[160] German Research Foundation (DFG), "Priority programmes (spp)," https://www.dfg.de/en/research-funding/funding-opportunities/programmes/coordinated-programmes/priority-programmes, accessed: Apr. 2025.

[161] German Research Foundation (DFG), "Collaborative research centres (sfb)," https://www.dfg.de/en/research-funding/funding-opportunities/programmes/coordinated-programmes/collaborative-research-centres, accessed: Apr. 2025.

[162] German Research Foundation (DFG), "New priority programme "soft material robotic systems" (spp 2100)," https://www.dfg.de/de/aktuelles/neuigkeiten-themen/info-wissenschaft/2021/info-wissenschaft-21-39, accessed: Apr. 2025.

[163] Federal Ministry of Education and Research (BMBF), "Robotics research – bmbf," https://www.bmbf.de/DE/Forschung/Zukunftstechnologien/Robotik/robotik_node.html, accessed: Apr. 2025.

[164] University of Bonn, "Bmbf funds "robots in everyday life" transfer center," https://www.uni-bonn.de/en/news/bmbf-funds-201crobots-in-everyday-life201d-transfer-center, accessed: Apr. 2025.

[165] Federal Ministry of Education and Research (BMBF), "Announcement: Robotics research action plan 2023," https://www.bmbf.de/SharedDocs/Bekanntmachungen/DE/2023/11/2023-11-23-Bekanntmachung-Robotik.html, accessed: Apr. 2025.

[166] Bavarian Research Foundation (BayFOR), "New research association forinfpro – €2 million for ai-based manufacturing," https://www.bayfor.org/en/news/latest-news/news-detail/4709-new-research-association-forinfpro-2-million-euros-for-the-development-of-ai-based-and-self-adapting-manufacturing-processes.html, accessed: Apr. 2025.

[167] FORTIS Project, "Open call analysis: Fortis boosting human-robot interaction," https://cascadefunding.eu/analysis-fortis-1st-open-call/, accessed: Apr. 2025.

[168] SWEET Project, "Social awareness for service robots (sweet)," https://www.sweet.unina.it/, accessed: Apr. 2025.

[169] RoboDK, "Robodk pricing," https://robodk.com/pricing, accessed: Apr. 2025.

[170] NVIDIA, "Nvidia omniverse enterprise," https://resources.nvidia.com/en-us-omniverse-enterprise/nvidia-omniverse-ent, accessed: Apr. 2025.

[171] Unchained Robotics, "Abb gofa crb 15000 – cobot price and specs," https://unchainedrobotics.de/en/products/robot/cobot/abb-gofa-crb-15000, accessed: Apr. 2025.

[172] academics.com, "Postdoc salary in germany," https://www.academics.com/guide/postdoc-salary-germany, accessed: Apr. 2025.

[173] Academic Positions, "Phd, postdoc and professor salaries in germany," https://academicpositions.com/career-advice/phd-postdoc-and-professor-salaries-in-germany, accessed: Apr. 2025.

[174] Fastepo, "Phd and postdocs salary in germany," https://fastepo.com/phd-and-postdocs-salary/phd-students-and-postdocs-salary-germany/, accessed: Apr. 2025.

[175] Lohntastik, "Tv-l e13 salary calculator," https://lohntastik.de/od-rechner/tv-salary-calculator/TV-L/E-13/1, accessed: Apr. 2025.

[176] Grand View Research, "Collaborative robot market size, share and trends analysis report," https://www.grandviewresearch.com/industry-analysis/collaborative-robots-market, accessed: Apr. 2025.

[177] GlobeNewswire, "Collaborative robot market size, share, industry growth analysis 2029," https://www.globenewswire.com/news-release/2024/05/01/2872879/0/en/Collaborative-Robot-Market-Size-Share-Industry-Growth-Analysis-2029.html, accessed: Apr. 2025.

[178] Grand View Research, "Europe collaborative robot market outlook," https://www.grandviewresearch.com/horizon/outlook/collaborative-robot-market/europe, accessed: Apr. 2025.

[179] International Federation of Robotics, "World robotics press conference 2024," https://ifr.org/img/worldrobotics/Press_Conference_2024.pdf, accessed: Apr. 2025.

[180] EPA HERO, "Autonomy and parameter tuning in robotics," https://hero.epa.gov/hero/index.cfm/reference/details/reference_id/7081907/#:~:text=Autonomy%20is%20increasingly%20demanded, accessed: Apr. 2025.

[181] IndustryWeek, "It's time for flexibility to become a robotic norm," https://www.industryweek.com/technology-and-iiot/article/21136264/its-time-for-flexibility-to-become-a-robotic-norm, accessed: Apr. 2025.

[182] Devonics, "Cobot faqs – setup and programming time," https://www.devonics.com/cobot-faqs#:~:text=How%20long%20does%20it%20take, accessed: Apr. 2025.

[183] Universal Robots, "Case study – stamit sro," https://www.universal-robots.com/case-stories/stamit-sro//#:~:text=When%20preparing%20for%20the%20robot, accessed: Apr. 2025.

[184] Cutting Tool Engineering Magazine, "Automated payback," https://www.ctemag.com/articles/automated-payback#:~:text=Tuohy%20noted%20that%20a%20Fanuc, accessed: Apr. 2025.

[185] L. F. S. Oliveira et al., "Cobots in smes: Implementation processes, challenges and success factors," https://www.researchgate.net/publication/379622985_Cobots_in_SMEs_Implementation_Processes_Challenges_and_Success_Factors, accessed: Apr. 2025.

[186] Global Growth Insights, "Collaborative robots market – trends and insights," https://www.globalgrowthinsights.com/market-reports/collaborative-robots-market-100872#:~:text=Despite%20the%20increasing%20advantages, accessed: Apr. 2025.

[187] Global Growth Insights, "Assistive robotics market – challenges and forecasts," https://www.globalgrowthinsights.com/market-reports/assistive-robotics-market-114163#:~:text=%2A%20Challenges%20%E2%80%93%2044,and, accessed: Apr. 2025.

[188] Robotics4EU, "Agrifood robotics – preview report," https://robotics4eu.eu/wp-content/uploads/2023/12/Report_Preview_Agrifood_Robotics4eu_jr_20240112-1.pd, accessed: Apr. 2025.

[189] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[190] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

Appendix A

Appendix

A.1 Hyperparameter Tuning 1

This section presents the results of the initial hyperparameter tuning study. The defined search space is summarized in Table A.1.

Hyperparameter	**Type of Iteration**	**Values or Range**
Entropy Coefficient	Uniform	[0.001, 0.01]
Number of Steps	Choice	500, 1000 or 2000
Number of Minibatches	Choice	16, 32 or 64
Hidden Dimension	Choice	32, 64, 96 or 128
Discriminator Heads	Choice	8, 12, 16, 20 or 24
λ_{GP}	Uniform	[5.0, 25.0]
Batch Size of Discriminator	Choice	2048, 4096, 8192, 16384, 32768 or 65536
Standard Deviation Initializer	Uniform	[3.0, 6.0]
Future Horizon (M)	Choice	2, 3, 4 or 5
Past Horizon (N)	Choice	7, 8, 9, 10, 11, 12 or 13

Table A.1: Search space for the first iteration of hyperparameter tuning.

Figure A.1 shows that lower entropy coefficient values are associated with lower loss values. Similarly, Figure A.2 indicates that lower initial standard deviation values tend to produce lower losses.

In Figure A.3, a hidden dimension of 96 results in consistently lower losses, while the lowest loss overall occurs at 128.

Figure A.4 illustrates that higher values of λ_{GP} correspond to lower loss values. The best

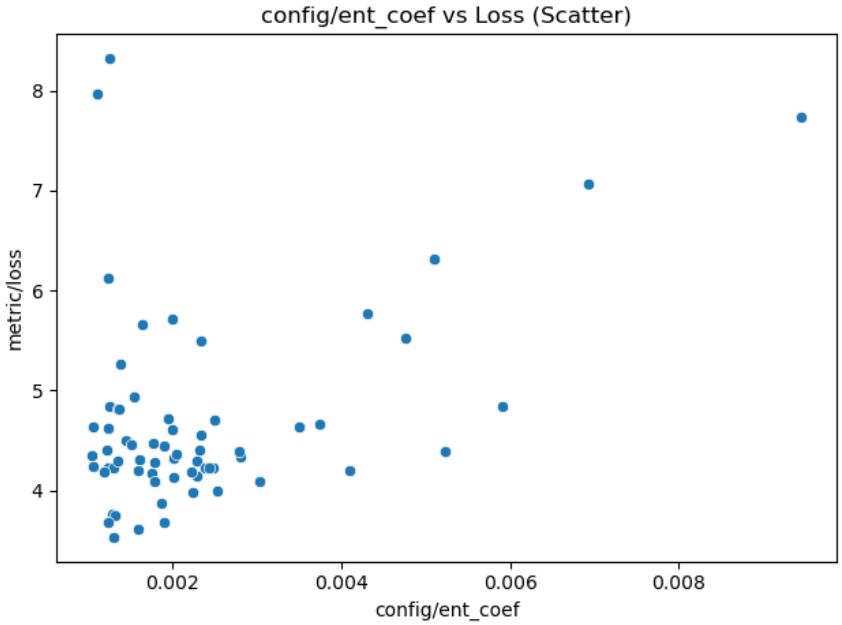


Figure A.1: Scatter plot of the entropy coefficient (*ent_coef*) versus the loss.

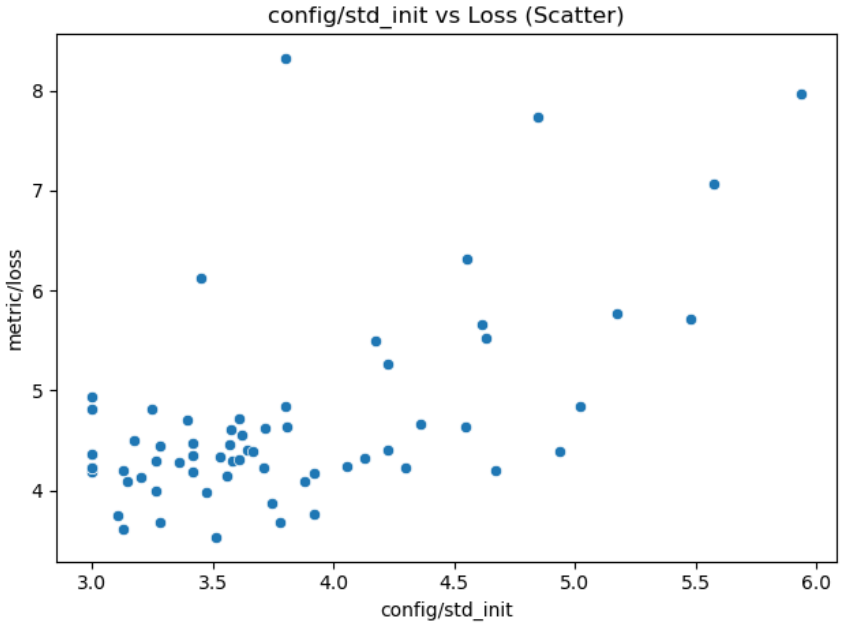


Figure A.2: Scatter plot of the initial standard deviation (*std_init*) versus the loss.

performance in this range is obtained with $\lambda_{GP} = 25$.

Figures A.5 and A.6 show that prediction horizon values of 11 and 12 correspond to lower losses. Although some outliers are observed, both values exhibit favorable behavior across samples.

Figure A.7 shows that history window sizes of 4 and 5 yield the lowest losses, with 5 being more consistent across trials.

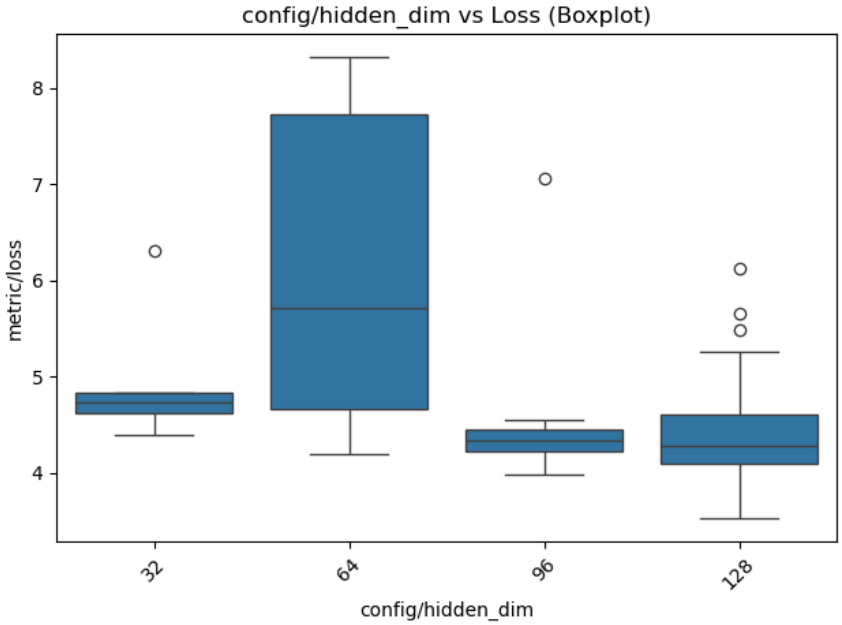


Figure A.3: Boxplot of the hidden dimension size (*hidden_dim*) versus the loss.

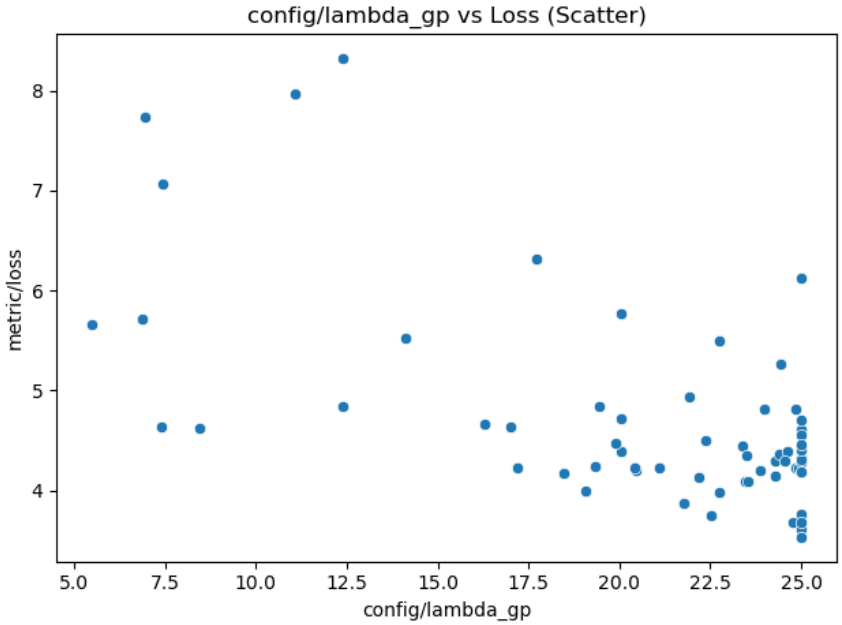


Figure A.4: Scatter plot of the gradient penalty coefficient (*lambda_gp*) versus the loss.

In Figure A.8, lower numbers of attention heads in the discriminator are associated with reduced loss values. The values 8 and 12 produce the best results.

Figure A.9 compares different rollout lengths. The configuration with 2048 steps achieves the lowest loss in this set of experiments.

Lastly, Figure A.10 indicates that a discriminator batch size of 4096 produces the most stable low-loss results across trials.

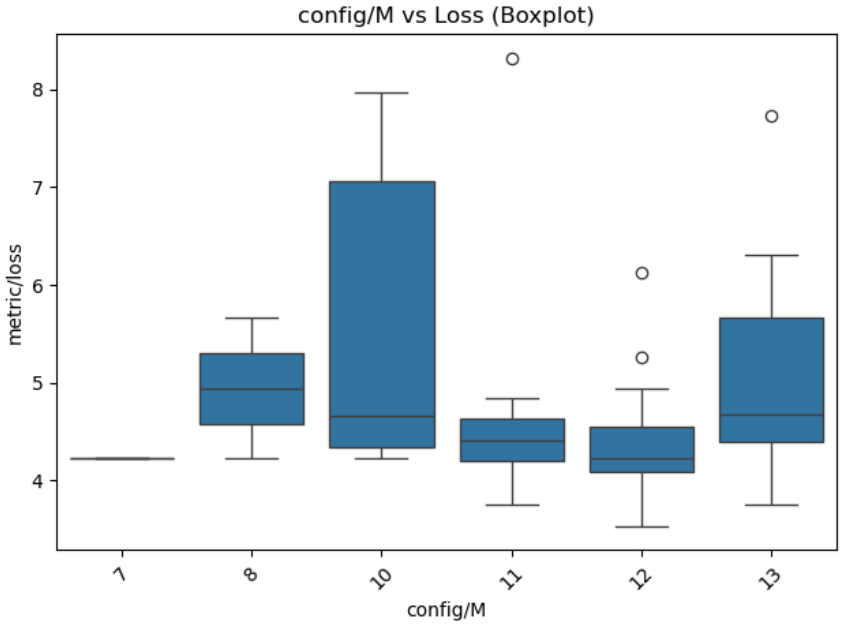


Figure A.5: Boxplot of the prediction horizon parameter (config/M) versus the loss.

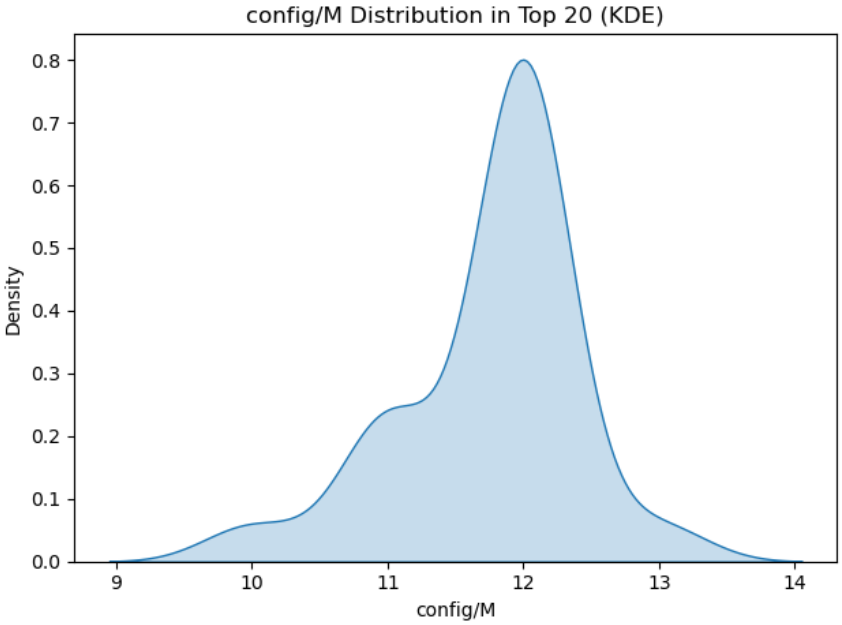


Figure A.6: Kernel density estimation of the top 20 values for the prediction horizon parameter (config/M).

A.2 FLOPs Estimation

To estimate the computational requirements for real-time inference, it is needed to calculate the number of floating-point operations (FLOPs) required to evaluate a forward pass of the neural network.

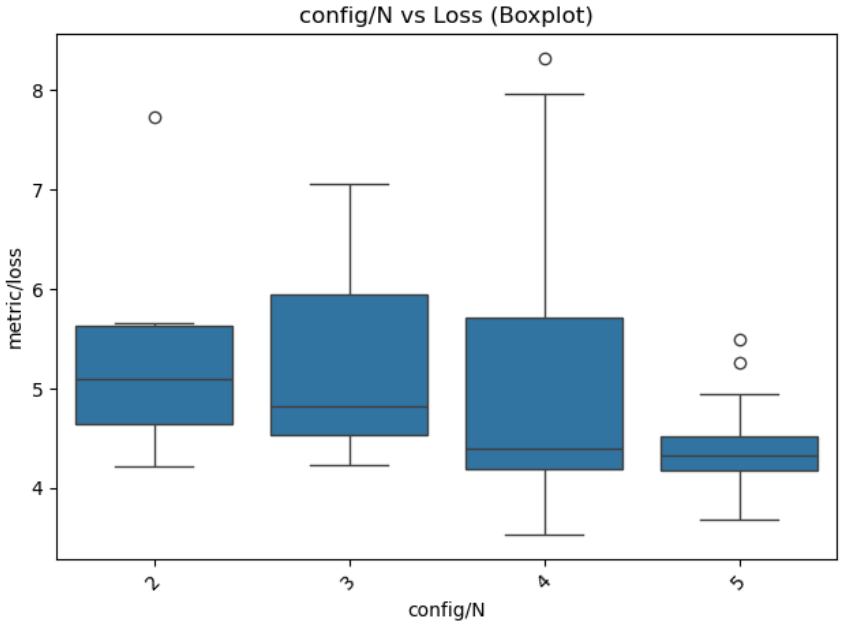


Figure A.7: Boxplot of the history window length (N) versus the loss.

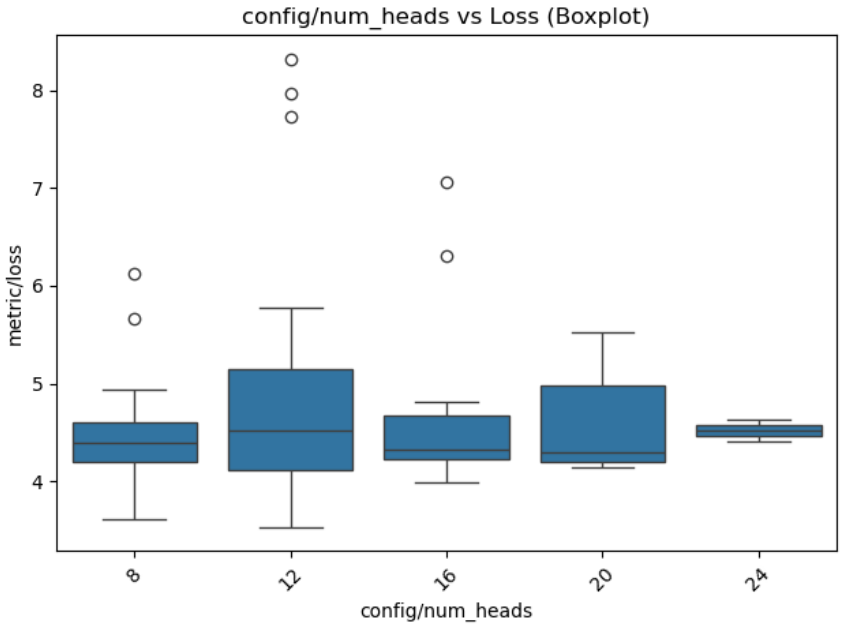


Figure A.8: Boxplot of number of heads of the discriminator (*num_heads*) versus loss.

GRU Layer FLOPs

The GRU cell performs the operations described from A.1 to A.4 per timestep [189].

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{A.1}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{A.2}$$

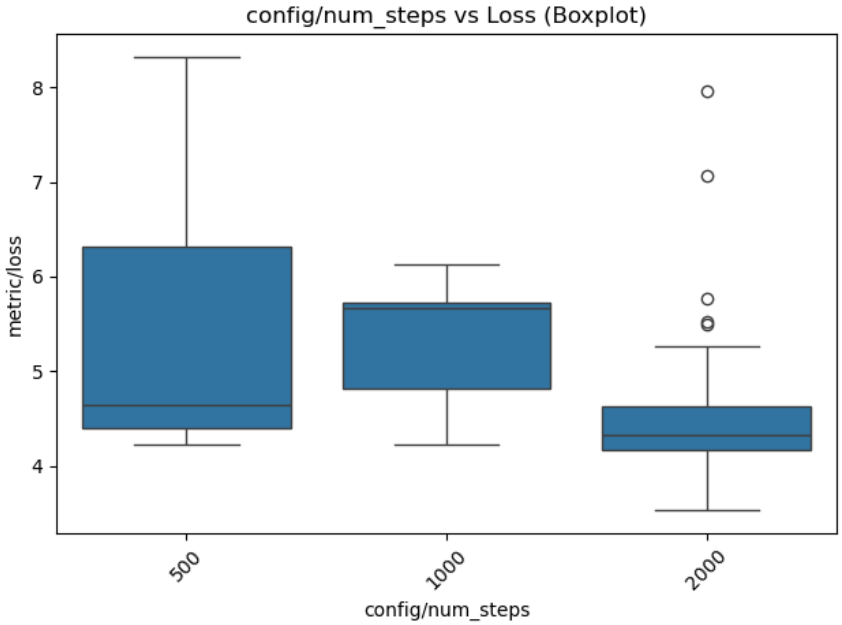


Figure A.9: Boxplot of the number of steps per rollout (*num_steps*) versus the loss.

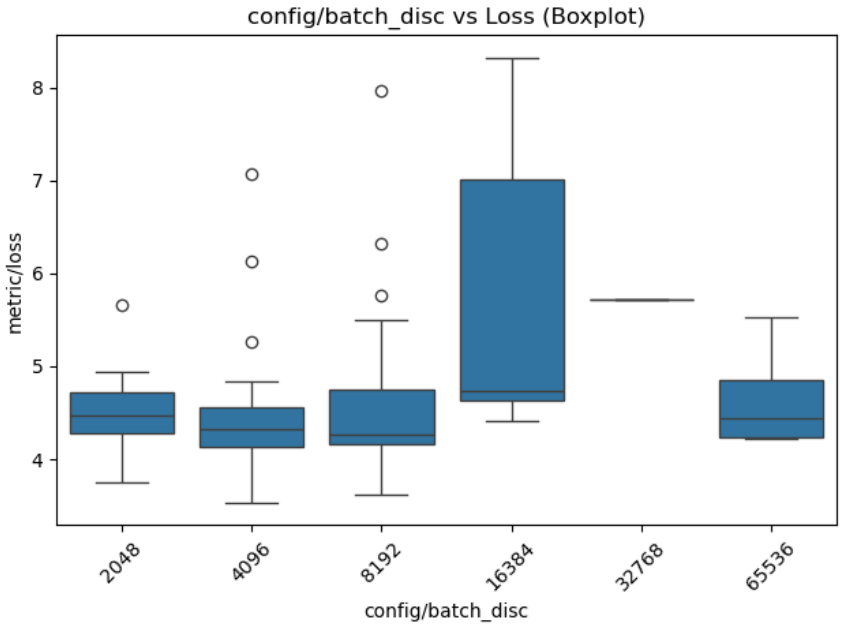


Figure A.10: Boxplot of the discriminator batch size (*batch_disc*) versus the loss.

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h) \tag{A.3}$$

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t \tag{A.4}$$

Each gate requires two matrix multiplications: one for the input $x_t \in \mathbb{R}^I$ and one for the previous hidden state $h_{t-1} \in \mathbb{R}^H$. Each matrix multiplication of size $m \times n$ contributes approximately $2mn$ FLOPs (1 multiplication and 1 addition per output element). Thus, the total cost of a GRU cell per timestep is approximately described in equation A.5

$$\text{FLOPs}_{\text{GRU}} \approx 3 \cdot 2 \cdot (I \cdot H + H \cdot H) = 6H(I + H) \tag{A.5}$$

This estimate excludes element-wise nonlinearities (sigmoid, tanh) as their cost is negligible relative to matrix multiplications.

Dense Layer FLOPs

For a fully connected layer with input dimension n_{in} and output dimension n_{out}, the number of FLOPs [190] is shown in equation A.6

$$\text{FLOPs}_{\text{Dense}} = 2 \cdot n_{\text{in}} \cdot n_{\text{out}} \tag{A.6}$$

The total FLOPs for a network with L such layers is the sum over all shown in A.7

$$\text{FLOPs}_{\text{FC}} = 2 \sum_{l=1}^{L} n_{\text{in}}^{(l)} \cdot n_{\text{out}}^{(l)} \tag{A.7}$$

Total Inference Cost

The total FLOPs per inference is the sum of the GRU and dense layer contributions, the final formulation is represented in A.8.

$$\text{FLOPs}_{\text{Total}} = 6H(I + H) + 2 \sum_{l=1}^{L} n_{\text{in}}^{(l)} \cdot n_{\text{out}}^{(l)} \tag{A.8}$$

Index

Abstract, e

Conclusions and Recommendations, 113

Design of a Control System for Execution of
Trajectories in Robots, 33

Introduction, 1

Objectives, 2

Results, 95
Resumen, g

Theoretical Framework, 5