

Trajectory Tracking Control of Robotic Manipulator using Deep Reinforcement Learning

Submitted in partial fulfilment of the requirements

of the degree of

Bachelor of Technology

by

**Anay Aggarwal (19115023), Anurag Srivastava (19117016) and
Sachin Agrawal (19115108)**

Supervisor

Prof. Sohom Chakrabarty






Department of Electrical Engineering
Indian Institute of Technology Roorkee

2022

Declaration

I hereby declare that the work which is presented here, entitled **Trajectory Tracking Control of Robotic Manipulator using Deep Reinforcement Learning**, submitted in partial fulfilment of the requirements for the award of the Degree of **Bachelor of Technology** in the Department of Electrical Engineering, Indian Institute of Technology Roorkee. I also declare that I have been doing my work from August, 2022 under the supervision and guidance of **Prof. Sohom Chakrabarty, Electrical Engineering Dept., Indian Institute of Technology Roorkee**. The matter presented in this dissertation report has not been submitted by me for award of any other degree of institute or any other institutes.

Date: 25/04/2023

Name	Enrolment No.	Signature
Anay Aggarwal	19115023	
Anurag Srivastava	19117016	
Sachin Agrawal	19115108	

Certificate

This is to certify that the above statement made by the candidate is true to best of my knowledge and belief.


Digitally signed by Sohom Chakrabarty
DN: cn=Sohom Chakrabarty, o=IIT Roorkee, ou=Electrical, email=s.sohom@gmail.com, c=IN
Date: 2023.04.26 11:04:58 +05'30'

Signature

Sohom Chakrabarty

Assistant Professor

Department of Electrical Engineering

Indian Institute of Technology Roorkee

Acknowledgement

We would like to extend our heartfelt gratitude to our project supervisor, Prof. Sohom Chakrabarty of the Electrical Engineering department at the Indian Institute of Technology, Roorkee. His invaluable guidance, support, and encouragement have not only given us the opportunity to work on this project but have also significantly contributed to our professional growth.

Furthermore, we express our sincere appreciation to Mr. Ashish Shakya, whose consistent guidance and expertise have been indispensable in addressing our concerns and advancing the project.

We are deeply grateful to the course coordinators, Prof. Abhisek K. Behera and Prof. Avik Bhattacharya, for their unwavering encouragement and support throughout our academic journey.

The training for this project was carried out at the Machine Learning Lab within the Department of Electrical Engineering at IIT Roorkee. We are thankful to the lab and its staff for generously providing us with the resources needed to successfully complete our work.

Lastly, we would like to acknowledge the collective efforts and contributions of the teaching faculty within the Department of Electrical Engineering. Their knowledge, ideas, and dedication have been instrumental in guiding us toward the completion of this project.

Abstract

Robotic manipulators play a crucial role in automating various industrial tasks that would otherwise require human intervention. In this project, we focus on developing a model-free deep reinforcement learning (RL) approach to control the trajectories of such robots. We have employed a 6-degree of freedom (DOF) robotic arm for this purpose and created suitable state and action representations. We utilized the Deep Deterministic Policy Gradient (DDPG) method, a highly adaptable deep RL technique, to train our RL agent. Initially, we applied the DDPG method to a 2D model of the robotic manipulator and achieved impressive accuracy and smoothness in tracking various 2D paths. After this success, we developed a 3D model using the Pybullet library and trained the same DDPG agent on it. The 3D model was trained using Python 3.8, using the TensorFlow and Pybullet libraries. Following the training, the 3D simulation demonstrated the ability to track complex 3D trajectories, such as helix, rectangle, crown, and more. We conducted extensive testing with numerous reward functions and neural network designs, ultimately arriving at an RL agent that enables our robotic arm to smoothly and swiftly track any given path. A significant advantage of our approach is its model-free nature, which contrasts with traditional model-based trajectory control methods that necessitate extensive calculations.

Table of Contents

Acknowledgement	iii
Abstract	iv
List of Figures	vii
List of Tables	vii
1 Introduction	8
1.1 Robotic Manipulator	8
1.2 Deep Reinforcement Learning	9
1.3 Components of Deep RL	11
1.4 Objectives	12
1.5 Related Work	13
2 Methodology	15
2.1 Deep RL Method Used	16
2.2 Simulation	17
2.2.1 2D Simulation using Pyglet	17
2.2.2 3D Simulation using Pybullet	18
2.2.3 Why did we do 2D simulation before 3D	19
2.3 Actor and Critic Network	20
2.4 Action Space	21
2.4.1 Action Space for 2D Model	22
2.4.2 Action Space for 3D Model	22
2.5 State Space	22
2.5.1 State Space for 2D Model	23
2.5.2 State Space for 3D Model	23
2.6 Reward Function	24
2.6.1 Reward Function for 2D Model	24
2.6.2 Reward Function for 3D Model (Old Version)	25
2.6.3 Reward Function for 3D Model (New Version)	26
2.7 Training Algorithm	27
2.8 Learning Process of DRL Agent	28

3 Result and Analysis	30
3.1 Training Details	30
3.2 Hyperparameters	30
3.3 Evaluation Metrics	32
3.3.1 Average reward of an episode	33
3.3.2 Accuracy of the last 100 episodes	33
3.3.3 Average Q value of an episode	33
3.3.4 Length (no. of steps) of an episode	33
3.4 Accuracy	34
3.5 Reward	35
3.6 Q Value	36
3.7 Episode Length	36
3.8 2D Trajectory Tracking	37
3.8.1 Semicircle	37
3.8.2 Rectangle	38
3.9 3D Trajectory Tracking	38
3.9.1 Crown	39
3.9.2 Rectangle	41
3.9.3 Helix	43
3.10 Analysis	45
4 Conclusion	46
4.1 Challenges	47
4.2 Future Work	48
References	49

List of Figures

1. An image of a 6 DOF robotic arm	8
2. Joints in a 6-DOF Robotic Arm	9
3. Working of Deep RL Methods	10
4. Overview of DDPG Algorithm	16
5. 2D Simulation of robotic manipulator in Pyglet	18
6. 3D Simulation of robotic manipulator in Pybullet	19
7. The Architecture of Actor Network	21
8. The Architecture of Critic Network	21
9. Variation of r_d with distance	26
10. Accuracy vs No. of episodes	34
11. Reward vs No. of episodes	35
12. Q Value vs No. of episodes	36
13. Episode Length vs No. of episodes	36
14. 2D Trajectory tracking of a semicircle	37
15. 2D Trajectory tracking of a rectangle	38
16. Tracking of a crown-shaped trajectory	39
17. Actual Trajectory (left) and tracked trajectory (right) for crown-shaped trajectory	40
18. Analysis of trajectory tracking of crown in different axes	40
19. Tracking of a rectangular trajectory	41
20. Actual Trajectory (left) and tracked trajectory (right) for rectangular trajectory	42
21. Analysis of trajectory tracking of rectangular in different axes	42
22. Tracking of a helical trajectory	43
23. Actual Trajectory (left) and tracked trajectory (right) for helical trajectory	44
24. Analysis of trajectory tracking of helical in different axes	44

List of Tables

1. Values of hyperparameters used during training	31
2. Accuracy analysis of different models	45

Chapter 1

Introduction

Robotic manipulators, commonly referred to as robotic arms, play a crucial role in modern industries, particularly as the world continues to embrace automation. These versatile machines can be employed to solve trajectory tracking problems [1], enabling them to move from one point to another in space or follow a specific, pre-determined trajectory, such as a circular path [2]. Given the substantial investment required for these advanced machines, it is imperative that any algorithms used to guide their motion do not inflict damage, such as causing sudden jerks or collisions with other objects. To prevent such issues, appropriate safeguards must be incorporated into the algorithms or methods utilized [3].

In the past, control-based methods have been the primary means of managing robotic manipulators [4]. These methods, while sophisticated and operating within closed-loop systems, have their limitations. One major drawback is that they necessitate comprehensive information about the robot and its surroundings, in addition to being tailored specifically for each application [4]. This is where deep reinforcement learning (DRL) offers a compelling alternative [5].

In this report, we will explore and employ deep reinforcement learning techniques to train a robotic manipulator, with the aim of achieving rapid, smooth, and secure motion. By leveraging the power of DRL, we seek to overcome the limitations of traditional control-based methods and develop a more adaptable and efficient approach for controlling robotic arms in various industrial settings.



Figure 1: An image of a 6 DOF robotic arm

1.1 Robotic Manipulator

In this project, we are employing a 6-degrees of freedom (DOF) robotic manipulator [6] consisting of three articulated arms, a 360-degree rotating base, and an end effector designed for grasping objects. This configuration allows for six degrees of motion, including translation in the x, y, and z axes, as well as rotation around the pitch, yaw, and roll axes. The manipulator is equipped with six servo motors, which control the movement of the individual joints and the closure of the end effector.

Our objective is to leverage deep reinforcement learning (DRL) techniques to control the trajectory of this robotic arm, ensuring that it can smoothly execute a range of trajectories, such as circular paths, once the DRL agent has been adequately trained. By doing so, we aim to enhance the robot's adaptability and efficiency in diverse applications, surpassing the performance of traditional control-based methods.

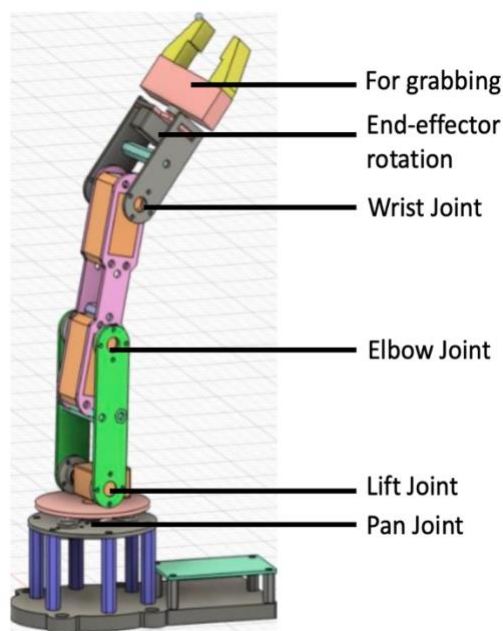


Figure 2: Joints in a 6-DOF Robotic Arm

1.2 Deep Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on enabling an agent to make decisions based on its current state to interact with its environment. The agent receives a reward for each action it takes and uses this information to update its knowledge, with the goal of maximizing cumulative future rewards. Deep Reinforcement Learning (Deep RL) [7] combines RL with deep learning techniques, such as artificial neural networks, which can be utilized as an agent's policy to guide its decision-making process.

Deep RL has rapidly evolved as a research area within machine learning, gaining widespread attention after Google's DeepMind successfully employed it to play Atari games [8]. Today, Deep RL is extensively applied in the field of robotics, particularly in model-free approaches. Model-free methods do not require comprehensive information about the robot or its environment; instead, they only need the state space and the reward to determine an action space. This makes Deep RL techniques much more accessible and straightforward to deploy compared to traditional control methods.

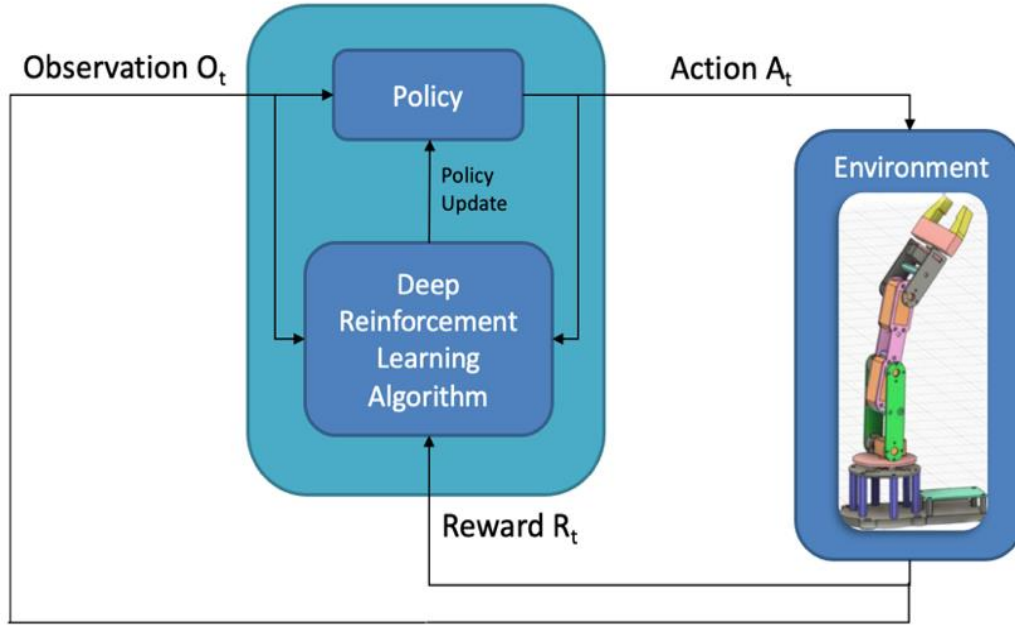


Figure 3: Working of Deep RL methods.

Deep RL methods can be broadly categorized into two groups based on their primary focus: immediate rewards or value. Policy-based methods prioritize immediate rewards associated with subsequent actions, while value-based methods emphasize the total expected future rewards. Some of the popular DRL algorithms used in this context include:

- **Deep Q-Networks (DQN):** DQN combines Q-learning with deep neural networks to approximate the action-value function, enabling the algorithm to handle high-dimensional state spaces. DQN has been used to control robotic manipulators in various tasks, including grasping, trajectory planning, and obstacle avoidance.

$$Q(s, a) = \sum_{s'} P_{ss'} [R_{ss'} + \gamma \sum_{a'} \pi(s', a') Q(s', a')]$$

The Bellman Equation used to calculate Q values in DQN.

- **Proximal Policy Optimization (PPO):** PPO is a policy gradient method that uses a surrogate objective function with a clipped probability ratio, ensuring the updated policy remains close to the old policy while still allowing for optimization. PPO has been applied to control robotic manipulators for tasks like reaching, grasping, and manipulation.
- **Soft Actor-Critic (SAC):** SAC is an off-policy, model-free algorithm that incorporates entropy regularization into the policy optimization process. This encourages exploration

and results in more robust policies. SAC has been used to control robotic manipulators for tasks like in-hand manipulation, dexterous grasping, and assembly.

- **Deep Deterministic Policy Gradient (DDPG):** DDPG is an off-policy, model-free algorithm that uses deep neural networks. DDPG is particularly suitable for continuous action spaces and has been used for controlling robotic manipulators in tasks like trajectory tracking and force control.

These deep reinforcement learning methods have shown promising results in controlling robotic manipulators with stochastic action and state spaces, enabling them to perform complex tasks efficiently and adaptively. By understanding these methods and their respective strengths and weaknesses, we can tailor our approach to training the robotic manipulator to achieve smooth, efficient, and safe trajectory execution.

1.3 Components of Deep RL

Deep Reinforcement Learning (Deep RL) methods combine the concepts of reinforcement learning and deep learning, utilizing artificial neural networks to approximate complex functions and policies. There are several key components in a Deep RL method that work together to enable an agent to learn how to interact effectively with its environment:

- **State Space:** The state space is the representation of the environment and the agent's position within it. It comprises all the possible states that an agent may encounter during its interaction with the environment. In Deep RL, the state space is often represented by high-dimensional data, such as images or sensor readings, which are used as input to the neural network.
- **Action Space:** The action space defines the set of possible actions the agent can perform at each state. In Deep RL, the action space can be discrete or continuous, depending on the problem being solved. The neural network's output layer usually represents the action space, providing either a probability distribution over discrete actions or a set of continuous values.
- **Policy:** The policy is a function that maps the current state to an action, guiding the agent's decision-making process. In Deep RL, the policy is often represented by a neural network that takes the current state as input and outputs an action or probability distribution over actions.
- **Reward Function:** The reward function evaluates the agent's actions by providing immediate feedback in the form of a scalar value. The objective of the agent is to

maximize the cumulative reward over time. The reward function plays a crucial role in shaping the agent's behavior and guiding its learning process.

- **Value Function:** The value function estimates the expected cumulative reward the agent will receive from a given state, considering its current policy. In Deep RL, value functions are often approximated using neural networks, helping the agent to evaluate the long-term consequences of its actions and make more informed decisions.
- **Exploration vs. Exploitation:** A critical aspect of reinforcement learning is balancing exploration (trying new actions) and exploitation (using the learned knowledge to make the best decisions). In Deep RL, various strategies can be employed to strike this balance, such as epsilon-greedy exploration, Boltzmann exploration, or using intrinsic motivation.
- **Learning Algorithm:** The learning algorithm updates the neural network's parameters based on the agent's experiences, aiming to improve its policy and value function. Some popular Deep RL algorithms include Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and Deep Deterministic Policy Gradient (DDPG), among others.
- **Replay Buffer:** In many Deep RL methods, the agent stores its experiences in a memory buffer called the replay buffer. By sampling experiences from this buffer to update the neural network, the agent can break correlations between consecutive experiences and improve the stability and efficiency of the learning process.

These components work together to create a Deep RL method capable of learning complex decision-making strategies for a wide range of problems. By leveraging the power of deep learning, Deep RL methods have achieved remarkable success in various domains, including robotics, control, and game playing.

1.4 Objectives

- **Develop a deep reinforcement learning (DRL) framework for robotic manipulators:** Design and implement a DRL-based control system that enables robotic manipulators to learn and adapt in real-time while performing tasks in cluttered and dynamic environments.
- **Design an appropriate reward function:** Develop a suitable reward function that encourages the DRL agent to achieve its goals effectively and efficiently, considering various factors such as accuracy, speed, etc.

- **Optimize the learning process:** Investigate and apply various techniques for improving the learning process, such as hyperparameter tuning, exploration-exploitation trade-offs, and network architecture design, to ensure the DRL agent learns effectively and converges to an optimal policy.
- **Evaluate the performance of the DRL-based control system:** Test the performance of the developed DRL agent on a range of predefined 3D trajectories, measuring various metrics such as accuracy to assess the effectiveness of the proposed approach.
- **Implement the DRL-based control system in a simulated environment:** Integrate the developed DRL agent into a simulation environment, such as Pybullet, to enable the testing and evaluation of the system in a controlled and realistic setting.
- **Investigate the scalability and generalization of the DRL-based control system:** Examine the ability of the developed DRL agent to adapt to new tasks, with the goal of creating a more versatile and scalable solution for robotic manipulation.
- **Explore potential real-world applications:** Identify and discuss potential use cases for the DRL-based control system in various industries, such as manufacturing, logistics, healthcare, and others, highlighting the potential benefits and challenges of implementing the proposed approach in practical settings.

1.5 Related Work

Over the past two decades, significant advancements have been made in the fields of robotics and Artificial Intelligence. These developments have led to various theoretical and practical applications, such as path planning algorithms [11], human-robot interaction [2], pick-and-place tasks [12], solving Rubik's cube using a single robotic hand through reinforcement learning [13], collision avoidance [3], and many others.

Path planning problems [14] have garnered considerable attention in recent years. Various methods have been proposed to address these challenges, including a Jacobian-based technique [15] for obstacle avoidance during path planning and Cyclic-Coordinate Descent (CCD), a heuristic iterative search algorithm detailed in [16] and [17].

Despite the progress made in path planning, several difficulties still persist. To overcome these challenges, researchers have turned to reinforcement learning techniques to tackle problems in engineering, computer science, and robotics. Reinforcement learning methods, particularly model-free approaches, offer flexibility for various applications. In recent

times, deep reinforcement learning has been employed to address classical problems with notable success. For instance, an actor-critic based reinforcement learning approach was developed in [18] to handle inputs for a classical formation control problem. Additionally, a standard path planning algorithm was implemented on a real UR5 robot arm using the Robot Operating System (ROS) [19].

The growing popularity of reinforcement learning techniques, especially deep reinforcement learning, underscores their potential in solving complex path planning issues in robotics. By leveraging these methods, we aim to improve the efficiency, safety, and adaptability of robotic manipulators in diverse applications.

Chapter 2

Methodology

In this project, we aim to develop a deep reinforcement learning (DRL) algorithm to train a robotic manipulator to track complex trajectories in 3D space. The primary motivation behind this project is to create an intelligent control system that can learn from its environment and adapt to various trajectories, reducing the need for manual programming or calibration. This project focuses on the application of DRL techniques, specifically the Deep Deterministic Policy Gradient (DDPG) algorithm, for controlling a UR5 robotic manipulator simulated in the Pybullet physics engine. The methodology of our project can be broadly divided into several key steps:

- First, we set up the environment using the Pybullet physics engine, which simulates the UR5 robotic manipulator and its interactions with the environment. This simulation allows us to model the robot's kinematics, dynamics, and collision handling, providing a realistic environment for training and testing the DRL agent.
- Next, we design the state and action spaces for the reinforcement learning problem. The state space is composed of various parameters representing the current state of the robotic manipulator and its environment, including joint angles, end-effector position, and target position. The action space, on the other hand, consists of continuous actions that control the manipulator's joint angles.
- During the training process, the DRL agent learns a policy to control the robotic manipulator by interacting with the simulated environment. It receives observations from the environment and takes actions based on its current policy, attempting to maximize the cumulative rewards over time. The agent learns by updating its policy using the observed state transitions, rewards, and estimated Q-values.
- We evaluate the performance of our DRL agent using various metrics, such as cumulative rewards, success rate, and trajectory tracking error. Once the model converges, we test its performance on various trajectories to assess its generalization capabilities and robustness to new situations.
- Finally, we discuss the challenges encountered during the project and outline potential future work for improving the model's performance and applicability to real-world robotic manipulator systems.

2.1 Deep RL Method Used

We have chosen to utilize a variant of the DDPG method, as discussed in the previous chapter, for our reinforcement learning model. The DDPG method is well-suited for handling stochastic action and state spaces, which are present in our robotic manipulator model, as the angles and distances of various components can be represented by any floating-point number. DDPG is renowned for its data efficiency and scalability, making it an ideal choice for this application.

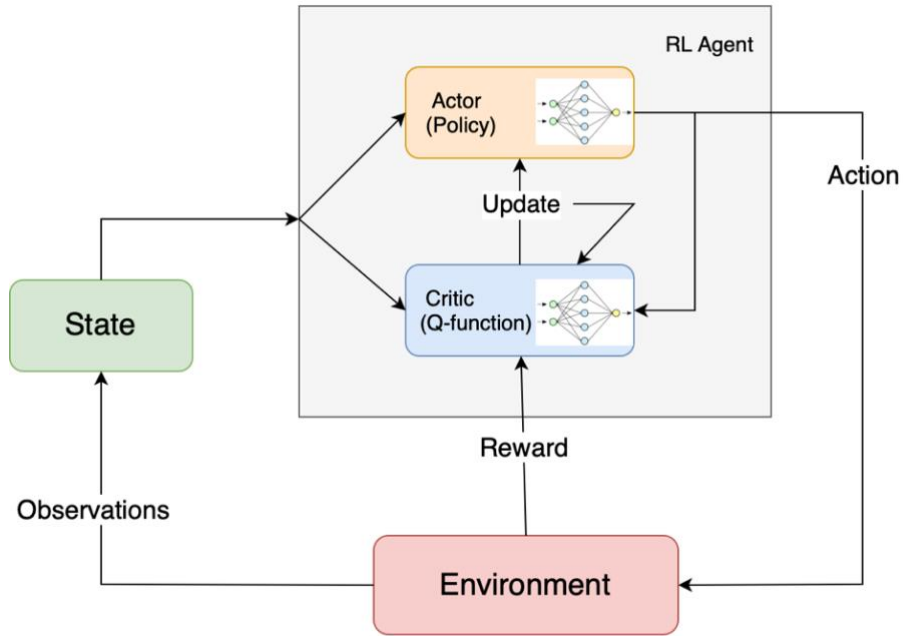


Figure 4: Overview of DDPG algorithm

To achieve the best possible results, we have implemented different structures for the actor and critic neural networks compared to the original DDPG architecture. These customizations enable our model to better adapt to the specific needs and constraints of our robotic manipulation task. For optimizing the neural networks, we have employed the Adam optimizer, which effectively performs backtracking using policy gradient techniques.

By leveraging the strengths of the DDPG method and customizing its architecture for our specific application, we have developed a reinforcement learning model that is capable of learning complex manipulation tasks with high precision and efficiency, making it an ideal solution for controlling the robotic manipulator. A general algorithm of a DDPG model is shown in Algorithm 1.

Algorithm 1: Deep reinforcement learning model-free off-policy actor-critic method

1. Initialize actor network, critic network, fixed behavior policy μ , maximum step limit S , maximum episode limit N , replay buffer R , maximum size of the replay buffer R_m .
 2. **For** episode = 1, N **do**
 3. Reset the environment.
 4. Set random goal.
 5. **For** step = 1, S **do**
 6. Choose action $a_t = \mu(s_t)$ according to fixed behaviour policy μ
 7. Execute action a_t , calculate reward r_t and get new state s_{t+1} .
 8. Store (s_t, a_t, r_t, s_{t+1}) in the replay buffer R .
 9. **If** size(R) > R_m **then:**
 10. Select a random sample of batch size from R .
 11. Update critic network by minimizing loss.
 12. Update actor network by policy gradient ascent.
 13. **End if**
 14. **End for**
 15. Store Reward, Accuracy and Q value for current episode.
 16. **End for**
-

2.2 Simulation

2.2.1 2D Simulation using Pyglet

To simulate and test a 2D version of the robotic manipulator, we have employed a graphical Python library called Pyglet. This library enables us to visualize the robotic arm within a graphical window based on the specified coordinates of each component, as illustrated in Figure 5. Once the agent provides an action, kinematics calculations are performed to determine the new coordinates for each arm and the target. Pyglet then redraws the arm with the updated coordinates. This process occurs multiple times per second, creating the impression of continuous movement along a trajectory.

By using Pyglet in conjunction with deep reinforcement learning techniques, we can effectively simulate and analyze the performance of our robotic manipulator in a 2D environment. This approach allows us to iteratively refine the agent's training and evaluate its ability to navigate complex paths, ultimately improving the efficiency and safety of the robotic arm's movements.

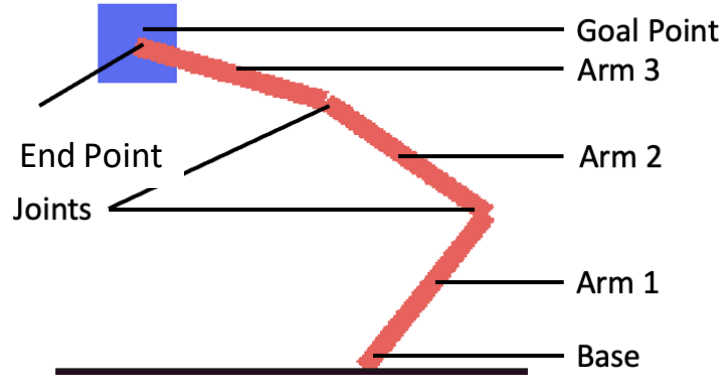


Figure 5: 2D simulation of robotic manipulator in Pyglet

2.2.2 3D Simulation using Pybullet

Pybullet is a powerful physics simulation engine used for simulating robotic systems and their interactions with the environment. It is an open-source library that is widely used in the robotics community for research, development, and testing purposes. Pybullet is written in C++ and has Python bindings, making it highly compatible with other machine learning libraries and frameworks, such as TensorFlow and PyTorch.

One of the key features of Pybullet is its ability to simulate both rigid and soft body dynamics, which enables it to model complex interactions between objects in the environment. This is particularly useful for simulating robotic manipulators, as it allows for accurate modeling of contact forces, friction, and collisions between the robot and its surroundings. In addition to its realistic physics simulation, Pybullet provides support for forward and inverse kinematics, making it possible to compute joint angles and end-effector positions based on desired trajectories.

Pybullet also offers a rich set of tools for visualizing and debugging simulations, including rendering 3D scenes, plotting joint trajectories, and displaying sensor data. This makes it easier for researchers and developers to analyze the performance of their robotic systems and identify potential issues or improvements.

In the context of this project, Pybullet serves as the simulation environment for developing and testing the DRL agent for trajectory tracking in cluttered and dynamic environments. The UR5 robotic manipulator, developed by Universal Robots, is used as the testbed for the DRL agent. Pybullet's realistic physics simulation and support for forward and inverse kinematics enable the accurate modeling of the manipulator's dynamics, while its visualization and debugging tools facilitate the analysis of the agent's performance and the fine-tuning of its control strategy.

Once the RL agent generates an action, Pybullet simulates the action and provides the resulting state of the robot. Based on this observation, we can compute the reward and the state vector, which are then fed back into the agent for further training. This process is repeated multiple times, creating the illusion of the robotic arm continuously moving along a trajectory. By using Pybullet for simulation, we can effectively train and test our RL agent in a realistic environment.

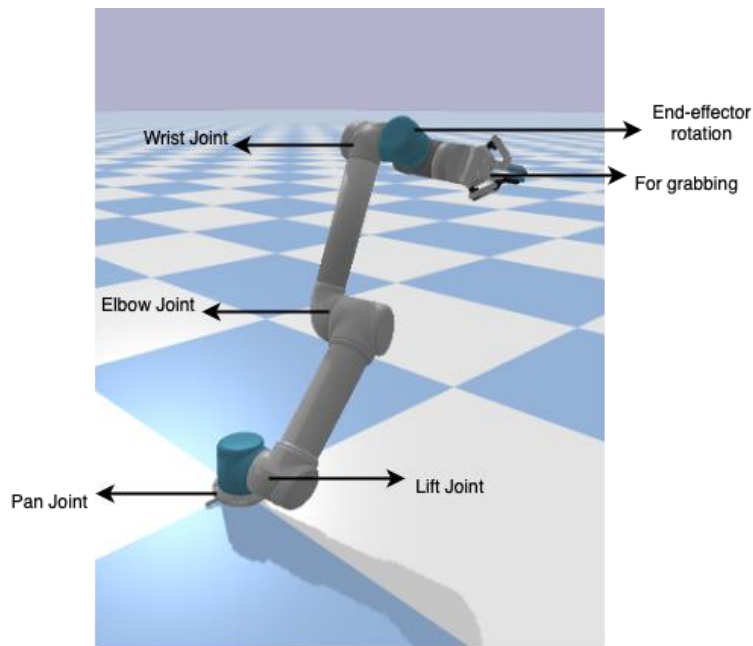


Figure 6: 3D simulation of robotic manipulator in Pybullet

2.2.3 Why did we do 2D simulation before 3D

Performing a 2D simulation before progressing to a 3D simulation offers several advantages in the development and testing process of robotic manipulators and reinforcement learning agents:

1. **Simplification:** A 2D simulation reduces the complexity of the problem, allowing researchers and developers to focus on fundamental aspects of the model, control algorithms, and reinforcement learning strategies without being overwhelmed by the intricacies of a 3D environment.
2. **Computational Efficiency:** 2D simulations typically require fewer computational resources and less processing time, enabling faster iterations and more efficient experimentation. This is particularly useful during the initial stages of development when multiple iterations may be necessary to refine the model and the learning algorithm.

3. **Debugging and Troubleshooting:** It is easier to identify and resolve issues in a 2D simulation due to its lower complexity. Debugging and troubleshooting in a 2D environment can provide valuable insights and reveal potential problems that could arise in the more complex 3D scenario.
4. **Visualization and Interpretation:** Results and behaviors observed in a 2D simulation are often more intuitive and easier to visualize, facilitating a better understanding of the learning process and the robotic manipulator's movements.
5. **Building a Foundation:** Developing and testing a reinforcement learning agent in a 2D environment can serve as a solid foundation for transitioning to a more complex 3D simulation. By refining the learning algorithms, state space representation, and reward functions in the simpler 2D context, developers can establish a strong basis for further expansion and adaptation to the 3D environment.

In summary, starting with a 2D simulation provides a more manageable and efficient development process, enabling researchers and developers to focus on key aspects of the model and learning strategies. Once a satisfactory level of performance is achieved in the 2D simulation, it can be extended and adapted to the more complex 3D environment, where additional challenges and considerations come into play.

2.3 Actor and Critic Neural Networks

The Deep Deterministic Policy Gradient (DDPG) algorithm [20] utilizes two artificial neural networks – the actor network, which calculates the policy, and the critic network, which computes the Q-value. Both networks consist of three layers, detailed as follows:

1. **Input layer:** The actor network's input layer features 31 neurons, which correspond to the state space, while the critic network's input layer has 35 neurons, accounting for both the state and action spaces.
2. **First hidden layer:** This layer contains 500 neurons with a Rectified Linear Unit (ReLU) activation function for both actor and critic networks.
3. **Second hidden layer:** This layer contains 1000 neurons with a Rectified Linear Unit (ReLU) activation function for both actor and critic networks.
4. **Third hidden layer:** The actor network's second layer comprises 500 neurons with a ReLU activation function, while the critic network's second layer features 500 neurons with a Hyperbolic Tangent (tanh) activation function.

5. **Output layer:** The actor network's output layer has four neurons with a tanh activation function, which provide the four values for the action space. In contrast, the critic network's output layer consists of a single neuron with a linear activation function, which calculates the Q-value.

The DDPG algorithm's architecture, which combines these actor and critic networks, enables efficient learning of continuous control policies for robotic manipulators, such as the one considered in this project. By training this model, we aim to achieve smooth, precise, and safe trajectory execution for the robotic arm.

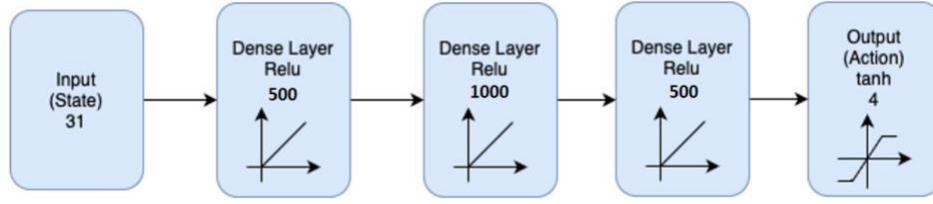


Figure 7: The architecture of Actor network

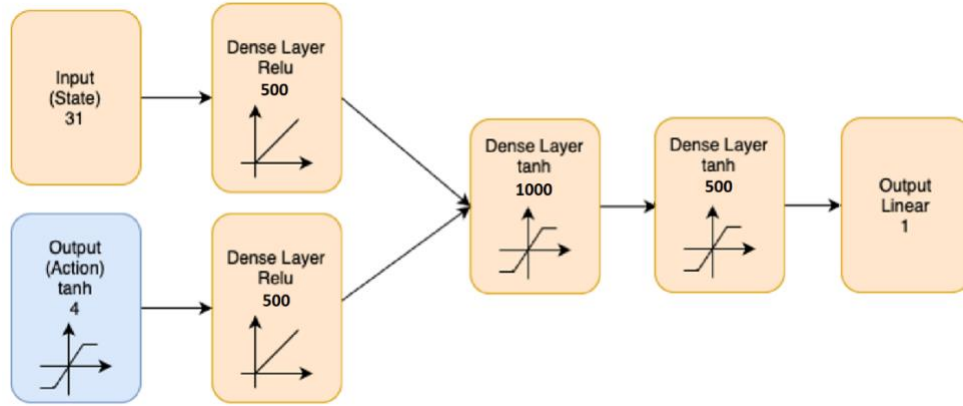


Figure 8: The architecture of Critic network

2.4 Action Space

The action space plays a crucial role in the DDPG algorithm, as it defines the set of actions that the agent can take in a given state. DDPG is specifically designed to handle continuous action spaces, making it suitable for a wide range of control problems, including robotic manipulation tasks. In continuous action spaces, actions are represented as real-valued vectors, which allows for more fine-grained control over the system.

2.4.1 Action Space for 2D model

The action space for our robotic manipulator consists of an array of three floating-point numbers, representing the angles by which each arm should move in the clockwise direction. The range for each angle is limited to between -1 and 1 radians to ensure that the arm does not execute excessively large movements. For instance, consider an action space of [0.32, -0.66, 0.82]. This would translate to the following arm movements:

1. **First arm:** Moves 0.32 radians clockwise.
2. **Second arm:** Moves 0.66 radians counterclockwise (since the value is negative).
3. **Third arm:** Moves 0.82 radians clockwise.

2.4.2 Action Space for 3D model

The action space for the 3D model consists of an array of four floating-point numbers, representing the angles by which each of the bottom four joints should move in the clockwise direction. The range for each angle is limited to between -0.5 and 0.5 radians to ensure that the arm does not execute excessively large movements. For instance, consider an action space of [0.32, -0.26, 0.42, 0.11]. This would translate to the following joint movements:

1. **Pan joint:** Moves 0.32 radians clockwise.
2. **Lift joint:** Moves 0.26 radians counterclockwise (since the value is negative).
3. **Elbow joint:** Moves 0.42 radians clockwise.
4. **Wrist joint:** Moves 0.11 radians clockwise.

By defining the action space in this manner, we can effectively control the robotic manipulator's movement and train it using deep reinforcement learning techniques to follow smooth and precise trajectories.

2.5 State Space

The state space is equally important in the DDPG algorithm, as it defines the representation of the environment and the information available to the agent for making decisions. The state space consists of all possible configurations of the environment that the agent can encounter. The state space directly impacts the input layer of both the actor and critic networks in DDPG. The state space's dimensionality and complexity determine the network architecture, including the number of input neurons and potentially the depth and width of the network. A well-designed state representation can facilitate learning, while an inadequate or overly complex state representation can hinder the learning process.

To accurately capture the state, it is essential to consider not only the current position of the arm (using angles and coordinates) but also the goal point. This is because the action taken depends on both the arm's current position and the desired target location. By including the goal point in the state space, we ensure that the deep reinforcement learning agent has sufficient information to make informed decisions regarding the optimal actions needed to move the robotic manipulator smoothly and precisely toward its target. This comprehensive representation of the state enables the agent to learn more effectively and achieve better performance in executing trajectories.

2.5.1 State Space for 2D model

The state space for the 2D robotic manipulator model is represented by an array consisting of 13 floating-point numbers, which are detailed as follows:

- The initial two numbers correspond to x and y coordinates of the end of first arm.
- Subsequently, the next pair of numbers represent the x and y coordinates of the end of the second arm.
- Following this, another set of two numbers indicate the x and y coordinates of the end of the third arm, which is also considered as the endpoint.
- The next two numbers describe the distance between the x and y coordinates of the end of the first arm and the goal point, measured along the x and y axes, respectively.
- The subsequent pair of numbers represent the distance between the x and y coordinates of the end of the second arm and the goal point, again measured along the x and y axes, respectively.
- The next two numbers illustrate the distance between the x and y coordinates of the end of the third arm (the endpoint) and the goal point, measured along the x and y axes, respectively.
- The final number is a Boolean value, which signifies if the endpoint is near the goal point. This determination is based on whether the coordinates of the goal point are less than 20 pixels away from the endpoint in both the x and y directions.

2.5.2 State Space for 3D model

For the 3D model, we have designed the state space as an array of 31 floating-point numbers, described as follows:

- 3 numbers for the x, y, and z coordinates of the pan joint, along with 3 numbers representing the differences between the x, y, and z coordinates of the goal point and the pan joint position.

- 3 numbers for the x, y, and z coordinates of the lift joint, along with 3 numbers representing the differences between the x, y, and z coordinates of the goal point and the lift joint position.
- 3 numbers for the x, y, and z coordinates of the elbow joint, along with 3 numbers representing the differences between the x, y, and z coordinates of the goal point and the elbow joint position.
- 3 numbers for the x, y, and z coordinates of the wrist joint, along with 3 numbers representing the differences between the x, y, and z coordinates of the goal point and the wrist joint position.
- 3 numbers for the x, y, and z coordinates of the end-effector, along with 3 numbers representing the differences between the x, y, and z coordinates of the goal point and the end-effector position.
- The final number indicates whether the current episode is completed.

2.6 Reward Function

One of the most important components of a reinforcement learning model is its reward function [21]. The actions of the agent will be judged by the reward function. We have incorporated several things in this function, like how close the endpoint is to the goal, how many actions it took to reach the goal, whether the action given by the agent made the robot to cross physical constraints, etc. We have designed the reward function keeping these things in mind.

This reward, along with the new state of the robot, is given back to the reinforcement learning agent. The agent uses this information to train itself and send the next action. This iterative process continues until stopped, allowing the agent to learn how to control the robotic manipulator efficiently and follow precise, smooth trajectories.

2.6.1 Reward Function for 2D Model

First, the reward function is penalized for how far it is from the goal point. This is done by taking negative of the distance between endpoint and goal point. This is called distance reward r_d .

$$r_d = -\sqrt{(e_x - g_x)^2 + (e_y - g_y)^2}$$

Here \mathbf{e} is the endpoint of the arm and \mathbf{g} is the goal point.

Next, the reward is subtracted by 1 if the arm is out of physical constraints. This happens when any point on the arm has a negative coordinate.

$$r_c = \begin{cases} -1 & \text{if any point on the arm is below } x \text{ axis} \\ 0 & \text{otherwise} \end{cases}$$

Finally, the reward is added by 1 if the endpoint is close to the goal point. This is called arrive reward r_a .

$$r_a = \begin{cases} 1 & \text{if } (g_x - 20) < e_x < (g_x + 20) \text{ and } (g_y - 20) < e_y < (g_y + 20) \\ 0 & \text{otherwise} \end{cases}$$

Final reward is given by:

$$r = r_a + r_c + r_d$$

2.6.2 Reward Function for 3D Model (Old Version)

First, the reward function is penalized for how far it is from the goal point. This is done by taking negative of the distance between endpoint and goal point. This is called distance reward r_d .

$$r_d = -\sqrt{(e_x - g_x)^2 + (e_y - g_y)^2 + (e_z - g_z)^2}$$

Here \mathbf{e} is the endpoint of the arm and \mathbf{g} is the goal point.

Next, the reward is subtracted by 1 if any joint rotates by more than $\pi/4$ radian in one step. This is called constrain reward r_c .

$$r_c = \begin{cases} -1 & \text{if any joint rotates by more than } \pi/4 \text{ radian in one step} \\ 0 & \text{otherwise} \end{cases}$$

Finally, the reward is added by 1 if the endpoint is close to the goal point. This is called arrive reward r_a .

$$r_a = \begin{cases} 1 & \text{if } \mathbf{abs}(\mathbf{e}_x - \mathbf{g}_x) < \gamma \text{ and } \mathbf{abs}(\mathbf{e}_y - \mathbf{g}_y) < \gamma \text{ and } \mathbf{abs}(\mathbf{e}_z - \mathbf{g}_z) < \gamma \\ 0 & \text{otherwise} \end{cases}$$

γ is the threshold value for distance between end-effector and goal point. We have taken its value as 0.05 for our model.

Final reward is given by:

$$r = r_a + r_c + r_d$$

2.6.2 Reward Function for 3D Model (New Version)

First, the Euclidean distance is calculated between the end effector position and the goal point.

$$dist = \sqrt{(e_x - g_x)^2 + (e_y - g_y)^2 + (e_z - g_z)^2}$$

Here \mathbf{e} is the endpoint of the arm and \mathbf{g} is the goal point.

Then, the distance reward is calculated in accordance with a threshold value as indicated below.

$$r_d = \begin{cases} 100 * \left(1 - \frac{dist}{threshold}\right)^{\frac{1}{2}}, & \text{if } dist \leq threshold \\ \left(1 - \frac{dist}{threshold}\right), & \text{otherwise} \end{cases}$$

We have taken threshold value as 0.1 for our model. r_d component encourages the robot to move closer to the goal.

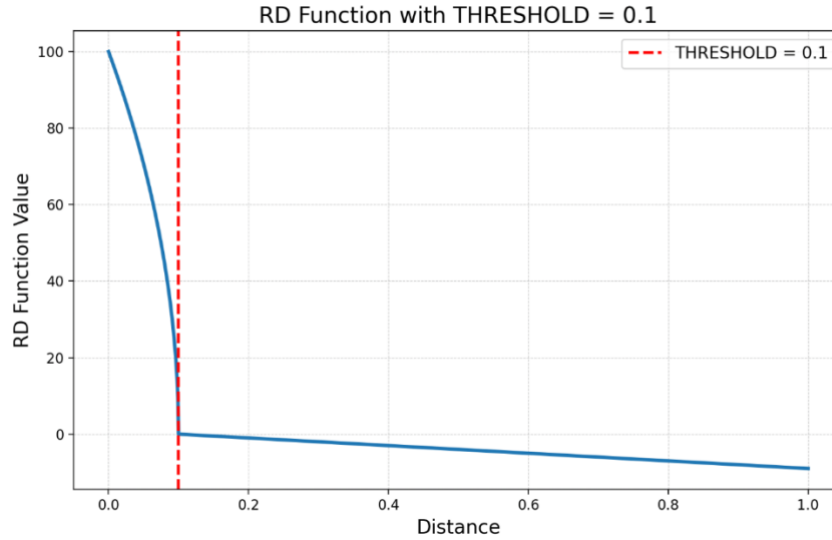


Figure 9: Variation of r_d with distance

Next, the reward is subtracted by the absolute value of difference of current angle and previous angle for bottom 4 joint, whole divided by a constant. This is called constraint reward. This component encourages the robot to minimize joint angle changes, thus promoting smoother and more efficient movements.

$$r_c = - \sum_{j=1}^4 abs(curr_angle_j - prev_angle_j)/4$$

Finally, the reward is increased in accordance with r_d if the endpoint is close to the goal point for a consecutive 25 steps. This is called arrive reward r_a . This component rewards the robot for achieving the goal.

$$r_a = \begin{cases} \sum_0^{25} r_d, & \text{if end point is within threshold for 25 consecutive steps} \\ 0, & \text{otherwise} \end{cases}$$

Final reward is given by:

$$r = r_a + r_c + r_d$$

2.7 Training Algorithm

1. Initialize the environment with a UR5 robotic arm instance.
2. Open files for logging accuracy, rewards, and Q-values.
3. Set the initial accuracy value to 0.
4. Iterate through episodes:
 - a. Reset the environment and obtain the initial state.
 - b. Initialize the episode reward and Q-value accumulators.
 - c. Set a random goal point.
 - d. Iterate through steps within each episode:
 - i. The agent chooses an action based on the current state.
 - ii. Execute the chosen action in the environment and obtain the next state, reward, and completion status (done).
 - iii. Store the transition (state, action, reward, next state) in the agent's memory.
 - iv. Update the episode reward and Q-value accumulators.

- v. If the agent's memory is full, update the agent's model by learning from a batch of sampled transitions.
 - vi. Set the current state to the next state.
 - vii. If the episode is complete or the maximum number of steps is reached:
 - 1. Update the accuracy value based on the completion status.
 - 2. Log the accuracy, episode reward, and Q-value.
 - 3. Print the episode result.
 - 4. Break out of the step loop and move to next episode.
 - e. Print the goal position and the final end-effector position for the episode.
 - f. If maximum no. of episodes is reached, break out of the episode loop.
- 5. Save the agent's model for testing and evaluation after training is complete.

2.8 Learning Process of DRL Agent

The learning process is based on the Deep Deterministic Policy Gradient (DDPG) algorithm, an actor-critic reinforcement learning method designed for continuous control tasks. Here's an explanation of the learning process:

1. **Initialization:** The DDPG class initializes two pairs of neural networks, the Actor and Critic, with separate networks for evaluation (online) and target (delayed update). It also initializes a replay buffer to store experience tuples (state, action, reward, next state).
2. **Interaction with the environment:** The agent chooses an action based on the current state using the online Actor network. The action is then executed in the environment, which returns the next state, reward, and the done flag. The experience tuple is stored in the replay buffer.
3. **Learning from the experience:** Once the replay buffer is filled enough, the learning process begins. A random batch of experiences is sampled from the buffer. The online Actor network generates actions for the next states, which are then fed into the target Critic network to calculate the target Q-values. The target Q-values are combined with the rewards from the sampled experiences to compute the target values for the online Critic network.

4. **Critic network update:** The online Critic network is trained using the sampled experiences and the computed target values, minimizing the mean squared error between the predicted Q-values and the target values.
5. **Actor network update:** The online Actor network is updated to maximize the Q-values predicted by the online Critic network for the current states and the generated actions. This is done using gradient ascent, effectively updating the policy to favor actions that yield higher Q-values.
6. **Soft target network update:** The weights of the target networks (both Actor and Critic) are updated using a soft update strategy, which is a weighted average of the online and target network weights. This helps to stabilize learning by maintaining a slowly changing target network.

In summary, the learning process involves the interaction of the Actor and Critic networks, using experience replay and soft target updates to learn a policy that maximizes the expected cumulative reward in a continuous control task.

Chapter 3

Result and Analysis

This chapter explains the training of the agent and results and simulation of various tests performed on the robotic manipulator.

3.1 Training Details

The agent's training process is divided into episodes. In each episode, the model attempts to reach the goal point within a specified distance and a given number of steps. If the agent successfully reaches the goal point, it receives a positive reward. However, if the model takes an excessive number of steps or fails to reach the goal, the reward becomes negative. We have set the maximum step value to 100, meaning that if the robot is unable to reach the goal within 100 actions, the episode is considered unsuccessful.

At the beginning of each episode, a random point is generated within the robot's reachable range, which serves as the goal point. The robot's objective is to move its end-effector to this goal point using a maximum of 100 actions. The algorithm is developed and trained using Python 3.8 and TensorFlow 2.8.

Various hyperparameters are employed during the training process, and these are critical in determining the model's performance. Some of the key hyperparameters include the learning rate for both the actor and critic, the reward discount factor (GAMMA), and the soft target update factor (TAU). Additionally, the training utilizes a BATCH_SIZE for sampling experiences from the memory buffer during the learning process. These hyperparameters play a significant role in the agent's ability to learn and adapt effectively to the environment, eventually leading to improved trajectory tracking performance.

3.2 Hyperparameters

The various hyperparameters used are essential for tuning the DDPG algorithm's performance. A brief explanation of each of them is included in this section.

<i>Hyperparameter</i>	<i>Value</i>
<i>Number of Episodes</i>	15000
<i>Steps per Episode</i>	100
<i>Learning Rate of Actor Network</i>	0.001
<i>Learning Rate of Critic Network</i>	0.001
<i>Reward Discount Factor</i>	0.9
<i>Soft target network update factor</i>	0.01
<i>Batch Size</i>	32
<i>Distance Threshold</i>	0.1
<i>Size of Replay Buffer</i>	100000

Table 1: Values of Hyperparameters used during training.

- **Number of Episodes (15000):** The number of episodes is a hyperparameter that determines the total number of training iterations the agent will go through during the learning process. Increasing the number of episodes allows the agent to gather more experience and refine its policy over time. However, too many episodes can lead to overfitting, where the agent becomes too specialized in the training environment, and it may also cause divergence of loss function.
- **Steps per Episode (100):** Steps per episode is a hyperparameter that defines the maximum number of steps an agent can take within a single episode. This value limits the length of an episode and prevents the agent from getting stuck in an infinite loop or taking too long to complete a task. By limiting the number of steps per episode, researchers can encourage the agent to learn more efficient policies that can solve the problem within a given time frame.
- **Learning rate for the Actor network (0.001):** The learning rate determines the step size during gradient ascent while updating the Actor network. A smaller learning rate ensures more stable but slower learning, while a larger learning rate can lead to faster but possibly unstable learning.
- **Learning rate for the Critic network (0.001):** Similar to the Actor learning rate, the Critic learning rate determines the step size during gradient descent while updating the Critic network.
- **Reward discount factor (0.9):** The discount factor determines how much the agent values future rewards compared to immediate rewards. A value closer to 1 means that the agent gives more importance to future rewards, while a value closer to 0 means the agent focuses on immediate rewards.

- **Soft target network update factor (0.01):** This hyperparameter controls the rate at which the target networks (Actor and Critic) are updated with the online networks' weights. A smaller value means slower updates, which can help stabilize learning, while a larger value means faster updates, which could lead to instability.
- **Batch Size (32):** The batch size determines the number of experience tuples sampled from the replay buffer during each learning step. A larger batch size can lead to more stable learning due to averaging over more samples but may also require more computational resources.
- **Distance Threshold (0.1):** The distance threshold is a hyperparameter that determines the allowable distance between the end-effector of the robot and the goal point for the task to be considered successful. It represents a tolerance value for the agent's performance, allowing for some degree of error in the robot's final position. By setting an appropriate distance threshold, researchers can balance the trade-off between precision and computational efficiency. A lower distance threshold requires the agent to be more accurate in reaching the goal points, potentially leading to slower learning, and increased computational resources. On the other hand, a higher distance threshold may allow the agent to learn more quickly but may result in less precise trajectory tracking.
- **Size of Replay Buffer (100000):** The size of the memory buffer, also known as the replay buffer, is a hyperparameter that determines the maximum number of transitions (state, action, reward, next state) that can be stored for use in the training process. The memory buffer plays a crucial role in off-policy learning algorithms like DDPG, where the agent learns from a set of past experiences rather than solely from the most recent interaction with the environment. A larger memory buffer can store more diverse experiences, but it will also require more memory and may increase the computational overhead for sampling and updating the stored transitions. On the other hand, a smaller memory buffer may be more memory-efficient but might limit the diversity of experiences available for training, potentially leading to slower learning or suboptimal policies.

3.3 Evaluation Metrics

Here are some of the evaluation metrics we have used to keep a check on the training progress of our model. These metrics have helped us in hyperparameter tuning, reward function modelling and finding bugs in our program.

3.3.1 Average reward of an episode

The average reward of an episode is an evaluation metric that quantifies the overall performance of the agent during a single episode. It is calculated as the sum of rewards received at each time step, divided by the total number of steps in the episode. A higher average reward indicates that the agent is effectively interacting with the environment and making progress towards its goals. This metric can be used to track the agent's learning progress and to identify potential issues with the agent's performance.

3.3.2 Accuracy of the last 100 episodes

The accuracy of the last 100 episodes is a rolling average measure of the agent's success in reaching the goal points during trajectory tracking tasks. It is calculated by taking the ratio of successful goal points reached to the total number of goal points in the last 100 episodes. This metric provides a smoother estimate of the agent's performance over time, as it reduces the impact of outliers and short-term fluctuations. By monitoring the accuracy of the last 100 episodes, we can assess the agent's generalization capabilities and its ability to adapt to different trajectories.

3.3.3 Average Q value of an episode

The average Q value of an episode is an evaluation metric that measures the agent's estimated value function during a single episode. The Q value represents the expected cumulative reward from taking a specific action in each state and following the current policy thereafter. By averaging the Q values over all the steps in an episode, this metric can provide insights into the agent's learning process and its convergence towards an optimal policy. High average Q values indicate that the agent is learning to associate higher expected rewards with its chosen actions, which can lead to better overall performance.

3.3.4 Length (no. of steps) of an episode

The length of an episode, measured in terms of the number of steps taken, is an important evaluation metric for assessing the efficiency and effectiveness of the agent in solving the trajectory tracking tasks. An ideal agent should be able to reach the goal points in as few steps as possible, thus minimizing the resources and time required to complete the task. By tracking the length of episodes, researchers can identify potential issues with the agent's exploration strategy or policy updates, such as excessive steps indicating an inefficient policy or inadequate exploration.

3.4 Accuracy

During training accuracy is calculated using the following formula.

$$acc = \begin{cases} 0.01 + 0.99 * acc & \text{if current episode is completed} \\ 0.99 * acc & \text{otherwise} \end{cases}$$

Accuracy is calculated after each episode. If the episode is completed, then the accuracy increases otherwise it decreases. The accuracy is graphed w.r.t no. of episodes in figure 10.

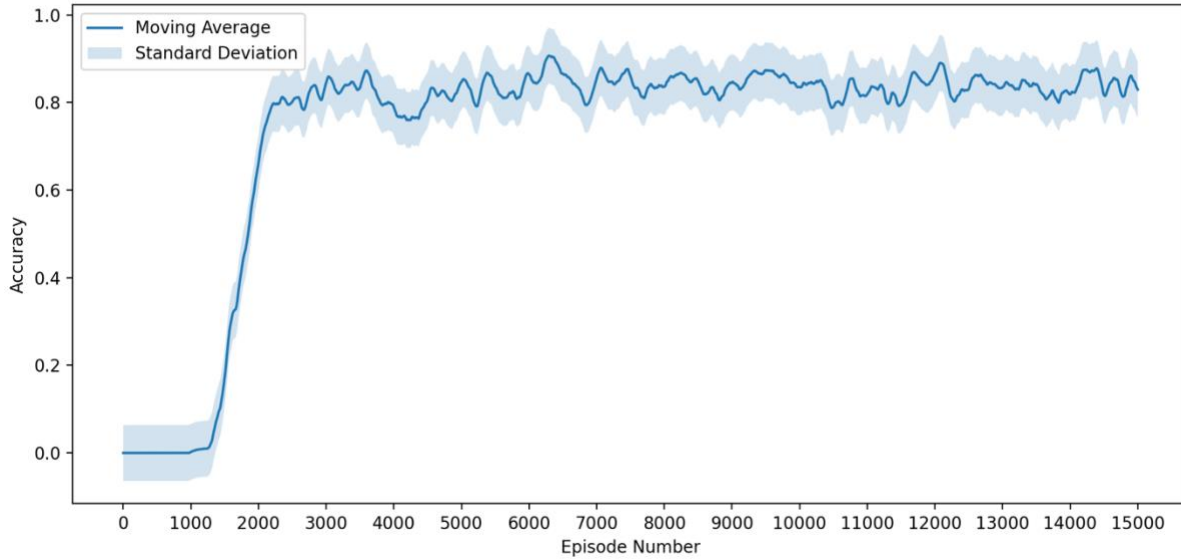


Figure 10: Accuracy vs No. of episodes

During the initial phase of training, lasting for around 500 episodes, the agent focuses on exploring the environment by taking random actions. This exploration phase is crucial for the agent to learn about the different states it can encounter and the corresponding rewards or penalties it can receive for its actions. During this phase, the agent's performance may be poor, as reflected by the 0% accuracy.

Once the agent reaches 500 episodes, it begins to learn from its actions, and the accuracy starts to improve. As the agent transitions from exploration to exploitation, it leverages its learned knowledge to make more informed decisions, aiming to maximize the cumulative rewards it receives. The accuracy rapidly improves and reaches saturation after 15,000 episodes, with a final training accuracy of 83.47%.

It is important to note that the training accuracy does not necessarily indicate the testing accuracy. The testing accuracy is a measure of the agent's performance in novel situations or environments that it has not encountered during training.

3.5 Reward

The reward is calculated using the reward function detailed in the previous chapter. The primary objective of any reinforcement learning (RL) algorithm is to maximize the accumulated reward. In the initial stages of training, the agent receives negative rewards as it explores and adapts to the environment, attempting various actions to learn the optimal policy. The reward is graphed w.r.t no. of episodes in figure 11.

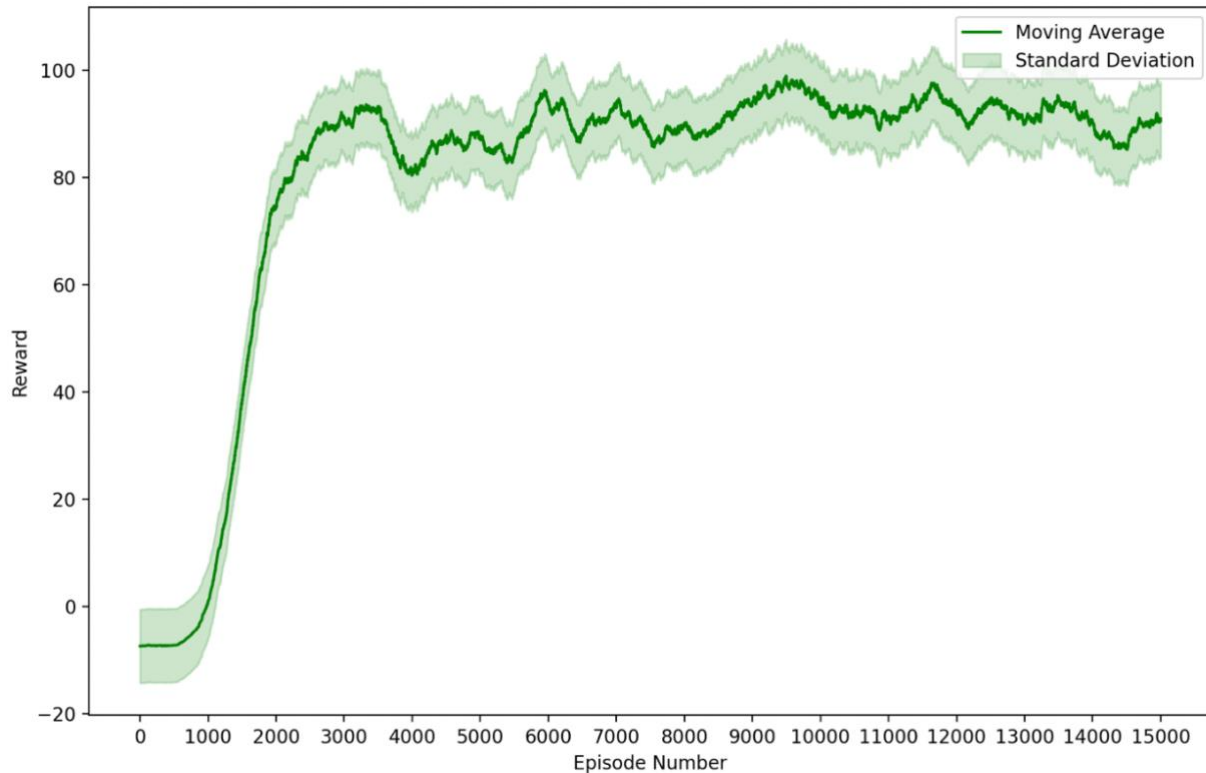


Figure 11: Reward vs No. of episodes

During the beginning of the training process, the reward values are highly negative, as the model predominantly takes random actions, resulting in the agent being far from the goal point. This phase is crucial for exploration, as it allows the agent to sample different actions and learn their consequences. As the model continues to learn from its experiences, it becomes more proficient in reaching the goal point with fewer steps per episode, leading to positive rewards. The rewards gradually increase throughout the training process, indicating the agent's improvement in performance and its ability to navigate the environment more effectively towards the desired goal.

3.6 Q Value

The Q Value is the output of critic network of the DRL agent. The Q value represents the expected cumulative reward from taking a specific action in each state and following the current policy thereafter.

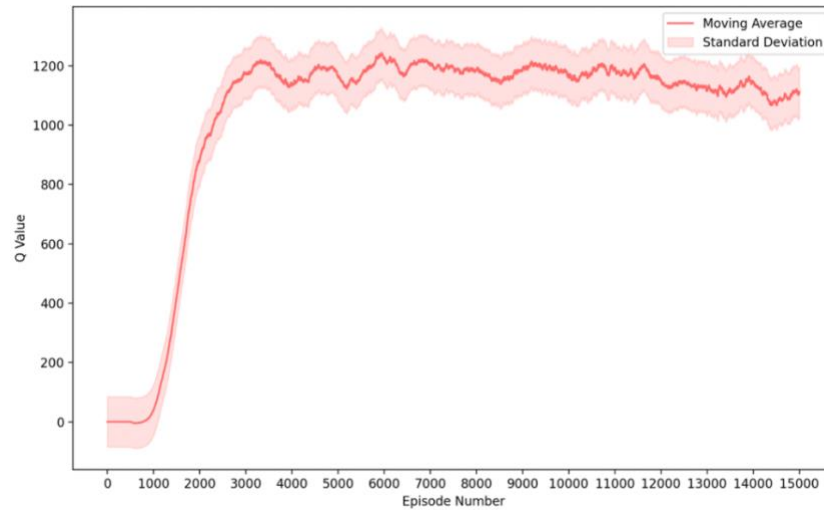


Figure 12: Q value vs No. of episodes

3.7 Episode Length

Episode length is the no. of steps taken to complete an episode. Lower value represents that the arm is able to reach its goal points quickly and the agent is performing well.

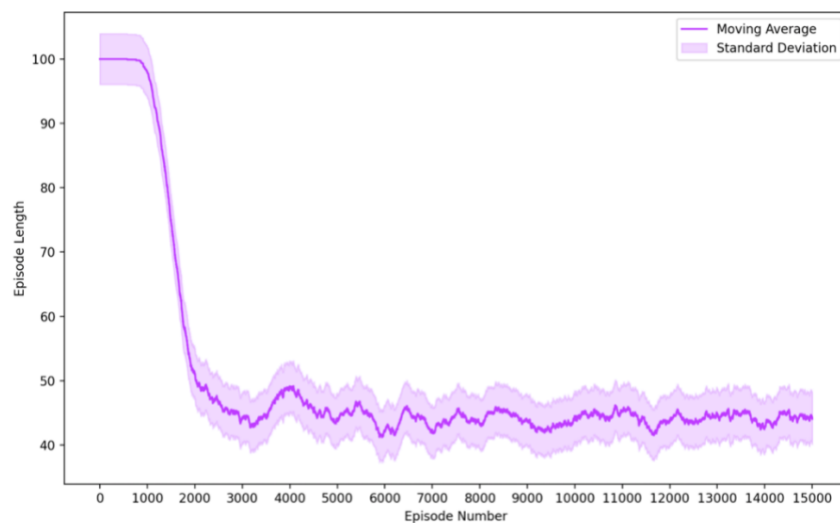


Figure 13: Episode length vs No. of episodes

3.8 2D Trajectory Tracking

Here are some of the trajectories tracked by the 2D model. This was shown in last semester work. 3D trajectories are given in section 3.9.

3.8.1 Semicircle

The arm was able to track the following semicircle with an accuracy of 99.39%.

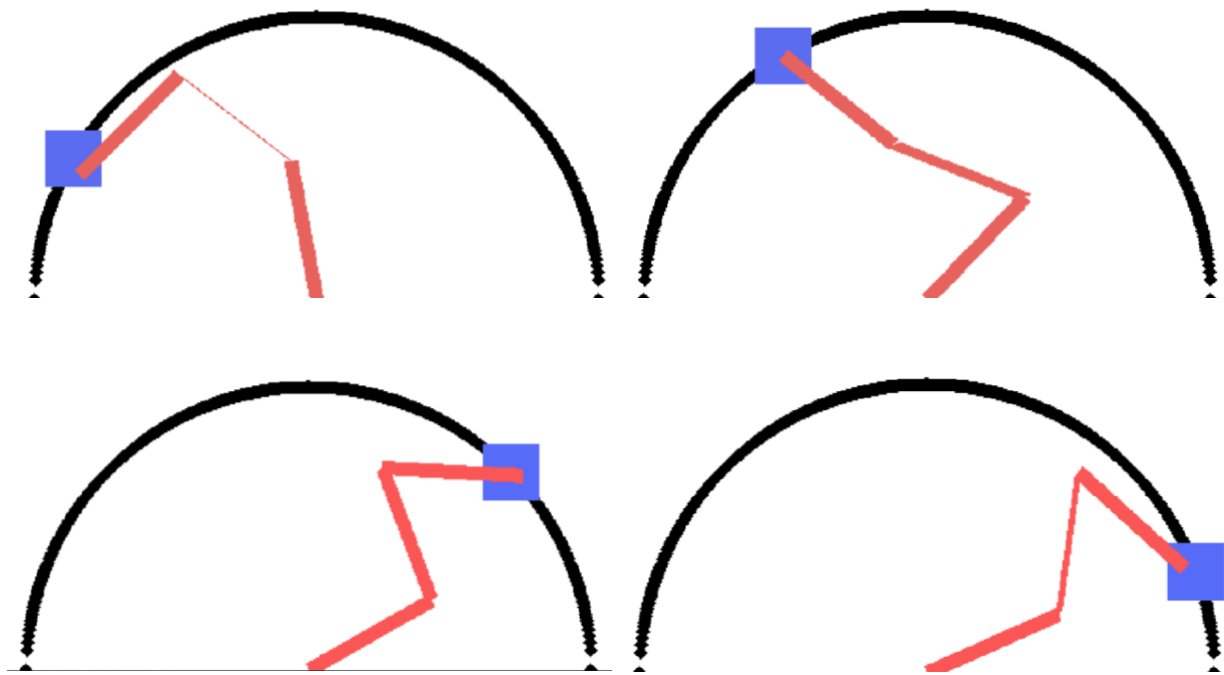


Figure 14: 2D Trajectory tracking of a semicircle

3.8.2 Rectangle

The arm was able to track the following rectangle with an accuracy of 99.51%.

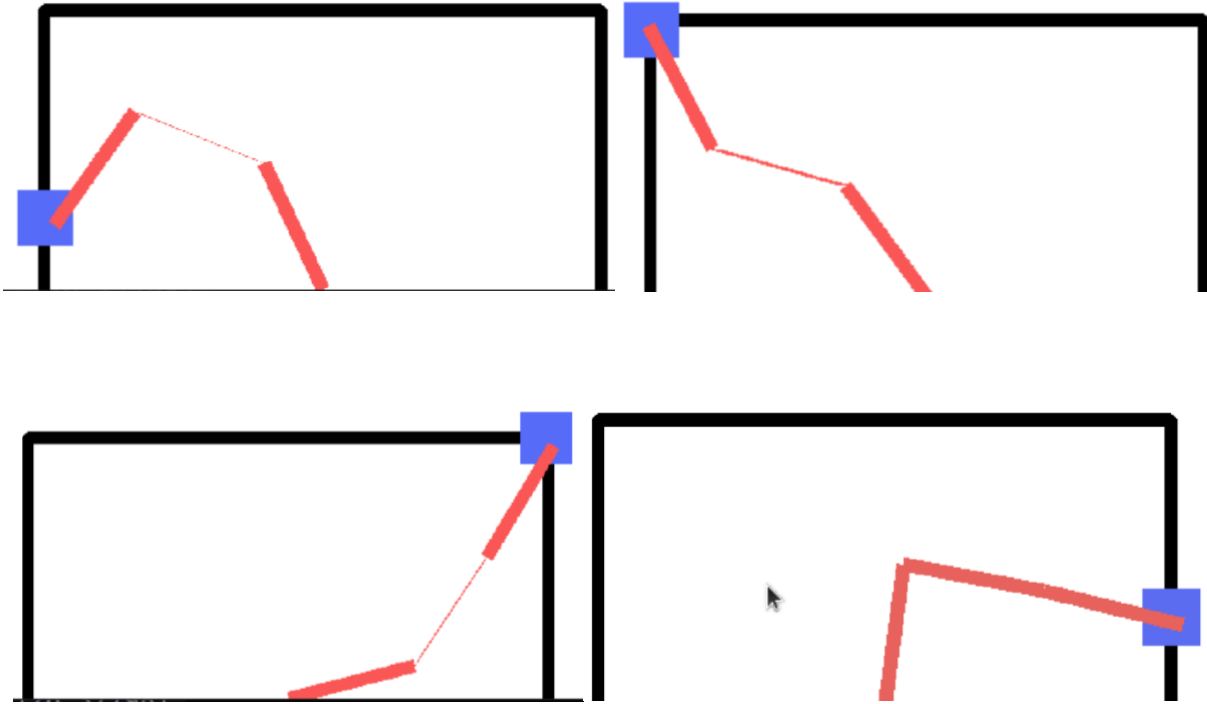


Figure 15: 2D Trajectory tracking of a rectangle

3.9 3D Trajectory Tracking

To track a trajectory, we first convert it into a series of closely spaced points. The manipulator must then reach each point sequentially, following the specified path. To evaluate the accuracy of trajectory tracking, we compute the ratio of the number of points successfully reached by the manipulator to the total number of points in the trajectory. This provides a clear measure of the manipulator's performance in adhering to the intended path.

After training the reinforcement learning (RL) agent, we tested its ability to track various trajectories to assess its performance. By evaluating the agent on diverse trajectories, we can gain a comprehensive understanding of its capabilities and identify areas where further improvements may be needed.

3.9.1 Crown

The arm was able to track the following crown-shaped trajectory with an accuracy of 100%.

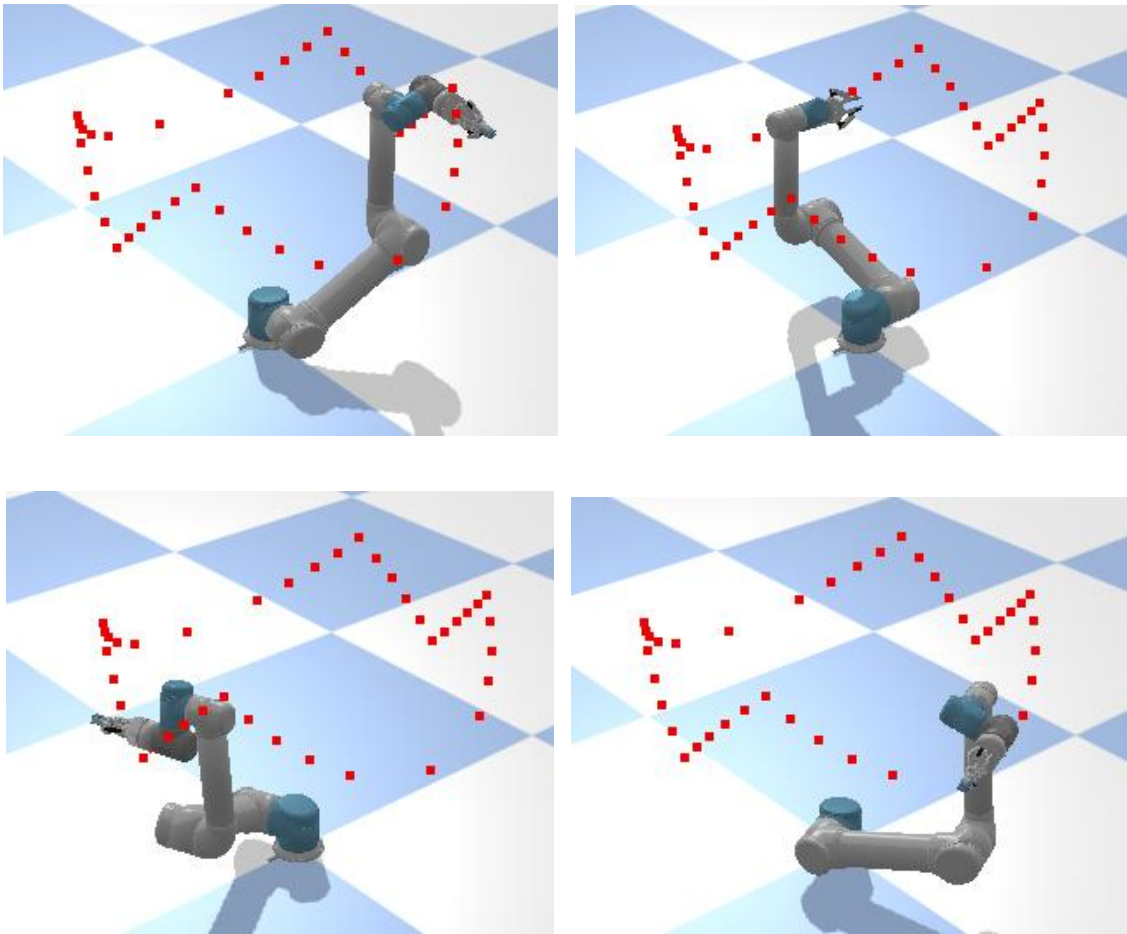


Figure 16: Tracking of a crown-shaped trajectory

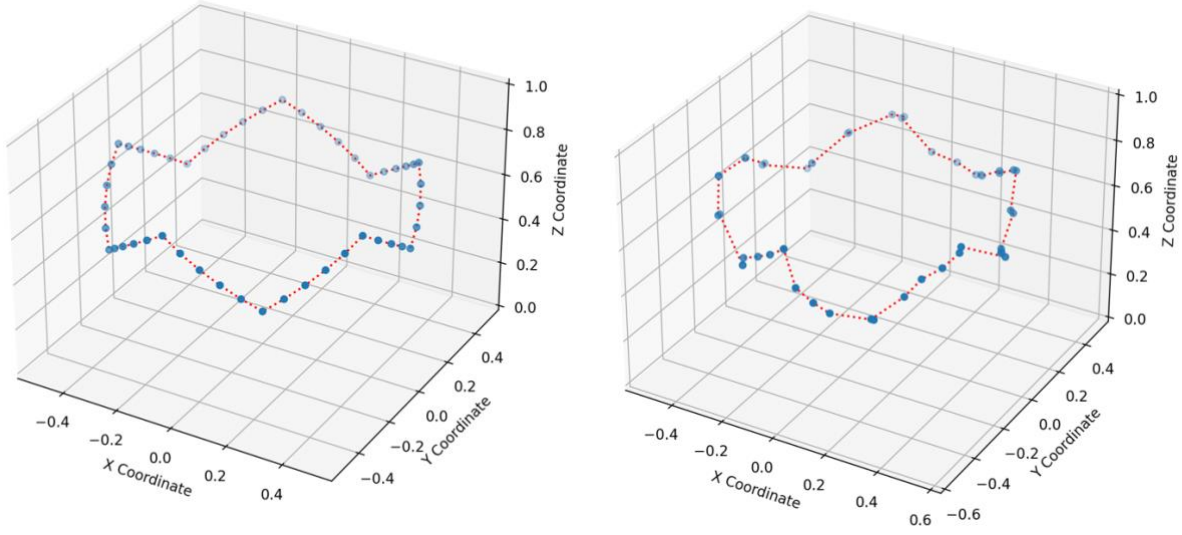


Figure 17: Actual Trajectory (left) and tracked trajectory (right) for crown-shaped trajectory

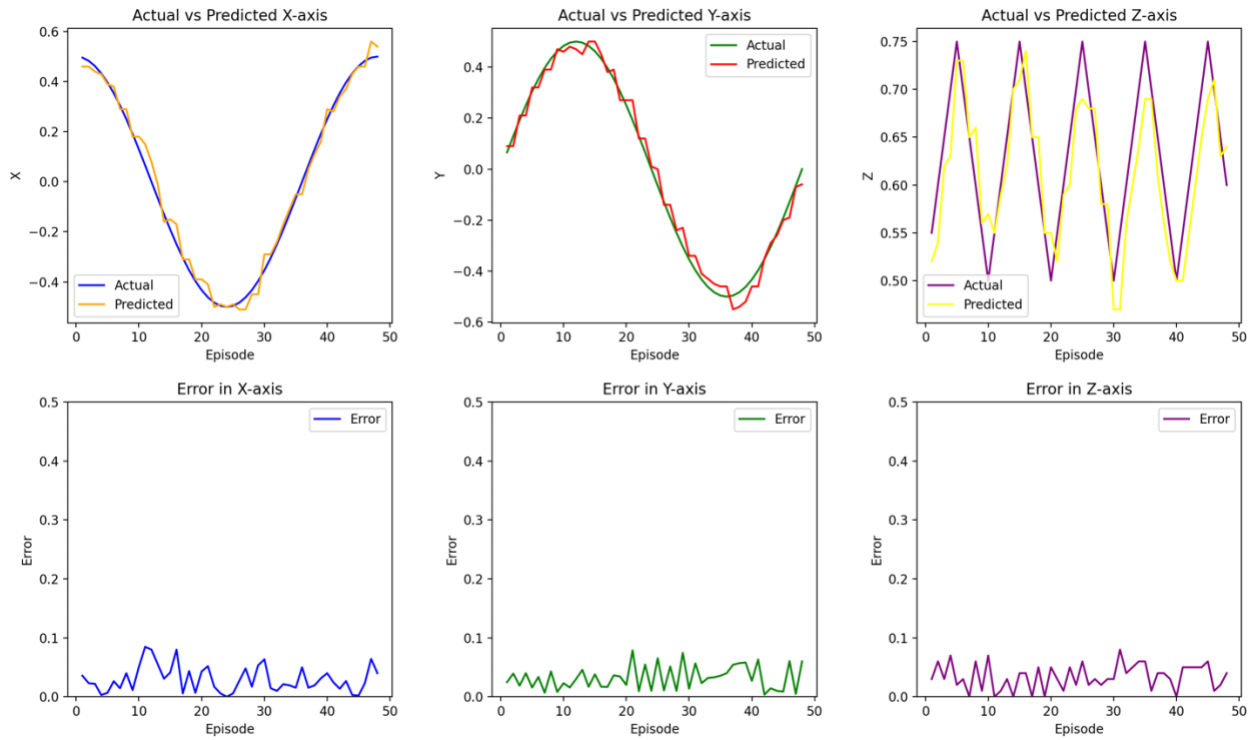


Figure 18: Analysis of trajectory tracking of crown in different axes

3.9.2 Rectangle

The arm was able to track the following rectangular trajectory with an accuracy of 100%.

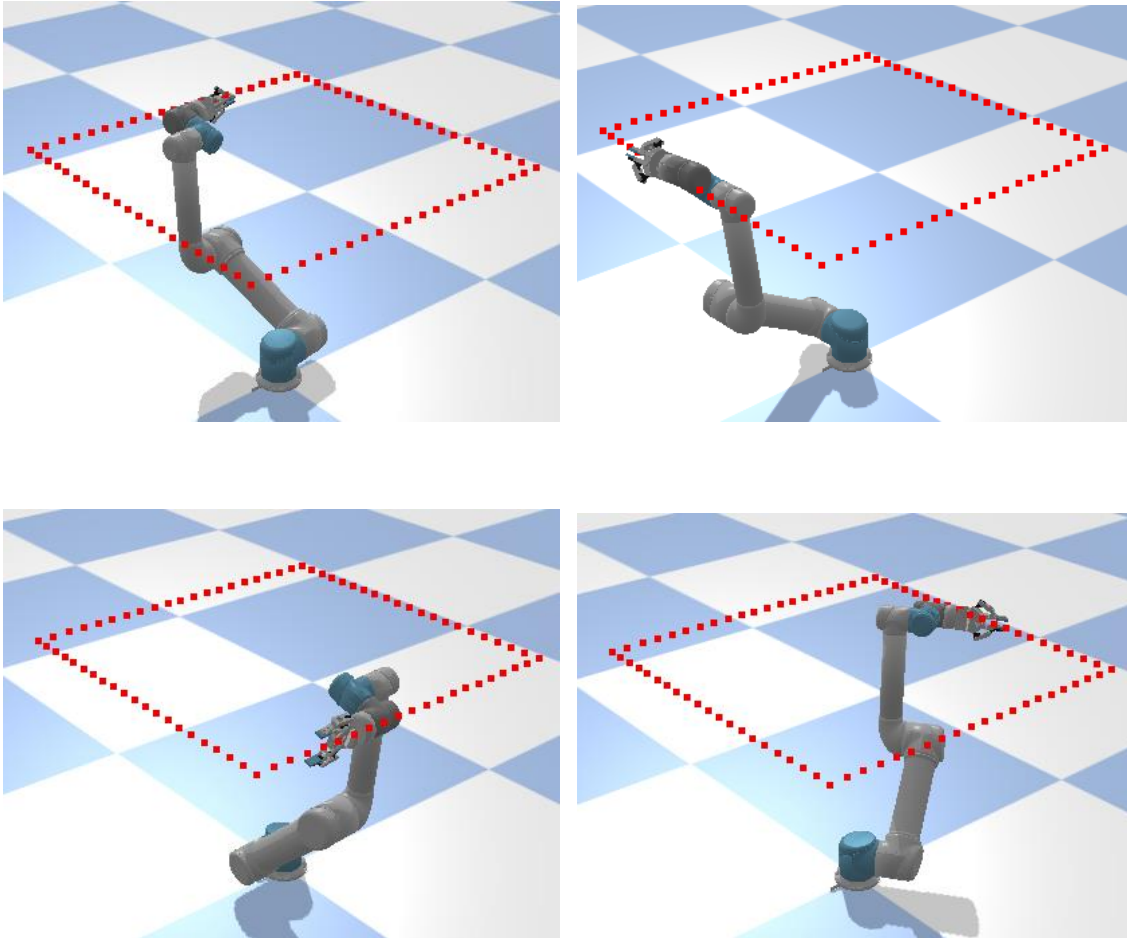


Figure 19: Tracking of a rectangular trajectory

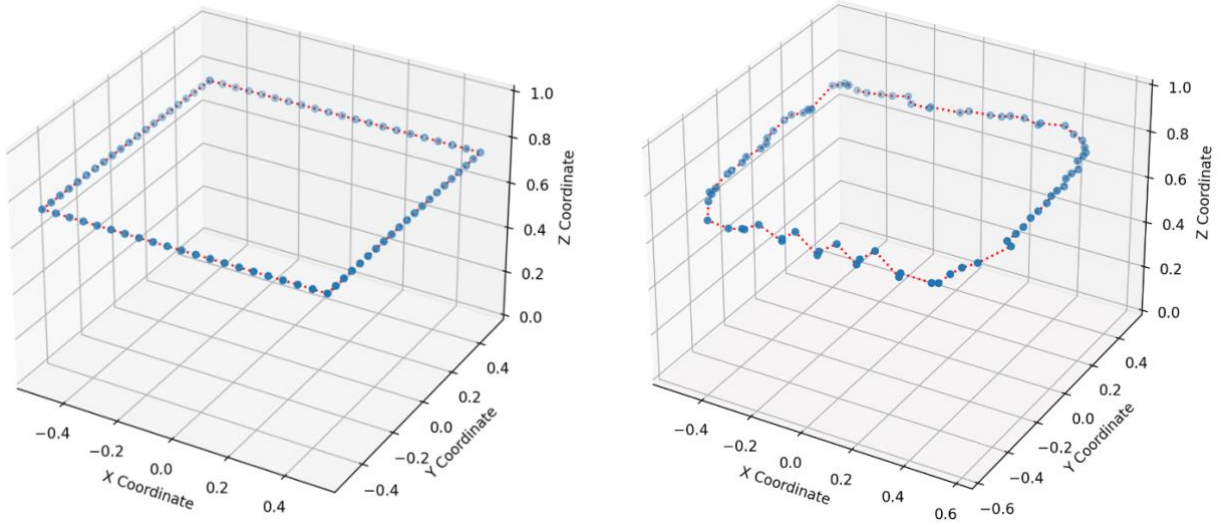


Figure 20: Actual Trajectory (left) and tracked trajectory (right) for rectangular trajectory

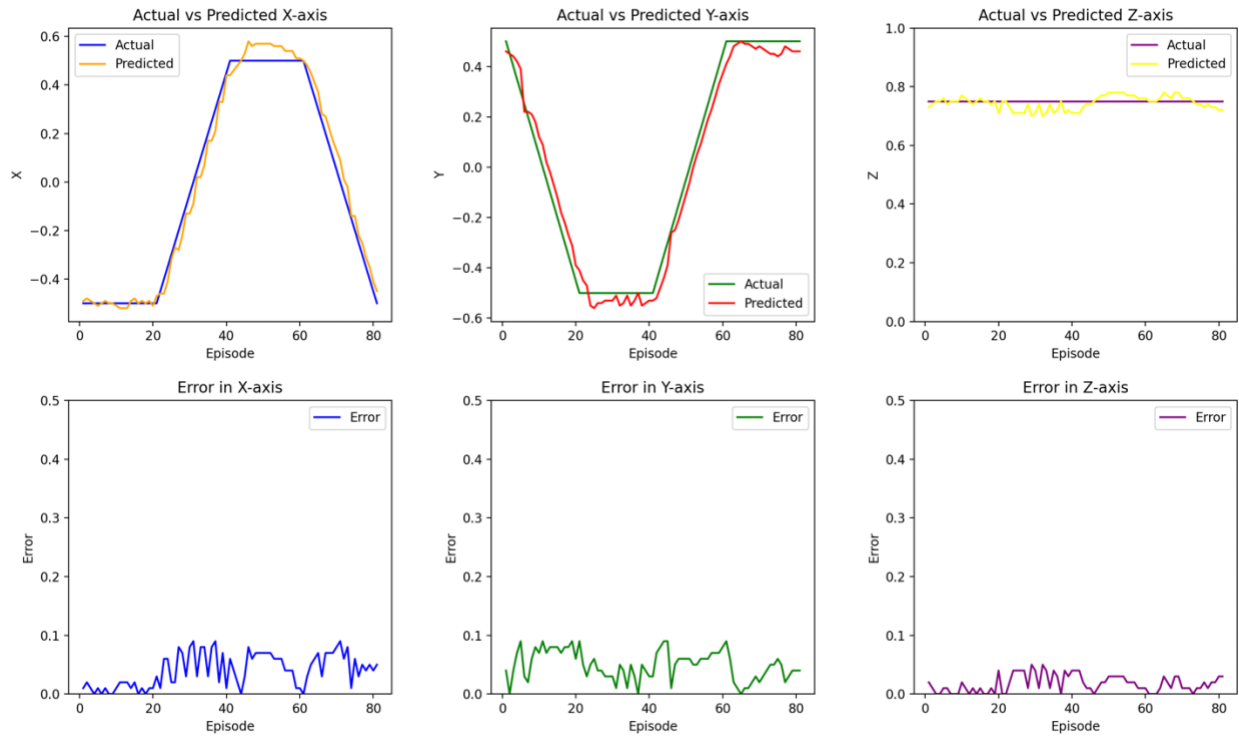


Figure 21: Analysis of trajectory tracking of Rectangle in different axes

3.9.3 Helix

The arm was able to track the following helical trajectory with an accuracy of 100%.

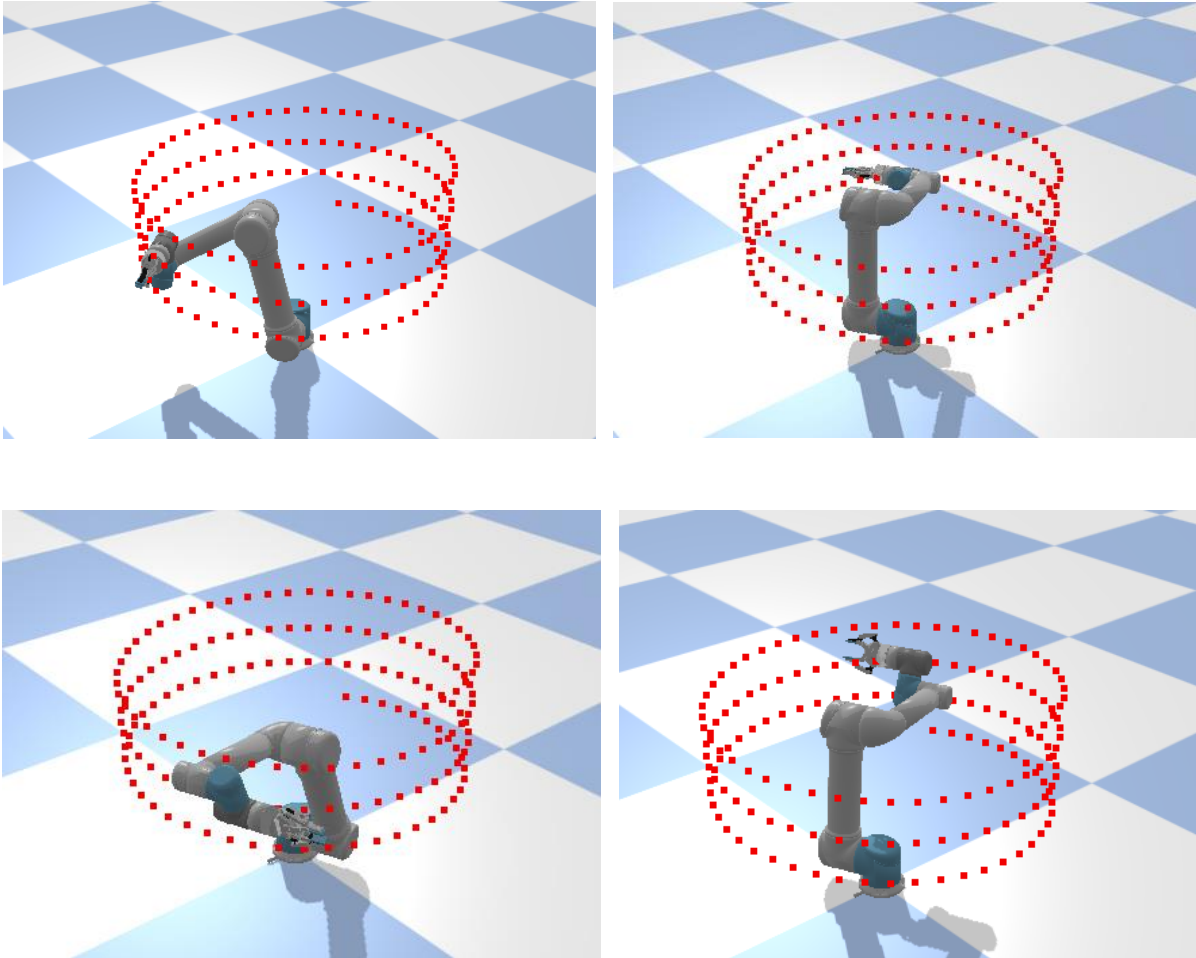


Figure 22: Tracking of a helical trajectory

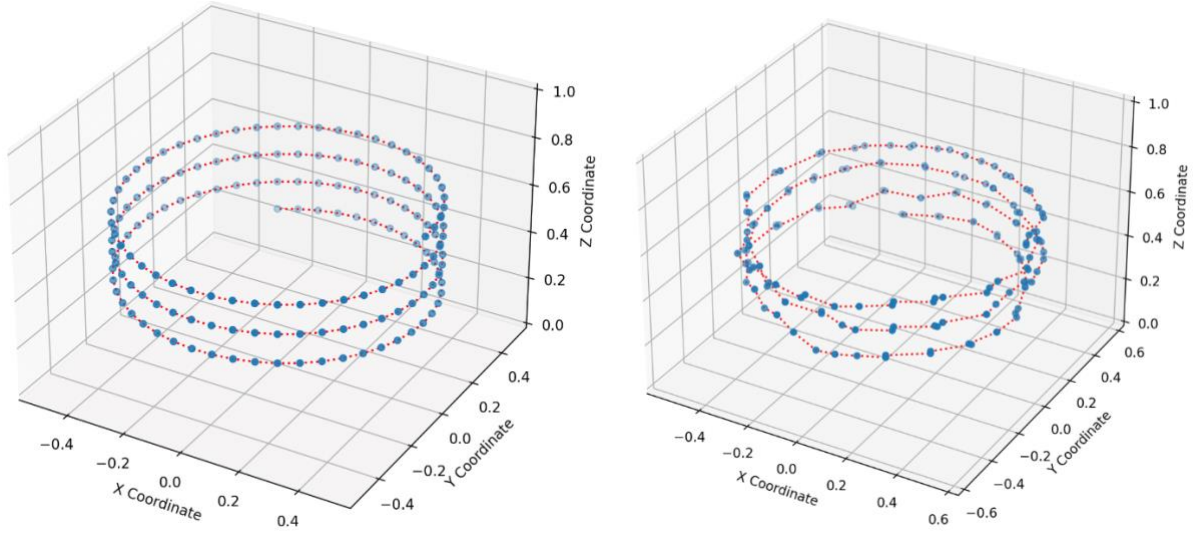


Figure 23: Actual Trajectory (left) and tracked trajectory (right) for helical trajectory

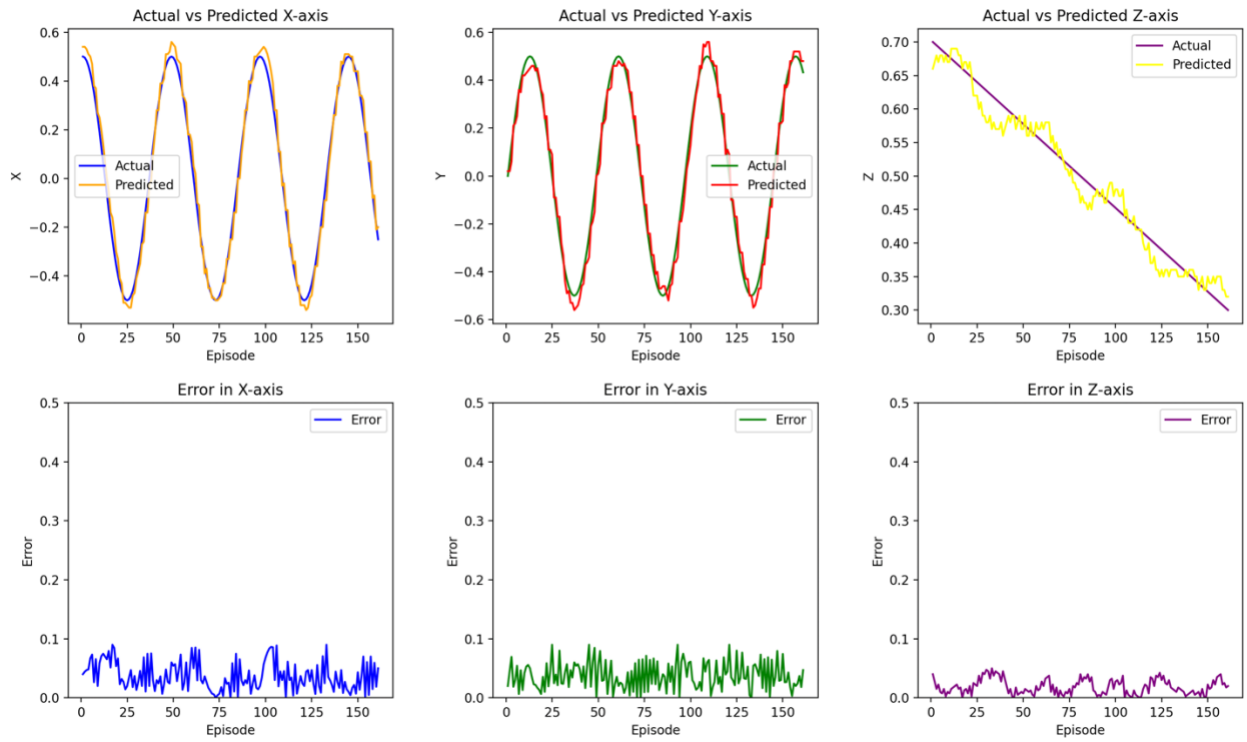


Figure 24: Analysis of trajectory tracking of Helix in different axes.

3.10 Analysis

Model	Crown Accuracy	Rectangle Accuracy	Helix Accuracy
Original	87.8%	83.95%	73.17%
Updated Neural Networks	93.77%	91.57%	83.45%
New Reward Function with Updated Neural Networks	100%	100%	100%

Table 2: Accuracy analysis of different models.

After training the model till 15,000 episodes we tested it on various trajectories, which are demonstrated in the figures above. The model has been able to successfully track all the trajectories given to it with 100% accuracy. Last semester we achieved similar results with 2D trajectories. We had been struggling with 3D trajectories for a while due to its complexity, but we have been able to not only increase the tracking accuracy, but also eliminate any jerky or unwanted movements by modelling the reward function and making the actor and critic neural networks more complex. In the above graphs we can see the exact trajectories tracked by the manipulator, which is close to the original. The tracking error in all 3 directions is also plotted, which is a lot less than before.

This validates our choice of using the DDPG algorithm for trajectory tracking purposes. The successful implementation of the DDPG algorithm has allowed our robotic manipulator to accurately track complex 3D trajectories while maintaining smooth, jerk-free movement. This has the potential to further enhance the performance of robotic manipulators in various industrial applications, paving the way for more efficient and precise automation solutions.

Chapter 4

Conclusion

In this project, we have implemented an off-policy model-free actor-critic based deep reinforcement learning algorithm (DDPG) for trajectory tracking of a robotic manipulator. The end effector (endpoint) of the robot can be moved in any trajectory we specify after the model is trained. The model of the robotic manipulator has been trained in such a way that the end effector can track three-dimensional trajectories. The method keeps the joint angles of the manipulator within the allowed angular range so that unnecessary and abrupt movements can be avoided which can damage the actual robotic manipulator. Smooth movement with minimum number of steps to reach the goal can also be ensured with this actor-critic method by introducing a smoothness reward to the total reward signal of the RL algorithm.

This project demonstrates the successful application of a deep reinforcement learning approach, specifically DDPG, for trajectory tracking in a 3D space using a robotic manipulator. After training the model for 15,000 episodes, we observed a remarkable improvement in both trajectory tracking accuracy and the smoothness of the manipulator's movements.

The high accuracy achieved in tracking various trajectories, as shown in chapter 3, highlights the effectiveness of our approach in addressing the complex problem of 3D trajectory tracking. Furthermore, by carefully designing the reward function, we have managed to eliminate jerky and unwanted movements in the robotic manipulator, resulting in a more precise and smooth motion that is desirable for various practical applications.

This project showcases the potential of deep reinforcement learning techniques in solving real-world robotics problems, particularly in trajectory tracking tasks. The results obtained from this study serve as a strong foundation for future work, including the exploration of more complex environments, the development of more advanced control strategies, and the application of our approach to other robotic systems. The success of our project not only validates the effectiveness of the DDPG algorithm in this context but also encourages further research in the field of deep reinforcement learning for robotics applications.

4.1 Challenges

- Designing a reward function for this robotic manipulator proved to be challenging due to the lack of a fixed methodology. It took numerous trials and iterations to develop the current reward function, with each modification requiring the model to be retrained before assessing its performance. This iterative process was both time-consuming and computationally demanding.
- The training of the model was carried out over multiple simulations to generate an accurate policy. Although numerous failures and sub-optimal outcomes were encountered throughout this process, they were essential for refining the model. The multi-episode training approach demanded considerable time and computational resources.
- Striking the right balance between exploration and exploitation was another challenge. While exploration is necessary for the model to discover optimal strategies, excessive exploration can prolong the training process and increase computational costs. On the other hand, excessive exploitation can result in sub-optimal policies due to insufficient exploration of alternative actions.
- The 6 DOF robotic manipulator necessitated a complex state space, consisting of 31 floating-point numbers, and an action space comprising 4 floating-point numbers. This high-dimensional representation of the problem increased the complexity of the training process. Any adjustments to the action or state space would require retraining the model and possibly modifying other aspects of the model or policy, which would once again be time-consuming and computationally costly.
- The entire training process was conducted using a simulated model, which means that potential hardware-related issues might arise when implementing the model on an actual robotic manipulator. Addressing such issues could involve significant changes to the model and entail revisiting the challenges described above.

4.2 Future Work

1. Noise can be added into a policy to aid in exploration during training. As of now, the model which has been trained did not have any noise added to it. Investigating how much noise is required and training the model with the said noise can yield better results.
2. The current model assumes a fixed environment. As future work, the model could be made more robust and adaptive to handle changing or dynamic environments, such as moving obstacles or varying environmental conditions.
3. The model must be implemented on the actual hardware. The model will be deployed on the robotic manipulator, and changes to the model, if any, will be taken care of accordingly.
4. The current project focuses on trajectory tracking for a robotic manipulator. As future work, the model can be extended to incorporate additional robotic tasks, such as object manipulation, grasping [22], and assembly.

References

- [1] K. Wu, J. Hu, B. Lennox, and F. Arvin, “SDP-based robust formation-containment coordination of swarm robotic systems with input saturation,” *Journal of Intelligent & Robotic Systems*, vol. 102, no. 1, pp. 1–16, 2021.
- [2] H. Niu, Z. Ji, F. Arvin, B. Lennox, H. Yin, and J. Carrasco, “Accelerated sim-to-real deep reinforcement learning: Learning collision avoidance from human player,” in *2021 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2021, pp. 144–149.
- [3] K. Wu, J. Hu, B. Lennox, and F. Arvin, “Finite- time bearing-only formation tracking of heterogeneous mobile robots with collision avoidance,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2021.
- [4] Kim J, Kim SR, Kim SJ and Kim DH. A practical approach for minimum- time trajectory planning for industrial robots. *Industrial Robot: An International Journal*, vol. 37, pp. 51–61, 2010.
- [5] Wenxing Liu, Hanlin Niu, Muhammad Nasiruddin Mahyuddin, Guido Herrmann and Joaquin Carrasco; A Model-free Deep Reinforcement Learning Approach for Robotic Manipulators Path Planning; 2021 21st International Conference on Control, Automation and Systems (ICCAS); 2021
- [6] Zeng R, Liu M, Zhang J, et al. Manipulator Control Method Based on Deep Reinforcement Learning[C]//2020 Chinese Control And Decision Conference (CCDC). IEEE, 2020: 415-420.
- [7] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, Anil Anthony Bharath , “A Brief Survey of Deep Reinforcement Learning” in *IEEE SIGNAL PROCESSING MAGAZINE, SPECIAL ISSUE ON DEEP LEARNING FOR IMAGE UNDERSTANDING*, rXiv:1708.05866v2 [cs.LG] 28 Sep 2017
- [8] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. *Nature* 518, 529–533 (2015). <https://doi.org/10.1038/nature14236>
- [9] Hado van Hasselt, Arthur Guez, David Silver; Deep Reinforcement Learning with Double Q-learning, arXiv:1509.06461 [cs.LG], 2015
- [10] Y. Zhang and M. M. Zavlanos, “Distributed off- policy actor-critic reinforcement learning with policy consensus,” in *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 2019, pp. 4674– 4679.

- [11] K. Wei and B. Ren, "A method on dynamic path planning for robotic manipulator autonomous obstacle avoidance based on an improved RRT algorithm," *Sensors*, vol. 18, no. 2, p. 571, 2018.
- [12] H. Niu, Z. Ji, Z. Zhu, H. Yin, and J. Carrasco, "3d vision-guided pick-and-place using kuka lbr iiwa robot," in *2021 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2021, pp. 592–593.
- [13] Akkaya I, Andrychowicz M, Chociey M, et al. Solving rubik's cube with a robot hand[J]. arXiv preprint arXiv:1910.07113, 2019.
- [14] B. Dasgupta and T. Mruthyunjaya, "Singularity-free path planning for the stewart platform manipulator," *Mechanism and Machine Theory*, vol. 33, no. 6, pp. 711–725, 1998.
- [15] A. A. Maciejewski and C. A. Klein, "Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments," *The international journal of robotics research*, vol. 4, no. 3, pp. 109–117, 1985.
- [16] J. Lander and G. CONTENT, "Making kine more flexible," *Game Developer Magazine*, vol. 1, no. 15- 22, p. 2, 1998.
- [17] R. Mukundan, "A robust inverse kinematics algorithm for animating a joint chain," *International Journal of Computer Applications in Technology*, vol. 34, no. 4, pp. 303–308, 2009.
- [18] H. Zhang, H. Jiang, Y. Luo, and G. Xiao, "Data driven optimal consensus control for discrete-time multi-agent systems with unknown dynamics using reinforcement learning method," *IEEE Transactions on Industrial Electronics*, vol. 64, no. 5, pp. 4091–4100, 2016.
- [19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open-source software*, vol. 3. Kobe, Japan, 2009, p. 5.
- [20] M. Mishra and M. Srivastava, "A view of Artificial Neural Network," *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*, 2014, pp. 1-3, doi: 10.1109/ICAETR.2014.7012785.
- [21] Rodrigo Toro Icarte, Torny Q. Klassen, Richard Valenzano, Sheila A. McIlraith, "Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning", *Journal of Artificial Intelligence Research* 73 (2022) 173-208, arXiv:2010.03950 [cs.LG]
- [22] Sekkat, H.; Tigani, S.; Saadane, R.; Chehri, A. Vision-Based Robotic Arm Control Algorithm Using Deep Reinforcement Learning for Autonomous Objects Grasping, *Appl. Sci.* 2021,11, 7917

trajectory

ORIGINALITY REPORT

7%

SIMILARITY INDEX

5%

INTERNET SOURCES

5%

PUBLICATIONS

3%

STUDENT PAPERS

PRIMARY SOURCES

1

www.coursehero.com

Internet Source

1%

2

"ECAI 2020", IOS Press, 2020

Publication

1%

3

Wenxing Liu, Hanlin Niu, Muhammad Nasiruddin Mahyuddin, Guido Herrmann, Joaquin Carrasco. "A Model-free Deep Reinforcement Learning Approach for Robotic Manipulators Path Planning", 2021 21st International Conference on Control, Automation and Systems (ICCAS), 2021

Publication

1%

4

dtk.tankonyvtar.hu

Internet Source

1%

5

"Neural Information Processing", Springer Science and Business Media LLC, 2018

Publication

<1%

6

Submitted to Indian Institute of Technology Roorkee

Student Paper

<1%

7	Hongsen Peng, Meixia Tao, Tobias Kallehauge, Petar Popovski. "Power Adaptation in URLLC over Parallel Fading Channels in the Finite Blocklength Regime", GLOBECOM 2022 - 2022 IEEE Global Communications Conference, 2022 Publication	<1 %
8	Redwan Alqasemi, Rajiv Dubey. "Maximizing Manipulation Capabilities for People with Disabilities Using a 9-DoF Wheelchair-Mounted Robotic Arm System", 2007 IEEE 10th International Conference on Rehabilitation Robotics, 2007 Publication	<1 %
9	repository.tudelft.nl Internet Source	<1 %
10	dokumen.pub Internet Source	<1 %
11	proceedings.mlr.press Internet Source	<1 %
12	deepai.org Internet Source	<1 %
13	www.grin.com Internet Source	<1 %
14	Submitted to City University of Hong Kong Student Paper	<1 %

15

www.researchgate.net

Internet Source

<1 %

16

Submitted to The University of Manchester

Student Paper

<1 %

17

nccastaff.bmth.ac.uk

Internet Source

<1 %

Exclude quotes On

Exclude matches < 20 words

Exclude bibliography On