Contents

[Introduction 3](#_Toc154616255)

[Historical overview 4](#_Toc154616256)

[Identification of the problem 4](#_Toc154616257)

[Rationale 4](#_Toc154616258)

[Objectives 5](#_Toc154616259)

[Scope and limitations 5](#_Toc154616260)

[Overview of methodology and resources 6](#_Toc154616261)

[Organization of the Study 6](#_Toc154616262)

[Frameworks and Resources 7](#_Toc154616263)

[ROS 7](#_Toc154616264)

[Gazebo 9](#_Toc154616265)

[ESP32 9](#_Toc154616266)

[C++ and Python 10](#_Toc154616267)

[Model design 11](#_Toc154616268)

[Deep Deterministic Reinforcement Learning definition 11](#_Toc154616269)

[Custom MARL with DDPG 12](#_Toc154616270)

[System structure 13](#_Toc154616271)

[States 13](#_Toc154616272)

[Termination state design 14](#_Toc154616273)

[Expected cumulative future reward 14](#_Toc154616274)

[Policy 14](#_Toc154616275)

[*States* 15](#_Toc154616276)

[*Value function* 15](#_Toc154616277)

[*Reward function* 16](#_Toc154616278)

[Simulation 17](#_Toc154616279)

[Bridging Gazebo and ROS2 18](#_Toc154616280)

[Implementation 18](#_Toc154616281)

[Material 18](#_Toc154616282)

[Frameworks 18](#_Toc154616283)

[Communication 18](#_Toc154616284)

[Serial communication 18](#_Toc154616285)

[Reality Gap 19](#_Toc154616286)

[Results 19](#_Toc154616287)

[Analysis and discussion 19](#_Toc154616288)

[Issues 19](#_Toc154616289)

[Integration drift 19](#_Toc154616290)

[Conclusions 19](#_Toc154616291)

[References 20](#_Toc154616292)

[Appendix I 22](#_Toc154616293)

[ESP32 22](#_Toc154616294)

[Quaternions 23](#_Toc154616295)

[Pointer casting 24](#_Toc154616296)

[Appendix II 25](#_Toc154616297)

[CDDPG Agent: 25](#_Toc154616298)

# Introduction

The growing field of Artificial Intelligence over the past decades has profoundly influenced our daily lives, altering the way we communicate with each other, learn, educate, interact with technology and numerous other aspects of life. Being able to teach machines how to learn and make predictions from data with Machine Learning opened many doors to scientists and businesses, enabling unprecedented levels of innovation, efficiency, and personalized customer services experiences across various industries.

One of the fields that has been influenced by AI is robotics, allowing the machine to learn and interact with the outside world through its actuators controlled by input sensory data processed with ML learning algorithms. Robotic Learning is what lies in the intersection of Robotics and Machine learning, and it takes advantage of Reinforcement Learning, a subfield of ML that teaches machines through trial and error allowing robots or agents in videogames to make intelligent decisions in complex environments.

Humans have the ability to learn through a process of trial and error, enabling complex locomotive tasks such as walking down the stairs and talking at the same time. This learning paradigm has already been thoroughly mimicked in virtual environments with digital agents [1], [2]. However, there are still many ongoing challenges regarding real-life scenarios, particularly in the realm of robotics and Reinforcement Learning (RL). When working with real-world scenarios, more things have to be taken into account, such as unpredictable changes [3], resilience and adaptability necessity [4] and safety operations [5], [6], among many others.

Although applying RL to a robot in real life introduces new challenges, it has already been researched [7]. In this context, Multi-Agent Reinforcement Learning [8] becomes an important area of focus. This technique naturally introduces the challenge of not only the need to adapt to the environment but considering the other agent actions. Despite this unique challenge, being able to make multiple robots cooperate with each other to accomplish a task would advance our capabilities in various fields, ranging from autonomous vehicles to collaborative robots in manufacturing, warehouses, and healthcare.

## Historical overview

[ TO BE FILLED]

## Identification of the problem

As previously mentioned, despite the extensive research conducted on RL and MARL in virtual environments and in some cases in real-life scenarios, there are still quite a few challenges opened and waiting to be solved, like a lack of sample efficiency, setting goals and specifying rewards in dynamic environments, generalization to new and different tasks and data collection without human supervision [4].

Among these ongoing challenges, we think that one of the main issues is the **reality gap** that exists when trying to implement a Reinforcement Learning policy (algorithm) learned in a simulation into the corresponding real-life agent. Addressing this particular problem can help solving other issues as well, like diminishing the number of samples needed, saving time, mechanical wear and consequently, money.

## Rationale

The amount of time to invest in this project is very limited, therefore a study [9] has been taken as a reference point. This is to have a benchmark for comparing simulation results, defining the task, and narrowing down the choice of algorithms to be implemented.

This project aims to reduce the reality gap, also called sim-to-real transfer problem, with a specific task involving multiple agents (MARL) and it is divided in three main parts:

1. Simulation of the environment and robots using Gazebo [10] and ROS 2 [11] for control and algorithm implementation.
2. Real implementation of the environment and robots, mimicking as much as possible Gazebo’s simulation, while trying to minimize the reality gap.
3. Discussion, comparison, and analysis of the results.

Improving the sim-to-real-transfer in a muti-agent scenario can help the community prosper and create or improve applications where robots must cooperate with each other to solve a problem.

## Objectives

The academic goals for this project include the implementation of RL algorithms in a simulated environment as well as in a real-case scenario and the construction of two robotic arms that cooperate with each other.

Besides the research goals, improving programming abilities with Python, C, and other languages, getting familiar with software and frameworks such as Linux systems and ROS 2 as well as applying electronics, robotics and AI knowledge learnt while pursuing my bachelor are the main goals of this project. Some questions that I want to answer and may help the reader are the following:

* Is it possible to reduce the reality gap using the existent techniques [12], [13]?
* Can I build two working robots that communicate and cooperate with each other using MARL algorithms?
* How much can the reality-gap can be reduced?

## Scope and limitations

While it would be ideal to fabricate the robot via a manufacturing partner and use generative design techniques, the constrained timeframe of this project regrettably does not allow such approaches.

That is why the scope of this project is building two functional 5 joints robotic arms that interact with each other to successfully perform a task, recreating the environment in a physics simulator and in addition, implement MARL algorithms in both real-case and virtual-based while trying to minimize the reality gap.

Having a limited amount of time and resources also makes certain aspects of the project inevitably limited. This is particularly evident while training the MARL models not only in the simulation but more notably in the real-world application. The challenge extends beyond the initial development of algorithms, encompassing optimization and improvement when receiving new data and feedback.

## Overview of methodology and resources

The study [9] has been chosen as a pipeline for this project, since it provides a specific task for a MARL case scenario application and therefore, the same physics simulator will be used. In this case they used Gazebo which is mor supported in Linux Ubuntu OS. This software will be thoroughly explained during all the project and in further sections.

Python, C++, and Visual Studio Code [14] will be used to develop the models and algorithms due to the richness of libraries and resources they provide [15], [16]. To be able to communicate Python in an organized and efficient way, Robot Operating System 2 (ROS 2) will be used as a bridge. ROS 2 will also be deeply commented on its own section.

Regarding the implementation of the robots, the hardware control will be an ESP32, the PCA9685 to control all servomotors (MG996R) and diverse sensors such as MPU6050 and HSCR04.

## Organization of the Study

# Frameworks and Resources

This project is built upon a custom system where different programming languages, frameworks, and software coexist, communicate, and collaborate. In this chapter, we aim to familiarize the reader with the resources used in the project. The explanation will be detailed but not overly extensive, allowing the reader to understand how the system works and why these specific components are chosen. If additional specific and technical details are required to understand a concept, refer to Appendix I.

Since an overview of the frameworks and resources used can be found in “[*Overview of methodology and resources*](#_Overview_of_methodology)”, below we are going to delve right into each framework description.

## ROS

ROS or Robot Operating System [11] is a useful framework to control robots both in simulation and real life. To avoid confusion, there is a need to clarify that ROS is a set of open-source software frameworks, not an operating system, as one may think at first instance. Before continue explaining ROS features, it needs to be said that in this project ROS 2 [17] is being used, specifically ROS 2 Humble, due to its compatibility with other software of interests such as Gazebo.

The use of ROS 2 in this project comes from benefitting from several powerful features which makes communication between certain software and frameworks more convenient. Among these features, in hierarchical order, **workspaces**, **packages** and **nodes** will be used to organize different parts of the algorithm.

All recent ROS2 versions (Foxy, Galactic, Humble and Iron) rely on workspaces, which is a ROS term for the location of the development space on the system. This is quite convenient as it allows different ROS2 distributions to run on the same computer, switch between them and keep track of all the ongoing changes in a project. In addition, the workspace is organized in packages. Packages offer a controlled way to separate and execute files that are usually stored into the same package if similar functionalities are presented.

Now one may ask, but how can the user control robots or agents and communicate with other software, such as Gazebo, using this software? The answers to this question are **nodes**, **topics,** and **services**.

A **node** is a core concept in ROS 2 and an important element of what is referred to as the “ROS 2 Graph”, which is a network of ROS 2 elements processing data in a collective way at the same time. Each node is and should be responsible for a single, modular purpose, (e.g., controlling joint motors or publishing sensor data from an ultrasonic sensor). Although, how can the nodes communicate and share data between them and other frameworks?

This is where **topics** and **services** come into the system. Topics serve as a communication mechanism within the ROS 2 Graph. Nodes can publish data to a topic, and other nodes can subscribe to that topic to receive and process the information. This publisher-subscriber model facilitates a decentralized and modular architecture, allowing nodes to communicate seamlessly without direct dependencies on each other. Topics are crucial for real-time data exchange in robotic systems, enabling coordination between diverse components, such as sensor data input and motor control commands as explained earlier.

In contrast, services offer a request-response pattern of communication. A node providing a service advertises its availability, and other nodes can send requests to it. The service-providing node then processes the request and sends a response back. This mechanism is particularly useful for scenarios where a specific task needs to be executed on-demand, such as querying a sensor for specific information or requesting a robot to perform a specific action.

A diagram of a service

Description automatically generated

**Figure 1.** Multiple nodes sharing information via topics and services.

A full robotic system, as in this project, contains multiple nodes working in concert. In ROS 2, a single package can contain multiple nodes written in different programming languages such as C++ or Python.

A very useful tool to visualize active nodes, topics and services is *rqt\_graph*. With this command it is possible to see in a graph real time changes and connections between actives nodes in a project. Below, there is an example with several active nodes that share data between them and Gazebo, the simulation software used.

[ADD RQT GRAPH WHEN DDPG AGENT IS RUNNING]

## Gazebo

Since the aim of this project is to minimize the reality gap, an error that is introduced when transitioning from simulations to real-world applications, Gazebo Sim emerges as a core tool. Gazebo, an open-source robot simulation software, plays a crucial role in bridging this gap by offering a realistic and dynamic environment for testing and refining robotic algorithms and control systems before their implementation on physical hardware.

Not only can Gazebo generate accurate 3D simulations but incorporate physics engines that faithfully replicate the dynamics of various robotic platforms. This realistic and dynamic simulation allows researchers to, among other things, train algorithms and test its performance across a wide variety of scenarios, ensuring a more accurate representation of real-world challenges.

In line with the project's goal, Gazebo seamlessly integrates with ROS2. This integration becomes extremely useful when deploying ROS 2 nodes within Gazebo simulations, providing a virtual environment for debugging and refining algorithms before they are executed on physical robots. The symbiotic relationship between Gazebo and ROS 2 enhances the overall capabilities of both platforms.

Furthermore, it is possible to construct multi-robot environments and the interactions between numerous robotic agents, which works perfectly in the current case of a MARL agent. On top of this, the simulation software supports a wide array of sensors, including cameras, lidars, and sonars.

Given these considerations, Gazebo stands out as a highly suitable choice for the current project. Its consistent integration of ROS 2, dynamic simulation capabilities, support for multi-robot (MARL) environments and diverse sensor simulation make this software a valuable tool in narrowing down the reality gap [12].

## ESP32

When it comes to control, processing, and communication between software, hardware, sensors, and actuators, the ESP32 microcontroller stands as one of the best options, if the computational cost is not too high. In this case the algorithm that has to be implemented does require more resources than the ESP32 can provide, that is why other choices such as Raspberry Pi were considered and, if this project is further developed, it is recommended to use one.

Therefore, the ESP32 is used as a middle component between sensors and actuators and a PC. The main computer takes the responsibility of training the model and sending/receiving values to the microcontroller as needed.

In “[Implementation](#_Implementation)” the reader will find how the ESP32 is being used, besides a description of the communications, sensors, and actuators. For further information about ESP32 (e.g., pinouts, features, capabilities) refer to Appendix I (“[ESP32](#_ESP32)” section).

## C++ and Python

Python and C++ play distinct roles in this project. Python, known for its user-friendly design, excels in easing the communication between devices and frameworks. Leveraging its Object-Oriented Programming (OOP) tools, extensive libraries, and concise syntax, Python contributes to the organized and readable development of Deep Learning models. Notably, Python's strengths lie in its ability to create sophisticated models with ease, thanks to powerful Machine Learning libraries such as PyTorch and TensorFlow. These libraries stand out as the primary reasons for choosing Python to develop Deep Reinforcement Learning in this project.

On the other hand, C++ takes the lead when it comes to microcontroller programming, specifically for the ESP32. While C boasts simplicity and computational efficiency, C++ elevates low-level programming with robust OOP features, akin to Python's strengths. The choice of C++ is reinforced by its widespread use, comprehensive documentation, and strong support, establishing it as a reliable option for programming the ESP32.

In summary, the deliberate use of Python and C++ in their specialized roles enhances the project's efficiency. Python's user-friendly design and powerful ML libraries excel in Deep Reinforcement Learning, while C++ proves to be exceptional for microcontroller programming. For detailed and further information on libraries and methods, refer to Appendix I.´

# Model design

To further understand how RL algorithms work, there is a need to first understand the Markov Decision Process. A reinforcement learning agent is designed to make a series of sequential decisions through interactions with its surroundings [18]. This environment is usually structured as an infinite-horizon discounted Markov decision process, or in a simpler way, MDP. See *Definition 1.*

**Definition 1.** A Markov decision process is defined by a tuple (S, A, P , R, γ), where S and A denote the state and action spaces, respectively; P : S × A → ∆(S) denotes the transition probability from any state s ∈ S to any state s’ ∈ S for any given action a ∈ A; R : S × A × S → R is the reward function that determines the immediate reward received by the agent for a transition from (s, a) to s’ ; γ ∈ [0, 1) is the discount factor that trades off the instantaneous and future rewards.

Parallel implementations of single-agent RL scale well on large multi-agent systems although it suffers from issues such as learning stability due to a continuous-changing environment [19] that each agent faces, therefore, to approach this challenge all agents should be jointly trained in a distributed manner.

Moreover, as previously said, one of the project’s goals is to reduce the reality gap from simulation to real-case scenario when applying a MARL algorithm. Since [9] is being taken as a reference to design the model and environment and to avoid stability issues in learning we will be using a custom deterministic policy model called Deep Deterministic Policy Gradient (DDPG), due to its promising results in other related works [20], [21], and its good capabilities dealing with continuous spaces similar to those in this project.

## Deep Deterministic Reinforcement Learning definition

Deep Reinforcement Learning has been successfully applied in [4], [21], [22], mostly in single-agent control problems. When it comes to multiple-agent control, managing the large number of degrees of freedom, heterogeneous physical constraints and partial or asymmetric observations for different robots, demands scalability. That is why DDPG is used.

The DDPG agent can handle many inputs and outputs in its networks. It is also possible to build several agents with different reward functions which are then coupled and aligned to the goal of the cooperative task. DDPG is a multi-agent reinforcement learning policy, and the defined system for a MARL algorithm is, as Markov Decision Process (MDP) dictates, the tuple (1).

(1)

Where:

* The set representing the number of agents.
* The set representing all possible states of the environment.
* The set representing available actions to the agent .
* The state transition function .
* The policy for the agent .
* The accumulative reward of an agent (also called Value function).

In this type of model, each agent aims to maximize the value function with the starting state at time as shown in (2).

(2)

Where:

* is the expected value taken over a random value which is the sum of discounted rewards.
* is the discount factor that weighs the future rewards.
* is the reward received by the agent at the time *.*

While it is true that [9] provides the definition of an algorithm, not code is provided, making it difficult to exactly replicate their model. That is why a custom version of a MARL algorithm using a DDPG policy is created.

## Custom MARL with DDPG

First, it needs to be clear that this model is still under development. That does not mean that it is not working, but that it is still extremely scalable, since its correct completion would solve problems such as generalization, reality gap and other main issues regarding RL nowadays.

Let the custom model be **CDDPG** (Custom Deep Deterministic Policy Gradient) for ease of use and reference. Now, CDDPG is working with one single agent that can control as many robots (or actuators) as the user commands due to its variable input and output network dimensions. To further understand the features of CDDPG, below there’s an explanation about what are the elements of the tuple (1), how are they defined, and how does the system come into place both in simulation and reality.

### System structure

To apply CDDPG in the simulated environment multiple steps will be followed consequently. The first step is to acquire the real-time state of each robot as well as the state of the object to be manipulated. The observed current state () will be sent to the policy () where an algorithm will decide which action to take next. Once the action () is executed, the environment will respond with a new state (). Based on the action’s () effectiveness, the reward function () will provide a reward as a scalar value. Finally, the agent learns from this data (, , and ) and the cycle is repeated.

While the reader can find the definition of these parameters below, how they are obtained is explained in the [Simulation](#_Simulation) and [Implementation](#_Implementation) sections, since each area uses different mechanisms to obtain states, terminal conditions and rewards.

### States

In the reference paper [9], the state set is defined taking into account only the joints angles and the end-effector global position. Because of the limitations of real-world number of sensors and other drawbacks (see [Integration drift](#_Integration_drift) in Appendix I), using only two parameters is a good way to go and a good beginning. It is also true that the more parameters or states are defined, the more precise will be the simulation, with a higher computational cost as well. This may be suitable for other applications although not so much for the current approach since one of the aims of this project is to minimize the reality gap, being the number of sensors in real world very limited due to lack of space, dynamics, number of samples and money expenses.

These reasons point to using the fewer and most critical number of sensors that will become the states of the robots. Therefore, taking [9] as a reference the states set will be defined as (3).

(3)

Consequently, there is a need to find a way to extract each joint angles () and both end-effector global coordinates ().

### Termination state design

The terminal state is a crucial concept that defines the conditions under which an episode concludes. Once the system reaches a terminal state, the ongoing episode ends, and the environment will be reset to its initial state for the start of a new episode. The design of the terminal state is essential for shaping the learning process and achieving specific goals in the training of an agent and can be triggered by the fulfillment of one or many conditions, such as task completion, fatal states, safety concerns, learning process stagnation or run out of time.

Since all conditions expressed before are important enough to reset the simulation if accomplished, the design of the terminal state will be the following state. Let *done* be a Boolean variable triggered by the veracity of (4), (5) or (6).

(4)

(5)

(6)

(7)

In (4), is the summation of each velocity of the joint therefore it becomes *True* when the velocity remains close to 0 for an incremental period indicating that the robot has stopped moving and it has either reached an optimal point or a local minimum. Similarly, equation (5) evaluates to *True* when the summation of the difference of reward values in an incremental period of time (, where is a custom variable for additional reward checks) is 0.

In the Python implementation of both equations (4) and (5), a margin control variable is utilized. This variable ensures that if the velocity or reward condition is in proximity to 0 but not precisely 0, it is still considered true.

Moreover, inequation (6) detects if the orientation pitch, yaw, or roll () of the object is changing too much, leading to undesired object placement, or indicating the object has fallen. In this case is being used as a threshold that can be customized by the user.

Finally, the variable `done` becomes true when any of the conditions is also true, resulting in the termination of the current episode and the initiation of a new one.

### Expected cumulative future reward

Bellman’s equation,

### Policy

DDPG agent (2 main networks Actor and Critic, 2 sub clone networks that stabilize training, replay buffer)

Joint torque values from Actor’s network

Can be found in [CDDPG Agent](#_CDDPG_Agent:) (Appendix II)

### *Value function*

### *Reward function*

*UPDATE REWARD WITH BOTH REWARDS. EXPLAIN 1/X OPTION.*

*Since the reinforcement learning algorithm settings work in a distributed manner, one of the most important things is to correctly define individual reward functions that captures both the specific robot goal and the common task goal.*

*The main constituents defined in [9] for the reward task are: i) those that capture the object displacement from target, (4) and (5) respectively for both robots, and ii) that which captures the object posture deviation (6).*

*(4)*

*(5)*

*(6)*

*Furthermore, two reward function structures are built and tested out, RS-1 and RS-2. On one hand each robot is concerned with both its end effector displacement to the target and the object posture deviation, as we can see in (7). On the other hand, one robot is concerned with the object displacement to the target, while the other is concerned with the object posture deviation, shown in (8).*

*RS-1: (7)*

*RS-2: (8)*

*Where:*

*[ ADD Here 𝑑 (𝑝, 𝑝′ ) = ∥𝑝 − 𝑝 ′ ∥1 characterizes the distance between 𝑝 and 𝑝 ′ , and 𝑎(𝑝 1 , 𝑝2 ) is the absolute angle between the vector (𝑝 1 − 𝑝 2 ) and the target (𝑝 1 𝑡𝑎𝑟𝑔𝑒𝑡 − 𝑝 2 𝑡𝑎𝑟𝑔𝑒𝑡)]*

*The paper results show that RS-2 leads to better performance compared to RS-1, therefore, equation (8) will be the structure used for the reward function in this project, and it is explained in depth below.*

# Simulation

Gazebo [10] will be used as the physics simulator due to the use of this software in [9]. Specifically, Gazebo Fortress will be used, due to its compatibility with the software robot control, ROS 2 Humble. Newer versions of Gazebo and ROS 2 (such as Gazebo Garden and ROS 2 Iron, respectively) have been used, although they have presented so many versions’ incompatibilities and errors that their use was ultimately avoided.

Since [9] do not provide the resources for the simulation, this has been built from scratch, only taking as a reference the environment setup showed in figure \_.



**Figure 3**. Environment setup reference [9]

To be able to represent and run a simulation in Gazebo two things are needed. Firstly, an *.sdf* file that describes the world (physics, models, plugins) and secondly an external software dedicated to controlling the simulation.

As previously said, the control software used is ROS 2, which allows the user to communicate with Gazebo through Python programmable *Nodes*. Nodes are the place where control and receiving and publishing data is happening.

[ … ]

## Simulated observations

Since the nature of the simulated and real-world environments is different, the way states, rewards, and actions are obtained and processed varies considerably. To enhance clarity, the “[*Custom MARL with DDPG*](#_Custom_MARL_with)” section provides explanations and general definitions for these terms, offering a comprehensive understanding of the system in both contexts.

### States

As explained in [States](#_States) section, the states set at a specific time () is defined by the tuple (3). How to obtain each joint angles () and both end-effector global coordinates () from Gazebo simulator is explained below.

(3)

To obtain these values, several components are required from both Gazebo and ROS 2. Firstly, the *PosePublisher* plugin for Gazebo is utilized, which provides the General Coordinates () and a Quaternion (, see *Definition 2*) describing the orientation of each link. Additionally, a ROS 2 node is necessary, subscribing to the topic where Gazebo publishes the *PosePublisher* data. Lastly, a bridge between ROS2 and Gazebo is established using the ros\_gz\_bridge, facilitating data exchange between these systems. Then, assuming that *'segment4\_1'* and *'segment4\_2'* represent the grippers of the respective robots, their global coordinates, and quaternions () can be directly obtained from the simulator.

**Definition 2.** A quaternion is a mathematical concept [23] that extend complex numbers first described by Sir William Rowan Hamilton in 1843. It is composed by four components, one real part and three imaginary parts, and can be written in the form . It is used in different fields, robotics among them, to represent three-dimensional rotations and orientations since it provides certain advantages over other methods such as Euler angles.

In this case quaternions are used instead of Euler notation to avoid ***gimbal lock*** [24] and due to the fact that they are more compact and computationally efficient than rotation matrices.

## Bridging Gazebo and ROS2

# Implementation

## Material

## Frameworks

## Communication

### Serial

Sending the values provided from the DDPG agent to its respective motors is being done through Serial port, due to its proximity to the main computer and the ease of use that both Arduino IDE and Python provide. These values are sent as packed bytes representing floats that will be unpacked when they get to the esp32 using *casting pointer* [25].

Casting pointer allows the user to change the interpretation of the bits in a particular memory location, which is useful when receiving a group of bytes that must be reinterpreted as float values, to be further processed and sent as servo motor angle values. See Annex I to find more information about pointer casting.

### I2C

## Real-world observations

### States

### Rewards

### Actions

# Reality Gap

# Results

# Analysis and discussion

# Issues

## Integration drift

# Conclusions

# References

[1] B. Osiński *et al.*, ‘Simulation-Based Reinforcement Learning for Real-World Autonomous Driving’, in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 6411–6418. doi: 10.1109/ICRA40945.2020.9196730.

[2] D. Silver, ‘Reinforcement Learning and Simulation-Based Search in Computer Go’, ERA. Accessed: Oct. 29, 2023. [Online]. Available: https://era.library.ualberta.ca/items/12fc0cb8-c990-4759-8fa1-5e12af33116a

[3] S. Padakandla, ‘A Survey of Reinforcement Learning Algorithms for Dynamically Varying Environments’, *ACM Comput. Surv.*, vol. 54, no. 6, p. 127:1-127:25, Jul. 2021, doi: 10.1145/3459991.

[4] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, ‘How to Train Your Robot with Deep Reinforcement Learning; Lessons We’ve Learned’, *The International Journal of Robotics Research*, vol. 40, no. 4–5, pp. 698–721, Apr. 2021, doi: 10.1177/0278364920987859.

[5] J. Garcıa and F. Fernandez, ‘A Comprehensive Survey on Safe Reinforcement Learning’.

[6] S. Gu *et al.*, ‘A Review of Safe Reinforcement Learning: Methods, Theory and Applications’. arXiv, Feb. 20, 2023. Accessed: Oct. 02, 2023. [Online]. Available: http://arxiv.org/abs/2205.10330

[7] X. Huang *et al.*, ‘Creating a Dynamic Quadrupedal Robotic Goalkeeper with Reinforcement Learning’. arXiv, Oct. 10, 2022. Accessed: Sep. 26, 2023. [Online]. Available: http://arxiv.org/abs/2210.04435

[8] L. Busoniu, R. Babuska, and B. De Schutter, ‘A Comprehensive Survey of Multiagent Reinforcement Learning’, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, Mar. 2008, doi: 10.1109/TSMCC.2007.913919.

[9] G. Ding *et al.*, ‘Distributed Reinforcement Learning for Cooperative Multi-Robot Object Manipulation’.

[10] N. Koenig and A. Howard, ‘Design and use paradigms for gazebo, an open-source multi-robot simulator’, *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, pp. 2149–2154, 2004, doi: 10.1109/IROS.2004.1389727.

[11] A. Koubaa, Ed., *Robot Operating System (ROS)*, vol. 625. in Studies in Computational Intelligence, vol. 625. Cham: Springer International Publishing, 2016. doi: 10.1007/978-3-319-26054-9.

[12] J. Tan *et al.*, ‘Sim-to-Real: Learning Agile Locomotion For Quadruped Robots’. arXiv, May 16, 2018. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1804.10332

[13] W. Yu, J. Tan, Y. Bai, E. Coumans, and S. Ha, ‘Learning Fast Adaptation with Meta Strategy Optimization’. arXiv, Feb. 15, 2020. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1909.12995

[14] ‘Visual Studio Code’.

[15] G. Brockman *et al.*, ‘OpenAI Gym’. arXiv, Jun. 05, 2016. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1606.01540

[16] T. Developers, ‘TensorFlow’, *Zenodo*, May 2021, doi: 10.5281/zenodo.4758419.

[17] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, ‘Robot Operating System 2: Design, architecture, and uses in the wild’, *Science Robotics*, vol. 7, no. 66, p. eabm6074, May 2022, doi: 10.1126/scirobotics.abm6074.

[18] K. Zhang, Z. Yang, and T. Başar, ‘Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms’, in *Handbook of Reinforcement Learning and Control*, K. G. Vamvoudakis, Y. Wan, F. L. Lewis, and D. Cansever, Eds., in Studies in Systems, Decision and Control. , Cham: Springer International Publishing, 2021, pp. 321–384. doi: 10.1007/978-3-030-60990-0\_12.

[19] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, ‘A survey and critique of multiagent deep reinforcement learning’, *Auton Agent Multi-Agent Syst*, vol. 33, no. 6, pp. 750–797, Nov. 2019, doi: 10.1007/s10458-019-09421-1.

[20] N. Casas, ‘Deep Deterministic Policy Gradient for Urban Traffic Light Control’. arXiv, Aug. 02, 2017. Accessed: Nov. 15, 2023. [Online]. Available: http://arxiv.org/abs/1703.09035

[21] S. Amarjyoti, ‘Deep Reinforcement Learning for Robotic Manipulation-The state of the art’. arXiv, Jan. 30, 2017. Accessed: Nov. 08, 2023. [Online]. Available: http://arxiv.org/abs/1701.08878

[22] I. V. Serban *et al.*, ‘A Deep Reinforcement Learning Chatbot’, *arXiv:1709.02349 [cs, stat]*, Nov. 2017, Accessed: Jul. 01, 2021. [Online]. Available: http://arxiv.org/abs/1709.02349

[23] J. Voight, *Quaternion Algebras*. Springer Nature, 2021. doi: 10.1007/978-3-030-56694-4.

[24] E. G. Hemingway and O. M. O’Reilly, ‘Perspectives on Euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments’, *Multibody Syst Dyn*, vol. 44, no. 1, pp. 31–56, Sep. 2018, doi: 10.1007/s11044-018-9620-0.

[25] S. H. Yong, S. Horwitz, and T. Reps, ‘Pointer analysis for programs with structures and casting’, *SIGPLAN Not.*, vol. 34, no. 5, pp. 91–103, May 1999, doi: 10.1145/301631.301647.

[26] ‘esp32-wroom-32\_datasheet\_en.pdf’. Accessed: Dec. 18, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32\_datasheet\_en.pdf

# Appendix I

## ESP32



**Figure 2.** ESP-Wroom-32 pinout

The specific model being used is ESP-Wroom-32 [26], with the above pinout. The main features used in the project are listed below:

* Wi-Fi and Bluetooth modules.
* 24 GPIO

Having a good understanding of the ESP32s’ PINOUT and capabilities is essential to create a working system that puts together sensors, actuators, and deep learning models that is why is recommended to refer to Appendix I for more information.

## Quaternions

## Pointer casting

Casting a pointer allows the user to reinterpret the content of a specific memory location, altering its interpretation without changing the actual data or its location. This is particularly useful when receiving a stream of bytes that needs to be reinterpreted as a different data type, such as converting a sequence of bytes into float values.

In the context of this project, pointer casting is employed to instruct the compiler to treat a block of bytes as a float array, enabling the conversion of raw byte data received from the serial port into meaningful float values. This is essential for subsequent processing, such as mapping these float values to servo motor angles for control.

-+----+----+----+----+----+----+-

| | | | | | |

-+----+----+----+----+----+----+-

^~~~~~~~

| byte array

d

```

After Pointer Casting:

-+----+----+----+----+----+----+-

| f1 | f2 | f3 | f4 | f5 | f6 |

-+----+----+----+----+----+----+-

^~~~~~~~~

| float array

D

# Appendix II

## CDDPG Agent:

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

from sub\_modules.rbuffer import ReplayBuffer

#SECTION - POLICY DRL ALGORITHM -

#SECTION - ACTOR NETWORK

class Actor(nn.Module):

    def \_\_init\_\_(self, state\_dim, action\_dim, actor\_dropout\_p):

        super(Actor, self).\_\_init\_\_()

        self.dropout = nn.Dropout(p=actor\_dropout\_p)

        self.fc1 = nn.Linear(state\_dim, 256)

        print(self.fc1)

        self.fc2 = nn.Linear(256, 128)

        self.fc3 = nn.Linear(128, 64)

        self.fc4 = nn.Linear(64, action\_dim)

    def forward(self, state):

        x = F.relu(self.dropout(self.fc1(state)))

        x = F.relu(self.dropout(self.fc2(x)))

        x = F.relu(self.dropout(self.fc3(x)))

        action = torch.tanh(self.fc4(x)) # normalise [-1, 1]

        return action

#!SECTION

#SECTION - CRITIC NETWORK

class Critic(nn.Module):

    def \_\_init\_\_(self, state\_dim, action\_dim, critic\_dropout\_p):

        super(Critic, self).\_\_init\_\_()

        self.dropout = nn.Dropout(p=critic\_dropout\_p)

        self.fc1 = nn.Linear(state\_dim + action\_dim, 256)

        self.fc2 = nn.Linear(256, 128)

        self.fc3 = nn.Linear(128, 64)

        self.fc4 = nn.Linear(64, 1)

    def forward(self, state, action):

        x = torch.cat([state, action], dim=1)

        x = F.relu(self.dropout(self.fc1(x)))

        x = F.relu(self.dropout(self.fc2(x)))

        x = F.relu(self.dropout(self.fc3(x)))

        value = self.fc4(x) # Estimated Q-Value for a given state-action pair

        return value

#!SECTION

#SECTION - DDPG AGENT

class DDPGAgent:

    def \_\_init\_\_(self, state\_dim, action\_dim, buffer\_size = 10000):

        self.actor\_lr = 0.5e-4

        self.critic\_lr = 1e-4

        self.discount\_factor = 0.95

        self.soft\_update\_rate = 0.01

        self.actor\_dropout\_p = 0.5

        self.critic\_dropout\_p = 0.5

        self.batch\_size = 64

        self.replay\_bufer = ReplayBuffer(buffer\_size)

        self.actor\_losses = []

        self.critic\_losses = []

        self.actor = Actor(state\_dim, action\_dim, self.actor\_dropout\_p)

        self.actor\_target = Actor(state\_dim, action\_dim, self.actor\_dropout\_p) # Has the same architecture as the main actor network but it's updated slowly --> provides training stability

        self.actor\_target.load\_state\_dict(self.actor.state\_dict()) # Get parameters from main actor network and synchronize with acto\_target

        self.critic = Critic(state\_dim, action\_dim, self.critic\_dropout\_p)

        self.critic\_target = Critic(state\_dim, action\_dim, self.critic\_dropout\_p)

        self.critic\_target.load\_state\_dict(self.critic.state\_dict())

        self.actor\_optimizer = optim.Adam(self.actor.parameters(), lr=self.actor\_lr)

        self.critic\_optimizer = optim.Adam(self.critic.parameters(), lr=self.critic\_lr)

    #SECTION - Select action

    def select\_action(self, state):

        state = torch.FloatTensor(state)

        action = self.actor(state)

        # remove gradients from tensor and convert it to numpy array

        return action.detach().numpy()

    #SECTION - Update

    def update(self, state, action, reward, next\_state, terminal\_condition):

        # Add the real-time experience to the replay buffer

        self.replay\_bufer.add((state,

                               action,

                               reward,

                               next\_state,

                               terminal\_condition)

        )

        # Sample a batch from the replay buffer

        batch\_size = self.batch\_size

        buffer\_batch = self.replay\_bufer.sample(batch\_size)

        # Unpacking buffer\_batch into separate lists for each variable

        buffer\_states, buffer\_actions, buffer\_rewards, buffer\_next\_states, buffer\_terminal\_condition = zip(\*buffer\_batch)

        # Convert lists to NumPy arrays for efficency

        buffer\_states = np.array(buffer\_states)

        buffer\_actions = np.array(buffer\_actions)

        buffer\_rewards = np.array(buffer\_rewards).reshape(-1, 1)

        buffer\_next\_states = np.array(buffer\_next\_states)

        buffer\_terminal\_condition = np.array(buffer\_terminal\_condition).reshape(-1, 1)

        # Convert lists to PyTorch tensors

        buffer\_states = torch.FloatTensor(buffer\_states)

        buffer\_actions = torch.FloatTensor(buffer\_actions)

        buffer\_rewards = torch.FloatTensor(buffer\_rewards)

        buffer\_next\_states = torch.FloatTensor(buffer\_next\_states)

        buffer\_terminal\_condition = torch.FloatTensor(buffer\_terminal\_condition)

        # Buffer data

        buffer\_values = self.critic(buffer\_states, buffer\_actions)

        buffer\_next\_actions = self.actor\_target(buffer\_next\_states)

        buffer\_next\_values = self.critic\_target(buffer\_next\_states, buffer\_next\_actions.detach())

        # BELLMAN EQUATION

        buffer\_target\_values = buffer\_rewards + self.discount\_factor \* buffer\_next\_values \* (1 - buffer\_terminal\_condition)

        # Critic loss for buffer data

        critic\_loss = F.mse\_loss(buffer\_values, buffer\_target\_values)

        # Actor loss for buffer data

        actor\_loss = -self.critic(buffer\_states, self.actor(buffer\_states)).mean()

        # Append losses to the history

        self.actor\_losses.append(actor\_loss.item())

        self.critic\_losses.append(critic\_loss.item())

        # Update networks

        self.actor\_optimizer.zero\_grad()

        actor\_loss.backward()

        self.actor\_optimizer.step()

        self.critic\_optimizer.zero\_grad()

        critic\_loss.backward()

        self.critic\_optimizer.step()

        # Update target networks with soft updates

        self.soft\_update(self.actor, self.actor\_target, self.soft\_update\_rate)

        self.soft\_update(self.critic, self.critic\_target, self.soft\_update\_rate)

    def soft\_update(self, local\_model, target\_model, tau):

        for target\_param, local\_param in zip(target\_model.parameters(), local\_model.parameters()):

            target\_param.data.copy\_((1.0 - tau) \* target\_param.data + tau \* local\_param.data)

#!SECTION

#!SECTION