ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Intelligent Robotic Systems

MULTI-AGENT DEEP REINFORCEMENT LEARNING FOR DRONE SWARMS IN STATIC AND DYNAMIC ENVIRONMENTS

CANDIDATE
Guido Laudenzi

SUPERVISOR
Prof. Andrea Roli

CO-SUPERVISOR
Prof. Akiya Kamimura

Academic year 2022-2023

Session 3rd

Abstract

The application of robotics, particularly drone swarms, in operational settings presents a frontier in leveraging collective intelligence for complex spatial tasks. While Deep Reinforcement Learning (DRL) has significantly advanced the autonomous control of wheeled robots in static, 2D spaces, the adaptation to flying drones navigating 3D and dynamic environments remains inadequately documented, requiring further exploration and characterization. This gap underscores the critical need for developing a new foundation of research, aimed at creating new sophisticated interaction models and intelligent agents to facilitate collaborative navigation and obstacle avoidance.

While single-agent control has been throughly documented both using reinforcement learning and supervised learning as Visual SLAM, multi-agent control is still an open research field. The state-of the art approaches rely on Optimization-Based Motion Planning, which consists in employing pre-programmed constraints (local behavior rules and control algorithms) shared among agents with limited learning capabilities. Furthermore, optimal motion planning falls in highly dynamic environments and different tasks that were not explicitly pre-programmed.

To overcome the limitations of prior methodologies, this thesis introduces a novel DRL application for drone swarm path planning in both static and dynamic environments, with an emphasis on obstacle avoidance. The core objective of this research is to showcase the drones' ability to autonomously learn an optimal trajectory in any given environment, achieving remarkable efficiency. This study underscores the potential of DRL in revolutionizing drone swarm navigation and enhancing their possible applicability in real-world scenarios, such as industrial cooperation or search and rescue.

Applying a simple unified model across varying scenarios, this study demonstrates the adaptability and generalization capabilities of the proposed DRL algorithm. The methodology also include employing Curriculum Learning as a technique to incrementally introduce complexity, thereby facilitating the drones' ability to navigate through dynamically changing conditions. Incorporated in a Proximal Policy Optimization (PPO) algorithm, the combination of a state encoding through convolutional neural networks of a simple observation space just provided by a RGB camera with drone's position and the correct reward function which includes information as reaching the goal, colliding or minimum interdistance between agents, enables the drones to learn and exhibit emer-

gent collective behaviors, addressing intrinsic challenges such as agent loss and mutual avoidance while training in a simulated environment.

The findings suggest that the proposed DRL model not only achieves training convergence and near-optimal trajectories in different maps but also presents a scalable solution for the control of drone swarms composed of varying numbers of agents. This research contributes to the growing body of knowledge by providing a viable alternative to supervised learning and classical control theory, challenging the current state-of-the-art in drone navigation. However, limitations such as computational demands and the need for extensive training data highlight areas for future improvement. Future developments may explore the integration of more efficient learning algorithms and more sophisticated representations of the triples state-action-reward to enhance the model's adaptability, while pursuing real-life implementation.

Abstract - Italian version

L'applicazione della robotica, in particolare degli sciami di droni, in contesti operativi rappresenta una frontiera nello sfruttamento dell'intelligenza collettiva per compiti spaziali complessi. Mentre il Deep Reinforcement Learning (DRL) ha fatto avanzare significativamente lo stato dell'arte per il controllo autonomo dei robot su ruote in spazi statici 2D, l'adattamento ai droni volanti che navigano in ambienti 3D e dinamici rimane inadeguatamente documentato, richiedendo ulteriori esplorazioni e caratterizzazioni. Questo divario sottolinea la necessità fondamentale di sviluppare una nuova base di ricerca, volta a creare nuovi modelli di interazione sofisticati e agenti intelligenti per facilitare la navigazione collaborativa e l'elusione degli ostacoli.

Mentre il controllo di un singolo agente è stato ampiamente documentato sia utilizzando il Reinforcement Learning che l'apprendimento supervisionato come con l'utilizzo di Visual SLAM, il controllo di più agenti è ancora un ambito di ricerca aperto. Gli approcci dello stato dell'arte si basano sull'Optimization-Based Motion Planning, che consiste nell'utilizzare vincoli pre-programmati (regole di comportamento locale e algoritmi di controllo) condivisi tra agenti con capacità di apprendimento limitate. Inoltre, questi metodi falliscono in ambienti altamente dinamici e compiti diversi che non sono stati esplicitamente preprogrammati.

Per superare i limiti delle metodologie precedenti, questa tesi introduce una nuova applicazione DRL per la pianificazione del percorso di sciami di droni sia in ambienti statici che dinamici, con un'enfasi sull'evitamento degli ostacoli. L'obiettivo principale di questa ricerca è mostrare la capacità dei droni di apprendere autonomamente una traiettoria ottimale in qualsiasi ambiente, ottenendo una notevole efficienza. Questo studio sottolinea il potenziale dei DRL nel rivoluzionare la navigazione degli sciami di droni e nel migliorare la loro possibile applicabilità in scenari del mondo reale, come la cooperazione industriale o la ricerca e salvataggio.

Applicando un semplice modello unificato a diversi scenari, questo studio dimostra le capacità di adattabilità e generalizzazione dell'algoritmo DRL proposto. La metodologia include anche l'utilizzo del Curriculum Learning come tecnica per introdurre in modo incrementale la complessità, facilitando così la capacità dei droni di navigare attraverso condizioni che cambiano dinamicamente. Incorporato in un algoritmo di Proximal Policy Optimization (PPO), la combinazione di una codifica dello stato attraverso reti neurali convoluzionali di un semplice spazio di osservazione

fornito semplicemente da una telecamera RGB con la posizione del drone e la corretta funzione di ricompensa che include informazioni come raggiungimento dell'obiettivo, collisione o un'interdistanza minima tra gli agenti, consente ai droni di apprendere ed esibire comportamenti collettivi emergenti, affrontando sfide intrinseche come la perdita di agenti e l'evitamento reciproco durante l'addestramento in un ambiente simulato.

I risultati suggeriscono che il modello DRL proposto non solo raggiunge una convergenza di addestramento e traiettorie quasi ottimali in diverse mappe, ma presenta anche una soluzione scalabile per il controllo di sciami di droni composti da un numero variabile di agenti. Questa ricerca contribuisce al crescente corpus di conoscenze fornendo una valida alternativa all'apprendimento supervisionato e alla classica teoria dei controlli automatici, sfidando l'attuale stato dell'arte nella navigazione con droni. Tuttavia, limitazioni come la richiesta computazionale e la necessità di dati dati dalle simulazioni, evidenziano aree di miglioramento futuro. Gli sviluppi futuri potrebbero esplorare l'integrazione di algoritmi di apprendimento più efficienti e rappresentazioni più sofisticate della tripla stato-azione-ricompensa per migliorare l'adattabilità del modello, perseguendo al contempo l'implementazione hardware dello stesso.

Acknowledgments

As I stand on the cusp of completion, reflecting on the whirlwind journey of this thesis, I find myself immensely grateful to those who have supported and inspired me along the way.

Firstly, I extend my sincerest appreciation to the University of Bologna for providing the funding that allowed me to undertake my internship in Japan. This opportunity has been a fundamental experience in my academic and personal growth.

To my university supervisor, Andrea Roli, your guidance and encouragement have been invaluable throughout this process. To my supervisor in Japan, Akiya Kamimura, for opening their doors to me and entrusting me with meaningful work. To the researchers whose meticulous work served as the foundation of my research.

To my family, both the established and the newest member, friends, and the countless individuals who have crossed my path, your unwavering support and encouragement have sustained me through the challenges and triumphs of this journey.

To a person who, though no longer part of my life, has left an indelible mark on my journey, thank you for your past support and encouragement.

Lastly, to myself—amidst the trials and tribulations, I persevered. This work stands as a testament to my resilience, determination, and passion for the subject matter.

In the words of Alan Turing,

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

May this thesis contribute, albeit modestly, to the ongoing advancement of Artificial Intelligence.

Contents

List of Acronyms

RL Reinforcement Learning

DRL Deep Reinforcement Learning

AI Artificial Intelligence

ML Machine Learning

CS Computer Science

DNN Deep Neural Networks

DQN Deep Q-Network

PPO Proximal Policy Optimization

SOTA State-of-the-art

DDPG Deep Deterministic Policy Gradient

TPRO Trust Region Policy Optimization

A2C Advantage Actor Critic

HER Hindsight Experience Replay

SAC Soft Actor-Critic

TD3 Twin Delayed DDPG

MADDPG Multi-agent DDPG

MAPPO Multi-agent PPO

DP Dynamic Programming

VO Visual Odometry

RNN Recurrent Neural Network

CNN Convolutional Neural Network

UAV Unmanned Aerial Vehicle

PID Proportional-Integral-Derivative

MC Monte Carlo

TD Temporal Difference

UCB Upper-Confidence-Bound

MDP Markov Decision Process

TD Temporal-Difference

SGD Stochastic Gradient Descent

MARL Multi-Agent Reinforcement Learning

POMG Partially Observable Markov Game

Dec-POMDP Decentralized Partially Observable Markov Decision Process

COMA Counterfactual Multi-Agent

ROS Robot Operating System

URDF Unified Robot Description Format

FOV Field of View

AEC Agent Environment Cycle

ReLU Rectified Linear Unit

WSL Windows Subsystem for Linux

SITL Software in the Loop

HITL Hardware in the Loop

List of Figures

1 Introduction

Contents

This chapter aims to provide a general idea of the topic discussed in the thesis. In section 1.1, a motivation for the use of artificial intelligence agents and algorithms for the path planning of mobile robots and drones is given, as opposed to the standard control theory. In section 1.2, the most relevant related work for Deep Reinforcement Learning (DRL) in robotic applications is covered, together with the most innovative algorithm for training agents. Section 1.3 serves as a connection between Reinforcement Learning (RL) and its inspiration given by human psychology. Finally, section 1.4 directs attention to the agents' properties and dynamics which will be considered for the research.

1.1 Motivation

Artificial Intelligence (AI) field is getting more important day by day. It enables human capabilities to software, such as reasoning, planning, perception, and recognition, in an increasingly effective, efficient, and at low-cost manner.

General tasks, including finding patterns in data, playing games, planning, and image recognition, that have been performed by software for many years, can also be performed using AI. Outstanding products using AI include autonomous vehicles, automated medical diagnosis, human-machine interaction using voice or video inputs, and decision-making.
One important branch of AI and Computer Science (CS) is Machine Learning (ML): it aims at the use of data and algorithms to find common patterns as a human would guess. Its mathematical foundations are provided by mathematical optimization methods. ML can be further divided into different approaches, depending on the data that must be handled.

Supervised learning has the main purpose of finding a mathematical model for a data set containing examples with both the inputs and the desired labels, or outputs. Using an objective function, these algorithms learn a function that can be used to predict the output associated with new examples. The function that will allow to find correct output for new examples is said to be optimal, or well-trained. The main split of supervised learning is about classification and regression: when outputs can only be a limited set of values, we are talking about classification; when they are within a numerical range, regression.

Unsupervised learning does not take into account labels but just input data. In this case, the aim is to form groups, or clusters, of examples that are considered similar according to a certain function. This approach is more statistical-oriented given that it is usually about finding a probability density function that suits well for the data in the study.

Reinforcement learning is not similar to supervised learning: does not rely on a set of labeled data; nor unsupervised: there are no labels, but here an intelligent agent(s) seeks to maximize a reward. The agent must determine the correct actions to take in various scenarios, described by a state. Each action is followed by a reward, which indicates how good it was. This approach has a biological inspiration (humans seem to learn similarly, see section 1.3) and it is also one with a broader scenario for application: it is studied in a multitude of fields, such as game theory, control theory, operations research, multi-agent systems, and swarm intelligence.
Because of its innate nature, lately, it has been the topic of many robotic researchers aiming to use it as a controller for autonomous vehicles, intelligent swarms, robots, and drones, as opposed to the standard control theory. There is an underlying connection between RL and control theory: their goals are similar; both methods aim to determine the correct inputs fed into a system that

will generate the desired behavior. So, the focus is on figuring out how to design the policy (or the controller) that maps the observed state of the environment (or the plant) to the best actions (the actuator commands). The feedback signal is the observations from the environment, and the reference signal is built into the reward function.

1.2 Related work

1.2.1 History

The field related to RL is improving at an extreme pace: hundreds of papers, including new possible implementations and algorithms, are published every year. Therefore, keeping up with the state-of-the-art is a challenging task. Anyway, its popularity is thanks to the many successful discoveries that happened just a few years ago.

A game-changing discovery was during 2013, when Deep Neural Networks (DNN) was first combined with RL and achieved impressive results. DeepMind group developed a training algorithm, namely Deep Q-Network (DQN), for an agent that was able to play several Atari video games with a level comparable to humans [1]. Its workflow was very simple: using raw RGB input images of each game, an action was chosen among several discrete actions, while the score of the game was used as a reward. DQN uses a Neural Network as a function approximator.

Again, in 2016, the DeepMing group developed AlphaGo [2], which is a DRL agent that also uses a heuristic search approach known as Monte Carlo Tree Search [3]. AlphaGo was able to beat the world champion in the Go board game. The two-stage training provided the outstanding result: firstly, the agent was trained with supervised learning thanks to a set of data containing recorded amateur games; secondly, the agent played against itself with an RL algorithm consisting of a tree search guided by a "value network" and a "policy network".

While the previous publications focus more on game-solving, many others focus on searching for better algorithms for training agents. An important example is given by Proximal Policy Optimization (PPO), introduced by OpenAI in 2017 [4]: PPO is a policy gradient method that efficiently optimizes policies with multiple epochs of minibatch updates. It addresses the instability issues associated with earlier policy gradient methods and has become a popular choice for training RL agents.

What has been mentioned so far is just a tiny amount of the RL field. RL has a much older origin: an important split that was recognized during the many years of research is the one that compares Value-Based and Policy-Based methods. The first aims to approximate a value function, a function that determines the value (or reward) for each action a in a known state s. What decides which action to take is a policy π, a mapping from the state to a probability distribution of the actions to be taken. An example of a Value-Based algorithm is DQN; it has been intensively investigated and many improvements have been proven: the most important is called *Rainbow* DQN [5]. One of its downsides is that it only supports discrete actions, while many real-world problems need a continuous representation.
On the other hand, in Policy-Based methods, the policy is learned directly, resulting in a more stable convergence versus a maximum. They still make use of a value function to learn the policy using an Actor-Critic architecture: the actor decides which action should be taken and the critic gives an indication to the actor on how good the action was and how it should be adjusted (by computing a value function). The origin of these methods is old, with the 1992's REINFORCE algorithm [6]. It was the first algorithm to learn stochastic policies by applying gradient-based methods. Today's publications are more focused on Policy-Based methods and most of the State-of-the-art (SOTA) algorithms belong to this class: most known and used ones are, in chronological order, Deep Deterministic Policy Gradient (DDPG) [7], Trust Region Policy Optimization (TPRO) [8], Advantage Actor Critic (A2C) [9], Proximal Policy Optimization (PPO) [4], Hindsight Experience Replay (HER) [10], Soft Actor-Critic (SAC) [11], Twin Delayed DDPG (TD3) [12]; many variants have been developed.

1.2.2 Multi-agent RL

RL algorithms can be applied to multi-agent problems; algorithms' variants have two main differences that can be generally recognized. First of all, agents should have a fixed behavior with respect to each other: we can define *cooperative games* when agents try to maximize a common reward; *competitive games* when each agent has its reward and its chosen action can go against the other agents' [13]; mixed behaviors can be implemented. A second difference is algorithm-oriented: it is possible to train each agent with an independent architecture (nowadays, mostly Actor-Critic), or have a centralized architecture that both predicts and judges the next actions of each agent. An early publication from 1993 compared the two algorithmic approaches, showing that agents engaging in partnership can significantly outperform independent agents although they may learn

slowly, in the beginning, [14]. This early research was the starting point for the extension of multi-agent settings algorithms, such as the introduction of Multi-agent DDPG (MADDPG) algorithm, an adaptation of Actor-Critic methods that considers action policies of other agents and can successfully learn policies that require complex multi-agent coordination [15]. Other important algorithms comprehend the recent QMIX, a method that can train decentralized policies in a centralized end-to-end fashion, employing a network that estimates joint action values as a complex non-linear combination of values [16]. On the other hand, single-agent algorithms were proved to be successfully extended to multi-agent environments, achieving outstanding results, as the PPO, namely Multi-agent PPO (MAPPO) [17]. This last algorithm will be the one used for the present research.

1.2.3 RL in Robotics

RL algorithms have the ability to find optimal policies given raw input data. It turns out to be especially interesting for Robotics because it allows learning control policies using a wide range of sensors available. Already in 2006, a successful RL implementation for autonomous vehicles was published [18], which learns how to control a helicopter using Dynamic Programming (DP). Many researchers have tried to use RL for robotics, using a wide range of autonomous robots, ranging from humanoid robots [19] [20], robotic arms [21] [22], navigation of wheeled robots [23] [24]. Nowadays, RL is used in a wide range of robotics topics, such as robotic manipulation [25] [26], self-driving cars [27]–[29]. Navigation of autonomous robots using global planners works well in static environments with globally known obstacles; new obstacles are handled by a local planner, but it fails frequently, especially with moving objects. Applying RL seems promising: ideally, robots should learn and adapt to the environment's changes. Anyway, RL training is a resource and time-consuming trial-and-error process. As a consequence, RL agents are trained in a simulation environment which gives a good representation of the world: the better the environment, the better the real-world experiment.

In opposition to the single-agent implementations, learning how to control a group or swarm of robots in a *centralized learning, decentralized execution* fashion is a very challenging task that has still to be exhaustively explored. Many papers regarding navigation and obstacle avoidance are available [30]–[32], but all of them consider only 2D simulations of wheeled robots. Navigation of

multiple flying drones, in particular quad-rotors, has not been very thorough yet, possibly because of the computational complexity of the algorithms with the huge size of the search space.
Nonetheless, single drones paired with RL have been mostly investigated in different situations, such as hovering control [33] [34] and autonomous landing [35] [36]. A recent publication shows that RL can be successfully applied for drone navigation [37], in particular considering the travel through a set of waypoints, that are included in the drone state and so fully observable.
Another approach for navigation is simply using Supervised Learning for the so-called Visual Odometry (VO) or Visual SLAM: in VO, a local trajectory is the main concern and a local map is used to obtain a more accurate estimate of the agent trajectory; SLAM is more concerned in constructing a global map [38]. As shown by [39], a supervised data set is used to train a network including Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) to estimate the pose of the drone; another paper published this year, effectively uses Transformers for VO and navigation using different sensors such as RGB and depth cameras [40].

The SOTA for the topic I will be investigating comes directly from the classical control theory: the most important publications regarding navigation and obstacle avoidance are [41]–[43] and consist of combining optimization-based trajectory planning approaches and local planners. I will show that RL and, in particular, PPO, can be successfully applied to learn a control policy for swarm behaviors for obstacle avoidance and navigation, achieving very good results in a collaborative scenario where each agent receives a different reward.

1.3 Neuroscience context

An important motivation for the ongoing research regarding Reinforcement Learning is that it is strictly correlated to how human beings learn. Neuroscientists have identified brain regions and neural circuits that are involved in various aspects of RL. Key brain regions include the *Striatum*, *Prefrontal Cortex*, *Hippocampus*, and other dopamine-producing areas. These regions are thought to play roles in representing reward prediction errors (the difference between expected and actual rewards), learning action-outcome associations, and updating decision-making policies [44].
Dopamine neurons in the midbrain are crucial in signaling rewards. These neurons fire with different frequencies when rewards are higher or lower than expected, contributing to the learning of associations between actions and their outcomes. The model known as the *dopamine prediction error hypothesis* suggests that these neurons encode a teaching signal that guides learning by

updating the value associated with different actions [45].
In particular, the neural activity observed in the *Striatum* and *Prefrontal Cortex* is consistent with TD learning processes (referred to as *Rescorla–Wagner model*), an RL method that we will see in Section 2.1.4. Moreover, the important concept of the policy (where actions are organized into sequences and subgoals) is also present in the human brain: *Prefrontal Cortex* and *Basal Ganglia* are believed to play roles in learning and executing these policies. On the other hand, dysregulation of reward processing and decision-making mechanisms can substantially contribute to human neuropsychiatric disorders, such as addiction, depression [46].

1.4 Quadcopter drone

For a better understanding of the research, a definition of the quadcopter drone and the dynamics involved in its movement are required: a quadcopter is a type of Unmanned Aerial Vehicle (UAV) that uses four rotors for flying. Each rotor generates some thrust, to achieve controlled flight, stability, and maneuverability of the drone. Lately, quadcopters have become popular platforms for various applications, such as photography, surveillance, search and rescue, and research. Their advantages with respect to fixed wings drones are that they can hover in place, fly in confined spaces, and perform agile maneuvers. Nowadays, their importance is increasing steadily thanks to their effective usage during important historic events, such as the war in Ukraine: given their cheap manufacturing cost, they have been successfully adopted with a military purpose, comprehending aid support.

1.4.1 Dynamics

The dynamics of a quadcopter involve the physical principles that govern its motion and control. Understanding quadcopter dynamics is crucial for designing effective control algorithms, including those based on reinforcement learning. Main components are described in a simplified fashion, taken by [47]:

- **Thrust and Torque**: each of the four rotors generates thrust by spinning, thus creating an upward force that counters the gravitational pull on the quadcopter. The difference in thrust between rotors creates torque, causing the rotation around its center of mass.
- **Degrees of Freedom**: there are six degrees of freedom: three translational (X, Y, and Z

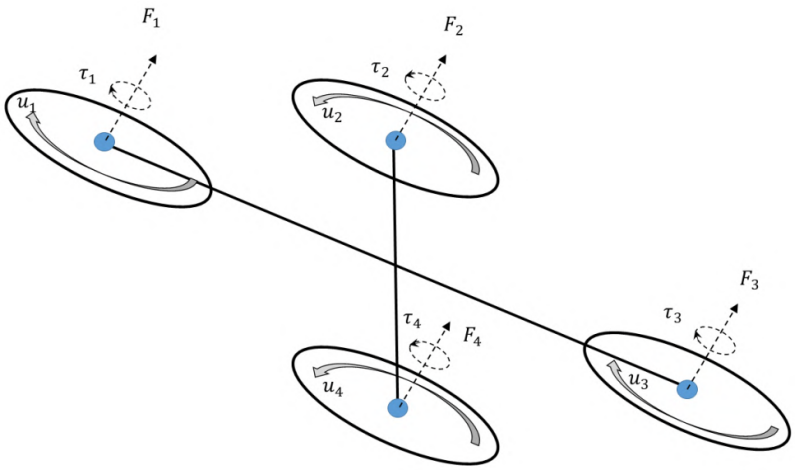


Figure 1.1: Vehicle model for quadrotor: the four-rotor vertices experience the spinning $\mu_1, ..., \mu_4$, which result in the forces $F_1, ..., F_4$ and the torques $\tau_1, ..., \tau_4$. This figure is taken from [48].

axes) and three rotational (*roll*, *pitch*, and *yaw*). Control inputs, which are usually velocity, acceleration, or force, adjust the thrust of each rotor to control these degrees of freedom.

- **Flight Dynamics**: the drones can perform various flight maneuvers by adjusting the speeds of the rotors, i.e., to achieve forward flight, the rear rotors spin faster than the front rotors. Tilting the quadcopter forward or backward controls pitch, while tilting left or right controls roll. Yaw control involves adjusting the speeds of the rotors on opposing sides.

- **Stability and Control**: there is a combination of sensor feedback (e.g., accelerometers, gyroscopes) and control algorithms to maintain stability and achieve desired flight trajectories. Proportional-Integral-Derivative (PID) controllers or more advanced control techniques are often used for stabilization.

- **Collision Avoidance**: to avoid obstacles, additional sensors could be added to the quadcopter, such as cameras or laser images. In swarming scenarios, quadcopters may also need to avoid collisions with each other to maintain a safe and coordinated flight, requiring coordination and communication.

Defining the dynamics is the starting point for the thesis. RL algorithms require an accurate understanding of the system's dynamics to effectively learn policies. Knowing how the agent's actions affect the quadcopter's state (i.e., position, velocity, thrust) is crucial for designing appropriate reward functions and training procedures, but also for generating training data efficiently from the simulation environment. Moreover, quadcopters will have to operate in physical environments, and understanding their dynamics contributes to safe and optimal operation, preventing crashes and ensuring efficient swarm behavior.

2

Background

Contents

This chapter summarizes the theoretical foundations for comprehending more in-depth the topic of the thesis. In section 2.1, the basics of Reinforcement Learning (RL) are presented, including algorithms like Monte Carlo (MC) and Temporal-Difference (TD) Methods. Section 2.2 explains the need for function approximation to give birth to Deep Reinforcement Learning, together with historical algorithms such as *Deep Q-Network* (DQN), *REINFORCE*, *Proximal Policy Optimization* (PPO). Section 2.3 extends the theoretical concept to the multi-agent scenario, describing the most common mechanisms and techniques. Most of the concepts are taken from [13], the most important book for Reinforcement Learning. Section 2.3 explores the theoretical concept of RL algorithms to a multi-agent scenario. Finally, section 2.4 introduces Curriculum Learning, a technique used to improve the learning process in Machine Learning and, eventually, the generalization of the agents in Reinforcement Learning.

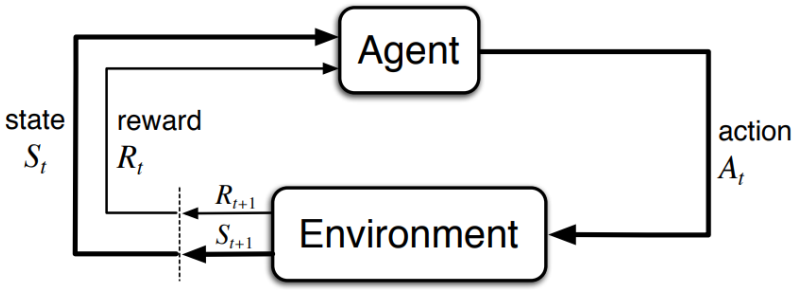


Figure 2.1: The agent–environment interaction in Reinforcement Learning, or, more precisely, in a Markov decision process. This figure is taken from [13].

2.1 Reinforcement Learning (RL)

Reinforcement learning is learning what to do (how to map states to actions) to maximize a numerical reward signal. The agent is not told which actions to choose but instead must discover which actions yield the biggest reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and all subsequent rewards. Trial-and-error search and delayed reward are the two most important distinguishing features of reinforcement learning [13].

Using these characteristics, it is possible to build a closed-loop scheme like the one in Figure 2.1: the agent and environment interact at each discrete time step $t = 0, 1, 2, ...$; at each t, the agent is in a certain state $S_t \in \mathcal{S}$, with $\mathcal{S}$ set of all the possible states, and select one of the possible available actions $A_t \in \mathcal{A}(s)$. As the action's consequence, the environment changes to a new state S_{t+1} and the agent receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ as a feedback about how good it was to take that action; and so on. This gives rise to a trajectory.

2.1.1 Multi-Armed Bandits

The multi-armed bandit problem serves as a foundational example of RL and the exploration-exploitation trade-off, a distinctive challenge that arises while learning: imagine a gambler facing a row of slot machines (here, referred to as bandits), each with an unknown probability distribution of paying out rewards. The gambler's goal is to maximize the cumulative reward obtained over a series of trials while deciding which machines to pull. The only difference with the full Reinforcement Learning problem is the *non-associativity* setting: it does not involve learning to act in more than one situation, since each state is not dependent on the action taken.

In a k-armed bandit problem (with k bandits), each of the k actions, if selected at time t, has

an expected reward, called the *value* of that action A_t. A reward R_t corresponds to the action. Therefore, the *value* of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a] \tag{2.1}$$

Assuming that you do not know the action values but just the estimates, we denote as $Q_t(a)$ (Q-value) the estimated value of action a at time step t: we would like it to be close to the real value. If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest, called *greedy* actions. Selecting one of these actions means you are **exploiting** the values of the actions; selecting one of the *non-greedy* actions, you are **exploring**, enabling the improvement of the estimate of the non-greedy action's value: exploitation is the right thing to maximize the reward on a single step, but exploration may increase the total reward in the long run.

This problem gives rise to the first important notion of the field, the *action-value* methods, which estimate the values of actions and use them to make action selection decisions: there are plenty of possible ways to do so; for now we consider averaging the rewards received during the trial-and-error process.

Another point that arises is the balancing problem between exploration and exploitation. The simplest action selection rule for exploitation is to select one greedy action and, if there is more than one, then select one randomly, to maximize the immediate reward:

$$A_t \doteq \text{argmax}_a Q_t(a) \tag{2.2}$$

This method never selects actions with lower estimates to check if they might behave better; a simple alternative is to behave greedily but every once in a while, with small probability ε, instead select randomly from the actions with lover estimates. These methods are called ε-*greedy* methods. In the limit as the number of steps increases, every action will be selected an infinite number of times, ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$.

The value estimates of actions can be troublesome: maintaining a record of all the rewards and performing the averages whenever the estimated value is needed is computationally prohibitive. Luckily, we can use incremental formulas for updating averages with only a small, constant computation. Given Q_n and the reward R_n, the new average of all n rewards can be computed by:

$$Q_{n+1} = Q_n + \frac{1}{n}\big[R_n - Q_n\big] \tag{2.3}$$

This equation is only suitable in the case of stationary problems, so when reward probabilities do not change over time. If the problem is non-stationary, it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter $\alpha \in (0, 1]$. The equation (2.3) then becomes:

$$Q_{n+1} \doteq Q_n + \alpha\big[R_n - Q_n\big] = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i \tag{2.4}$$

Resulting in Q_{n+1} being a weighted average of past rewards and the initial estimate Q_1; sometimes it is convenient to vary the step-size parameter α from step to step.

Other important considerations can be underlined (see Figure 2.2):

- **Optimistic Initial Values**: given the dependency on the initial action-value estimate Q_1, we can encourage exploration by giving a greater initial estimate to all the actions.
- **Upper-Confidence-Bound (UCB) Action Selection**: opposed to the ε-greedy action selection, it selects actions that have high estimated Q-values and high uncertainty.

$$A_t \doteq \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2.5}$$

  Where $N_t(a)$ denotes the number of times that action a has been selected before t ($N_t(a) = 0$ denotes a as a maximizing action) and $c \leq 0$ controls the degree of exploration.

2.1.2 Finite Markov Decision Processes

When the problem becomes *associative*, so when different actions affect and generate different situations, we can talk about Markov Decision Process (MDP). In MDPs, we try to estimate the value $q_*(s, a)$ of each action a in each state s, or the value $v_*(s)$ of each state given optimal action selections. These quantities are important to assigning value to the long-term consequences of actions.

In a *finite* MDP, the states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$ are finite sets. Therefore, R_t and S_t have discrete probability distributions dependent only on the preceding state and action:

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \qquad \forall s', s \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}(s) \tag{2.6}$$

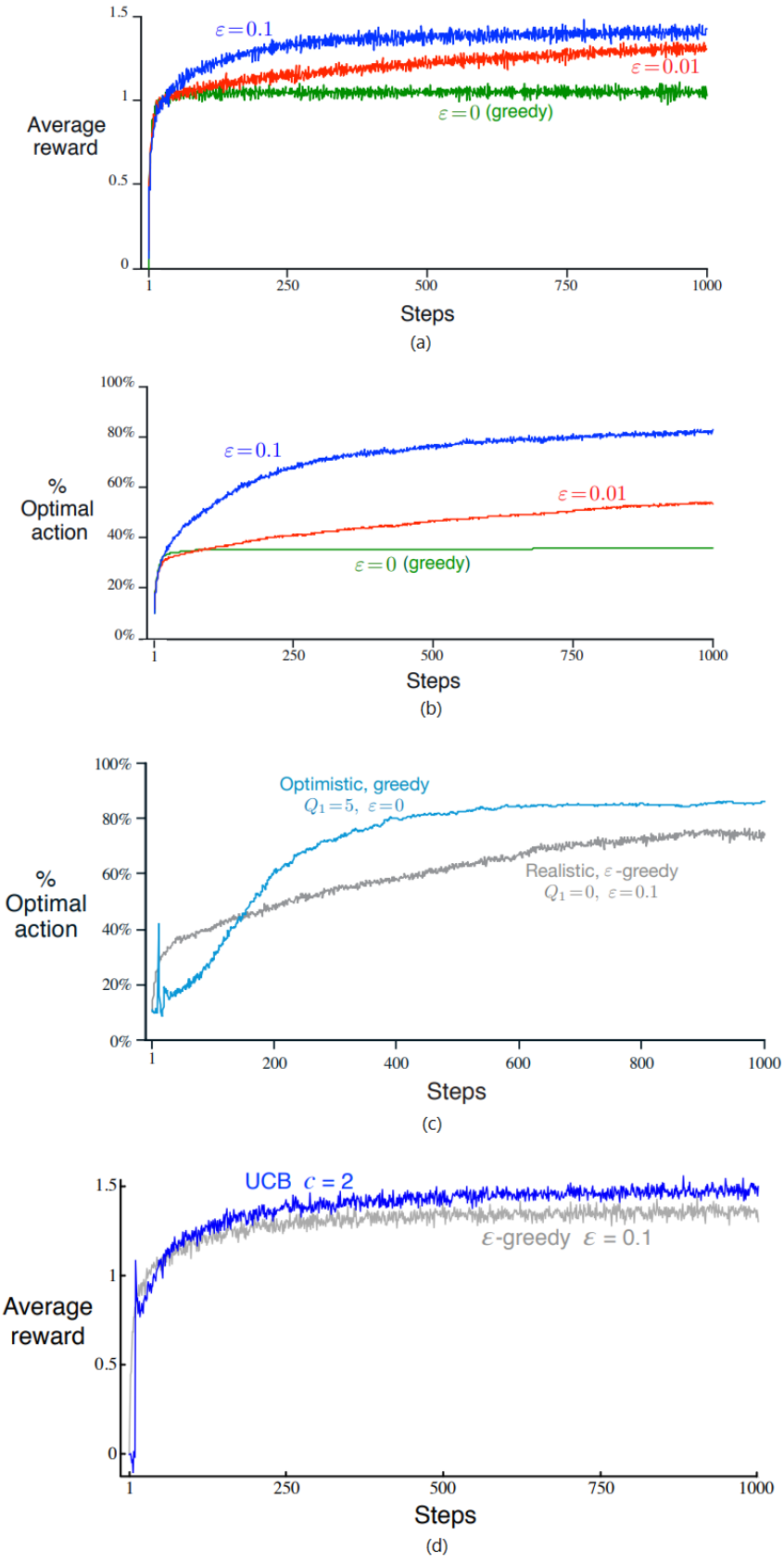


Figure 2.2: Comparison of the different methods using an exemplar 10-armed bandit problem. Results are run on 2000 randomly generated instances. **(a)** and **(b)** compare the performances at different ε values. **(c)** shows the improvement with higher initial estimates. **(d)** compares UCB with ε-greedy action selection. Figures are taken from [13].

The function p defines the *dynamics* of the MDP. By definition, the probability of each possible value for S_t and R_t depends on the preceding state and action, S_{t-1} and A_{t-1}, and not on earlier ones. The state must include information about all the past agent–environment interactions and, if so, the state is said to respect the *Markov property*. Much information is obtainable by Equation (2.6), such as the *state-transition probabilities*:

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{2.7}$$

In a MDP, the agent's goal is to maximize the reward it receives: not the immediate reward, but the cumulative reward in the long run. This cumulative reward is called *expected return* G_t and it is defined as a function of the reward sequence. In the simplest case, with T final time step:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.8}$$

This creates a first subdivision of RL problems: when it is possible to determine a terminal state, with a final time step, we have an *episodic task*; if the trial continues indefinitely, we call the problem *continuing task*.

Another concept that is important to define is the one of *discounting*: a parameter $0 \leq \gamma \leq 1$ called *discount rate* determines the present value of future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.9}$$

Moreover, rewards at successive time steps are related to each other:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.10}$$

Finally, the notion of MDP gives rise to the main points of RL:

- **Policy**: a mapping from states to probabilities of selecting each possible action. Following a policy π at time t, $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$.
- **Value functions**: states (or state–action pairs) functions that indicate how good it is for the agent to be in a given state (or to act in a given state).
    1. **State-value function for policy** π is the expected return when starting in s and

following π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \qquad \forall s \in \mathcal{S} \tag{2.11}$$

2. **Action-value function for policy** π is the expected return starting from s, selecting an action a, and following π:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.12}$$

Both value functions have a relationship similar to the one of Equation (2.10):

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\Big] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\big[r + \gamma v_\pi(s')\big] \qquad \forall s \in \mathcal{S} \end{aligned} \tag{2.13}$$

This equation takes the name of *Bellman equation* for v_π and shows the relationship between the value of a state and the values of the successors.

The main purpose of RL is about finding an *optimal* policy π_* (there may be more than one) from which originates an *optimal state-value function* v_* and an *optimal action-value function* q_*:

$$v_*(s) \doteq \max_\pi v_\pi(s) \qquad q_*(s,a) \doteq \max_\pi q_\pi(s,a) \qquad \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.14}$$

q_* can be written in terms of v_*:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{2.15}$$

Finally, both optimal functions can be rewritten in terms of the Bellman equation, namely *Bellman*

optimality equations:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, , A_t = a] \\ &= \max_a \mathbb{E}_{\pi}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.16}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\big] \\ &= \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma \max_{a'} q_*(s', a')\big] \end{aligned} \tag{2.17}$$

For finite MDPs, the Bellman optimality equations (a system of equations, one for each state) have unique solutions. Then, determining an optimal policy is straightforward: with v_*, any policy that assigns a nonzero probability to actions that get maximum with the Bellman optimality equation is an optimal policy; with q_*, for any state s, simply find any action that maximizes $q_*(s,a)$.

2.1.3 Monte Carlo Methods

Monte Carlo (MC) methods are the first learning methods for estimating value functions and discovering optimal policies. To do so, they use a trial-and-error process to retrieve sequences of states, actions, and rewards from actual or simulated interaction with an environment. We will consider MC only for episodic tasks: they will be able to converge to an optimal policy only in an episode-by-episode sense (there is no online learning such as step-by-step that will be considered in the next section): it means that value estimates and the policy are only changed at the end of each episode.

The idea of MC methods is simply to average the returns observed after visiting each sampled state, both for state-value functions and action-value functions. As such, the policy converges towards the optimal with the increasing number of episodes.

It is possible to recognize two different MC prediction (here, prediction referring to the fact that they compute the state-value function v_π for an arbitrary policy π) methods: *first-visit* MC, which estimates the value of a state by averaging the returns of all the first visits to that state across

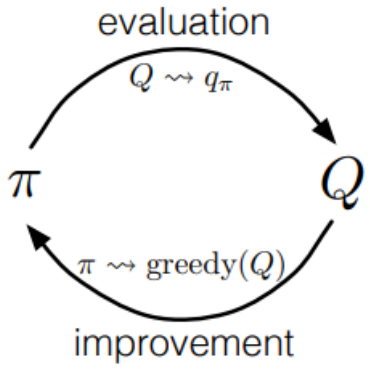


Figure 2.3: MC control diagram: during evaluation, the value function is modified to approximate the value function for the current policy; during the improvement, the policy is improved with respect to the current value function. Figure taken from [13].

multiple episodes; *every-visit* MC, which estimates the value of a state by averaging the returns of all visits to that state across episodes. Both adopt the same formula:

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)} \sum_{i=1}^{N(S_t)} G_i \tag{2.18}$$

Where $N(S_t)$ is the number of first visits (or every visit) to state s, and G_i is the return obtained in the i-th visit.

The same formula can be used for a control problem for policy improvement (see Figure 2.3). Given a policy π, we can estimate the action-value function $q_\pi(s, a)$ using MC methods and then improve the policy by choosing the action with the highest estimated Q-value for each state:

$$Q(S_t, A_t) = Q(S_t, A_t) + \frac{1}{N(S_t, A_t)} \sum_{i=1}^{N(S_t, A_t)} G_i \tag{2.19}$$

where $N(S_t, A_t)$ is the number of visits of the state-action pair (S_t, A_t).

Again, in practical implementations, incremental updates are used to efficiently update value function estimates, as already shown in Equation (2.3):

$$V(S_t) = V(S_t) + \alpha[G_t - V(S_t)] \tag{2.20}$$

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)] \tag{2.21}$$

Where G_t is the observed return at time t and α is a different weight than $1/N(S_t, A_t)$, so that, instead of taking the true average, recent returns can be weighted more or less strongly.

To ensure exploration, the exploring-starts method involves starting each episode with a random initial state-action pair. This guarantees that every state-action pair has a non-zero probability of being selected. Another approach would be using a ε-greedy policy: the policy selects the action with the highest estimated Q-value with probability $1 - \varepsilon$ and selects a random action with probability ε.

Given the MC control problem, a further subdivision of RL problems can be introduced: **on-policy** algorithms use the same policy to evaluate and improve; in **off-policy** algorithms the policy used to generate behavior, called the *behavior policy*, may be unrelated to the policy that is evaluated and improved, called the *estimation policy*. Until now, we have always considered the first kind of algorithms; even if an off-policy Monte Carlo method exists, the notion will be more important in the next Section, when talking about *Q-learning*.

2.1.4 Temporal-Difference Learning

TD methods are crucial for Reinforcement Learning theory: they are a combination of MC and Dynamic Programming (DP) ideas. Unlike MC methods, they update estimates based on incomplete sequences of experience, without waiting for the episode to finish (*online* learning). It can be used both for a prediction (estimating the value function for a given policy) and a control problem (finding an optimal policy). The key idea is to update the value estimate of a state using the current estimate of its successor state's value, making these methods less reliant on complete episodes and more efficient than MC methods.
The most simple TD method is TD(0), which updates the state-value function in this way:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.22}$$

MC must wait until the end of the episode to determine the increment to $V(S_t)$ (when G_t is known); TD needs to wait only until the next time step: at time $t + 1$, TD updates using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. It can also be called *one-step* TD since it updates the value using the estimates of only one step ahead. Furthermore, the quantity in brackets can be considered as an error, called *TD error*:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.23}$$

Other important prediction TD methods include *n-step* TD, where the error considers more than one step, and TD(λ). The latter introduces *eligibility traces* which serve as a connection between TD and MC methods: if $\lambda = 1$, we have Monte Carlo; if $\lambda = 0$, we have TD(0). TD(λ) algorithm is a particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^{n-1} ($\lambda \in [0, 1]$) normalized by a factor of $1 - \lambda$:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{2.24}$$

Regarding control algorithms, TD offers both on-policy and off-policy methods. They are the most important RL constituents, from which the SOTA algorithms we have today originated. The pseudo-code is also easily implementable.

- **Sarsa**: is the on-policy TD control algorithm that learns action values. To balance exploration and exploitation, it uses an ε-greedy policy. Its update rule is defined as:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.25}$$

  This rule uses a quintuple of elements $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ which give rise to the algorithm name.

Algorithm 1: Sarsa (on-policy TD control)

Data: $\alpha \in (0, 1]$, $\varepsilon > 0$
Initialize $Q(s, a)$ $\forall s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$
$Q(terminal, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (ε-greedy)
  foreach *step in episode* **do**
    Take action A
    Observe R, S'
    Choose A' from S' using policy derived from Q (ε-greedy)
    $Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S = S'$
    $A = A'$
  end
  Until S is terminal
end

- **Q-learning**: is the off-policy TD control algorithm that learns action values without following the behavior policy. It updates action values using the maximum estimated Q-value of

the next state. Its update rule is defined as:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.26}$$

Algorithm 2: Q-learning (off-policy TD control)

Data: $\alpha \in (0, 1]$, $\varepsilon > 0$
Initialize $Q(s, a)\ \forall s \in \mathcal{S}^+,\ a \in \mathcal{A}(s)$
$Q(terminal, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  foreach *step in episode* **do**
    Choose A from S using policy derived from Q (ε-greedy)
    Take action A
    Observe R, S'
    Choose A' from S' using policy derived from Q (ε-greedy)
    $Q(S, A) = Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S = S'$
  end
  Until S is terminal
end

2.2 Deep Reinforcement Learning (DRL)

The Reinforcement Learning methods discussed in the previous section considered problems in which state and action spaces are small enough for the value functions to be represented as arrays, or tables. They are usually referred to as *Tabular Methods*: these methods can often find exact solutions, including the optimal value function and the optimal policy. Anyway, in real-world problems, the state space can easily become large, i.e. the possible number of camera images is larger than the number of atoms in the universe. Therefore, it is not feasible to visit all possible states and save the value for all action-state pairs in a table and it is almost impossible to find an exact optimal policy or optimal value function: an approximate solution should be found instead. An idea is about substituting the table with a function that can estimate all the values inside it and that can also generalize and recognize the similarity between states or values, so to decrease the complexity of the problem: a *function approximation* is needed. Examples of function approximators include Machine Learning, Artificial Neural Networks, but also statistical curve fitting; the correct approach depends on the kind of problem.

Deep Learning is a well-known function approximator that can approximate non-linear functions

and extract relevant features from raw inputs. When it is used together with Reinforcement Learning, it produces Deep Reinforcement Learning (DRL) together with its family of methods, the *Approximate Solution Methods*.

2.2.1 Value Based Methods

Value-based methods are the natural extension of the Temporal Difference (TD) methods from classical Reinforcement Learning discussed in the previous chapter. The idea is, again, to replace the value table with an approximation obtained using a Deep Neural Network that is trained using Stochastic Gradient Descent (SGD). The network's output provides probabilities for each possible action of the case in the study. Then, a traditional policy chooses the action among the probabilities (e.g., ε-greedy policy).
The only problem that arises while considering function approximations is the convergence assumption. Proving the convergence of the algorithms is much more difficult, especially considering off-policy algorithms: famous is the so-called Baird's counterexample [49], which shows that off-policy approximate methods can diverge if the initialization is not correctly handled.
Anyway, convergence proofs are outside the scope of this thesis; also, basic methods such as linear approximation methods and gradient Monte Carlo or Sarsa will not be discussed here.

Deep Q-Network (DQN)

The first outstanding results of combining Deep Neural Networks and Reinforcement Learning come from a paper from 2013 [1]. These results were further investigated and gave birth to the first famous DRL algorithm, the Deep Q-Network (DQN) [50]. Google's DeepMind group presented an approach, which combined off-policy Q-Learning with DNN, that could learn successful policies from high-dimensional inputs using RL. Deep Q-network agent, receiving pixels and game scores as inputs, was able to achieve a level comparable to that of a professional human gamer across a set of 49 classic Atari 2600 games, using the same algorithm, network architecture, and hyper-parameters. The output of the network provides a probability distribution over all possible discrete actions; the selection of the best one is straightforward.

Q-Learning with function approximation has shown a problem of unstable learning [51]; the reasons are many, i.e. small updates to Q may significantly change the policy and change the data distribution. Two additional mechanisms were added to address it: *experienced replay* and *frozen*

target network. The first one is about storing the agent's experiences $S_t, A_t, R_{t+1}, S_{t+1}$ in a buffer. In each training step, a batch of experiences is sampled from the buffer and fed to the network, ensuring that old experiences are repeated from time to time, causing a smoothing effect over changes in the data distribution. The latter mechanism involves a second network to be trained together with the first one, with the same structure but different weights: θ for the Q-network and θ^- for the target network. The Q-network is regularly updated according to the standard loss function; the target network copies the parameters of the Q-network every C time step (thus, it is *frozen*). It smooths oscillating policies and leads to more stabilized learning.
As summary, a Deep CNN approximates the optimal action-value function:

$$Q^*(S, A) = \max_a \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = S, A_t = a, \pi] \tag{2.27}$$

The loss function for the Q-learning network update at iteration i:

$$L_i(\theta_i) = \mathbb{E}_t\Big[\Big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \theta_i^-) - Q(S_t, A_t, \theta_i)\Big)^2\Big] \tag{2.28}$$

DQN has been modified during the years and improved drastically. Most of the improvements are combined in the *Rainbow DQN* which achieves SOTA for value-based methods [5]:

- **Prioritized Experienced Replay** [52]: is about prioritizing experiences, to replay important transitions more frequently, so to learn more efficiently. Each experience has an additional priority value that gives a different sampling probability and lets some of them remain longer in the buffer. As a measure, the TD error is used: if it is high, the agent can learn more from the corresponding experience. DQN with this technique outperforms 41 out of 49 Atari games.

- **Double DQN** [53]: is an idea firstly introduced for the tabular counterpart. In practice, the Q-learning algorithm is known to overestimate action values under certain conditions; double DQN takes advantage of the two networks to modify the loss function and to reduce the observed over estimations:

$$L_i(\theta_i) = \mathbb{E}_t\Big[\Big(R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a, \theta_i), \theta_i^-) - Q(S_t, A_t, \theta_i)\Big)^2\Big] \tag{2.29}$$

- **Dueling DQN** [54]: creates two separate estimators: the typical one which estimates the state value function and another one that estimates the advantage of taking action in that

state, the *state-dependent action advantage function*. The main benefit is to generalize learning across actions: in fact, it leads to better a policy in the presence of many similar-valued actions and, surprisingly, identifies state information where actions have no effect, avoiding taking them into account.

2.2.2 Policy Gradient Methods

Value-based methods are important to deeply understand RL and DRL backgrounds. On the other hand, they have many limitations and most of today's research is shifted towards a different approach: *Policy Gradients Methods*. These methods directly learn a parameterized policy $\pi(a \mid s, \theta)$ that can select actions without consulting a value function, which may still be used to learn the policy parameter but is not required for action selection:

$$\pi(a \mid s, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\} \tag{2.30}$$

$\theta \in \mathbb{R}^d$ is the policy's parameter vector and is learned based on the gradient of some scalar performance measure $J(\theta)$ concerning the policy parameter. Since it is a *maximization* problem, the gradient is *ascent* (which can easily be converted to a descent, minimizing its negative value):

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.31}$$

The policy can be parameterized in any way, as long as $\pi(a \mid s, \theta)$ is differentiable concerning its parameters. Different parameterizations exist, both for discrete action spaces, where the most famous is the *soft-max in action preferences*, and continuous spaces, such as a *normal probability density over a real-valued scalar action*.

An *actor-critic* architecture is a Policy Gradient method that uses two parametrized functions for building the policy. The *critic* function estimates the value function, either the action-value $Q(s)$ or the state-value $V(s)$; the *actor* updates the policy distribution in the direction suggested by the critic. In other words, the learning of the actor is based on a gradient approach and the critics evaluate the action produced by the actor by computing its value (Figure 2.4). Actor-critic architectures are also present in the human brain and studies show their participation in the process of human decision-making [55].

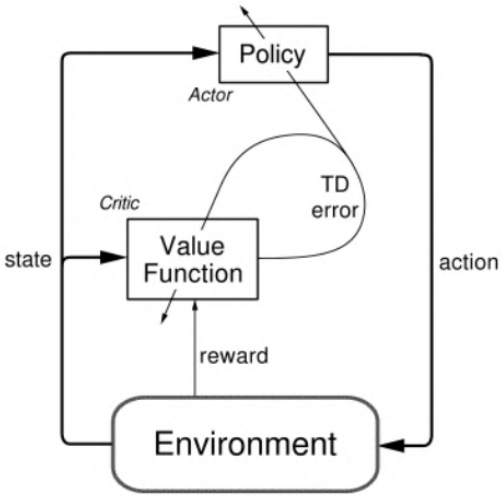


Figure 2.4: Typical structure of an Actor-Critic network; TD-error can be substituted with different errors depending on the algorithm which uses this structure. This figure is taken from [13].

One advantage of parameterized policies is that the approximate policy can get similar to a deterministic policy; another advantage is that it can select actions with arbitrary probabilities (creating stochastic policies), while action-value methods can not; policy-based methods learn faster and yield better policies compared to value-based methods [56]; finally, the choice of policy parameterization could be a good way of injecting prior knowledge to the problem. The most important advantage is strictly theoretical: stronger convergence guarantees are available for policy-gradient methods than for action-value methods, thanks to the *Policy Gradient Theorem* [57]. It establishes that:

$$\nabla J(\theta) \propto \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a) \nabla \pi(a \mid s, \theta) \tag{2.32}$$

Where μ is the on-policy distribution under μ and the constant of proportionality is the average length of an episode for episodic tasks, 1 for continuing tasks (so it becomes equality). Thanks to this equation, it is possible to estimate the gradient concerning the policy when the gradient depends on the unknown effect of policy changes on the state distribution.

REINFORCE

The REINFORCE algorithm was introduced in 1992 and is at the foundation of the Policy Gradient Methods [6]. REINFORCE uses the complete return from time t, which includes all future rewards

up until the end of the episode, in case of an episodic task. Because of this, REINFORCE is considered a Monte Carlo Policy-Gradient Control algorithm [13].
To derive the update rule for the algorithm, we must recall the *Policy Gradient Theorem* (2.32):

$$\begin{aligned}
\nabla J(\theta) &\propto \mathbb{E}_\pi\Big[\sum_a q_\pi(S_t,a)\nabla\pi(a \mid S_t,\theta)\Big] && \text{(called } \textit{all-actions} \text{ method)} \\
&= \mathbb{E}_\pi\left[\sum_a \pi(a \mid S_t,\theta) q_\pi(S_t,a)\frac{\nabla\pi(a \mid S_t,\theta)}{\pi(a \mid S_t,\theta)}\right] && \text{(divide and multiply)} \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t \mid S_t,\theta)}{\pi(A_t \mid S_t,\theta)}\right] && \text{(replace } a \text{ by sample } A_t \sim \pi) \\
&= \mathbb{E}_\pi\left[G_t\frac{\nabla\pi(A_t \mid S_t,\theta)}{\pi(A_t \mid S_t,\theta)}\right] && (\mathbb{E}_\pi[G_t \mid S_t,A_t] = q_\pi(S_t,A_t))
\end{aligned} \tag{2.33}$$

With G_t return; the fractional vector can also be written as $\nabla \ln x$ and is called *eligibility vector*. Given this expectation, we can derive the REINFORCE update:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla\pi(A_t \mid S_t,\theta_t)}{\pi(A_t \mid S_t,\theta_t)} \tag{2.34}$$

Each increment is proportional to the product of G_t and the gradient vector of the probability of taking the action actually taken divided by the probability of taking that action, which has direction that most increases the probability of repeating A_t on future visits to S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. In the discounted case, the factor γ^t is included.

Algorithm 3: REINFORCE (Monte Carlo Episodic Policy-Gradient Control)

Data: a differentiable policy parameterization $\pi(a \mid s,\theta)$, $\alpha > 0$
Initialize policy parameter $\theta \in \mathbb{R}^d$
foreach *episode* **do**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot \mid \cdot, \theta)$
  foreach *step in episode* $t = 0, 1, ..., T-1$ **do**
    $G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta = \theta + \alpha\gamma^t G \nabla \ln \pi(A_t \mid S_t,\theta)$
  end
end

The policy gradient theorem can be generalized to compare the action value q_π to a *baseline* $b(s)$ that can have any value. The only change in the update rule (2.34) is $(G_t - b(S_t))$. This baseline, which could be used also for easy problems such as the bandit described in Section 2.1.1, can be helpful to reduce the REINFORCE variance, inherited from MC methods, and thus speed up learning. As an initial value, it could be set up as an estimate of the state value $V(S_t)$.

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is the most popular SOTA Policy Gradient Method and was developed by OpenAI in 2017 [4]. Compared to other approaches such as TRPO [8], DDPG [7], or SAC [11], it is supposed to learn a more stable policy while being much simpler to tune, even though its hyperparameters are larger. This makes PPO often the first choice when it comes to solving RL problems: OpenAI still uses it as the baseline for their research.
PPO has some benefits of the TRPO algorithm and extends them with simpler implementation and lower general complexity. Starting from the REINFORCE with baseline algorithm:

$$\nabla J(\theta) \propto \hat{g} = \mathbb{E}_\pi[\hat{A}_t \nabla \ln \pi(A_t \mid S_t, \theta)] \qquad \text{where} \qquad \hat{A}_t = \sum_{k=0}^{T-t-1} R_{t+k+1} - V_\pi(S_t) \tag{2.35}$$

Differentiation software constructs an objective function (or loss) whose gradient is the policy gradient estimator; the estimator $\hat{g}$ is obtained by differentiating the objective $L^{PG}(\theta)$ [4]:

$$L^{PG}(\theta) = \mathbb{E}_t[\hat{A}_t \ln \pi(A_t \mid S_t, \theta)] \tag{2.36}$$

- **Importance Sampling**: in REINFORCE, at each time step a new trajectory is generated, the policy learns from it and then it is thrown away. Importance sampling was introduced in TRPO [8] so that collected trajectories with older policies can be effectively reused, giving them importance in the policy update. The resulting objective function, called *surrogate* function:

$$L^{IS}(\theta) = \mathbb{E}_t\left[\hat{A}_t \frac{\pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta_{old})}\right] \tag{2.37}$$

- **Adaptive KL Penalty Coefficient**: is, again, inspired by TRPO algorithm. Since the variance between the older and the new policies can get quite large and can overshoot the maximum, the step size during the optimization process must be controlled. One way to do so is with the *Trust Region* optimization [8]: the difference between the policies can be measured using the *KL divergence*.

$$KL[\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s)] = \sum_{a \in A} \pi_{\theta_{old}}(a|s) \frac{\pi_{\theta_{old}}(a|s)}{\pi_\theta(a|s)} \tag{2.38}$$

  This KL divergence can be taken into account in $L^{IS}(\theta)$, weighted using a hyperparameter β. The more the difference between the old and new policy, the more the objective function

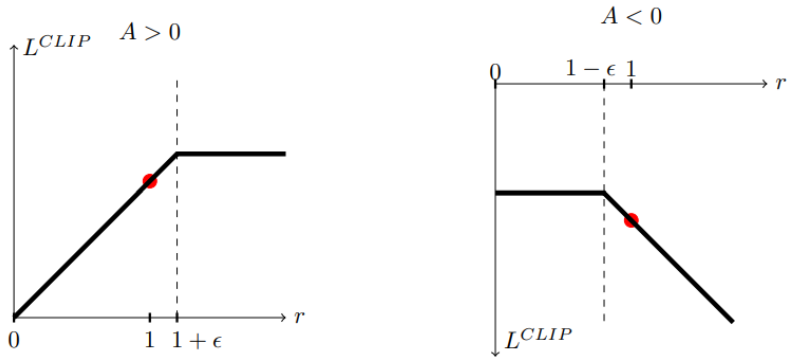


Figure 2.5: Single time-step of L^{CLIP}; $\hat{A}_t > 0$ on the left, $\hat{A}_t < 0$ on the right. L^{CLIP} sums many terms like these. This figure is taken from [4].

is punished:

$$L^{KL}(\theta) = \mathbb{E}_t\left[\hat{A}_t \frac{\pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta_{old})} - \beta KL[\pi_{\theta_{old}}(\cdot|S_t)||\pi_\theta(\cdot|S_t)]\right] \tag{2.39}$$

PPO further improves this technique by introducing an adaptive β, which is dependent on the KL divergence measure. Given $d = \mathbb{E}_t[KL[\pi_{\theta_{old}}(\cdot|S_t)||\pi_\theta(\cdot|S_t)]]$ and given a KL divergence target value d_{targ}:

$$\beta = \begin{cases} \beta/2 & \text{if } d < d_{targ}/1.5 \\ \beta \times 2 & \text{if } d > d_{targ} \times 1.5 \end{cases} \tag{2.40}$$

The initial value of β can be designed, but the algorithm quickly adjusts it [4].

- **Clipped Surrogate Objective**: is the proposed modification given by PPO [4] and, as the KL penalty, stabilizes learning by restricting the update size from one policy to another. Denoting $r_t(\theta) = \frac{\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta_{old})}$ as the *probability ratio*, PPO clips it between the range $[1-\epsilon, 1+\epsilon]$. The new objective function becomes:

$$L^{CLIP}(\theta) = \mathbb{E}_t\big[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big] \tag{2.41}$$

Notice that the first term of the minimum is the standard surrogate objective L^{IS}, while the second *clip* term modifies the first by clipping the probability ratio to avoid that $r_t(\theta)$ moves outside $[1-\epsilon, 1+\epsilon]$ (usually, $0.1 \leq \epsilon \leq 0.3$). The minimum is used to get a final objective which is a lower, pessimistic bound of the objective ignoring the change in probability ratio when it would make the objective improve but including it when it would make it worse (Figure 2.6).

- The last two approaches can be used alternatively or at the same time, with different weights.

Algorithm 4: PPO with Actor-Critic Style [4]

Initialize policy parameter $\theta_{old} \in \mathbb{R}^d$
foreach *episode* **do**
  foreach *actor* $\in 1, ..., N$ **do**
    Run policy $\pi_{\theta_{old}}$ in the environment for T time steps
    Compute estimates $\hat{A}_1, ..., \hat{A}_T$
  end
  Optimize chosen surrogate L wrt θ (using *SGD* or *Adam*, for K epochs, batch size $M \leq NT$)
  $\theta_{old} = \theta$
end

2.3 Multi-Agent Reinforcement Learning (MARL)

Reinforcement Learning can also be extended to a multi-agent setting with the name of Multi-Agent Reinforcement Learning (MARL). This special case of RL studies the interaction of multiple agents in an environment, where each agent aims to maximize its cumulative reward over time while agents' actions collectively influence the environment. A multiple-agent problem is inherently more complex since future rewards depend on others' actions, and, more importantly, because the computational complexity increases together with the search space.

2.3.1 Markov Games

First of all, the MDP definition described in Section 2.1.2 must be extended to comprehend multi-agent settings. It is known as **Markov Game**, or Stochastic Game [58]: it is defined by a set of states $\mathcal{S}$, and a collection of action sets, $\mathcal{A}_1, ..., \mathcal{A}_k$, one for each agent in the environment. State transitions are defined by the current state and one action from each agent; each agent also has an associated reward function, which can differ from the others, and seeks to maximize its expected total reward in the long term [59]. Moreover, MARL has a strong theoretical background in *game theory*, revolving around *social dilemmas*, such as *prisoner's dilemma* [60].

In Markov games, another complication that arises is the *non-stationarity*: in single-agent RL, the environment is often assumed to be stationary, meaning that the transition dynamics and rewards do not change, while in MARL, the environment can become non-stationary due to different agents affecting it, changing its dynamics, which adds complexity to learning. Anyway, it is proven that a Markov game has a non-empty set of optimal policies, at least one of which is stationary [59].

Markov games can include *partial observability*: it refers to a situation where the agents do not have complete information about the current state of the game. They have limited knowledge of the state, which impacts their decision-making and strategies. It can be caused by communication constraints, i.e. agents might not be able to communicate their observations or intentions to each other, or limited sensor information, e.g. in a board game, an agent might have visibility to only a portion of the board. This can be modeled by adopting an extension of the Markov game definition: Partially Observable Markov Game (POMG) or Decentralized Partially Observable Markov Decision Process (Dec-POMDP).

- **Dec-POMDP**: all agents attempt to maximize the joint reward function while having different individual objectives [61]. At every t, each agent takes an action and receives a local observation that is correlated with the state and an immediate joint reward. A local policy maps local histories of observations to actions, and a joint policy is a tuple of local policies. These problems are not solvable with polynomial-time algorithms and the best solutions are found using *planning* algorithms [62].

- **POMG**: instead of a joint reward function, agents optimize their individual reward functions in a partially observable environment. DP algorithms are suitable for POMG [63]. High dimensional space problems are intractable; autonomous driving is a POMG example [64].

MARL problems can be divided into two branches: *cooperative* and *competitive* games. In cooperative settings, agents work together to achieve a common goal, trying to maximize a common reward signal that they simultaneously receive, which evaluates their collective behavior. It gives rise to the *structural credit assignment* problem, that is the decision of which agent deserves credit for a higher reward, or must be blamed for an unfavorable one [13]. In competitive settings, agents' objectives conflict, and they aim to outperform each other. They receive different reward signals, and only focus on their one, without caring if other ones decrease. This creates a situation where each agent tries to challenge other agents' strategies in a loop, called *autocurriculum* [65]. The two branches can merge and result in a mixed-behavior problem. Algorithms like *Minimax-Q* can work well in competitive scenarios [59]; *Nash-Q* on cooperative ones [66].

Different approaches can be applied to solve a MARL problem. *Independent Learning* consists of treating other agents as part of the environment and assuming that their policies are fixed. Therefore, each agent learns its optimal policy as if it were the only learner in the environment. Standard algorithms like *Q-learning* and *SARSA* described in Section 2.1.4 can be extended to this

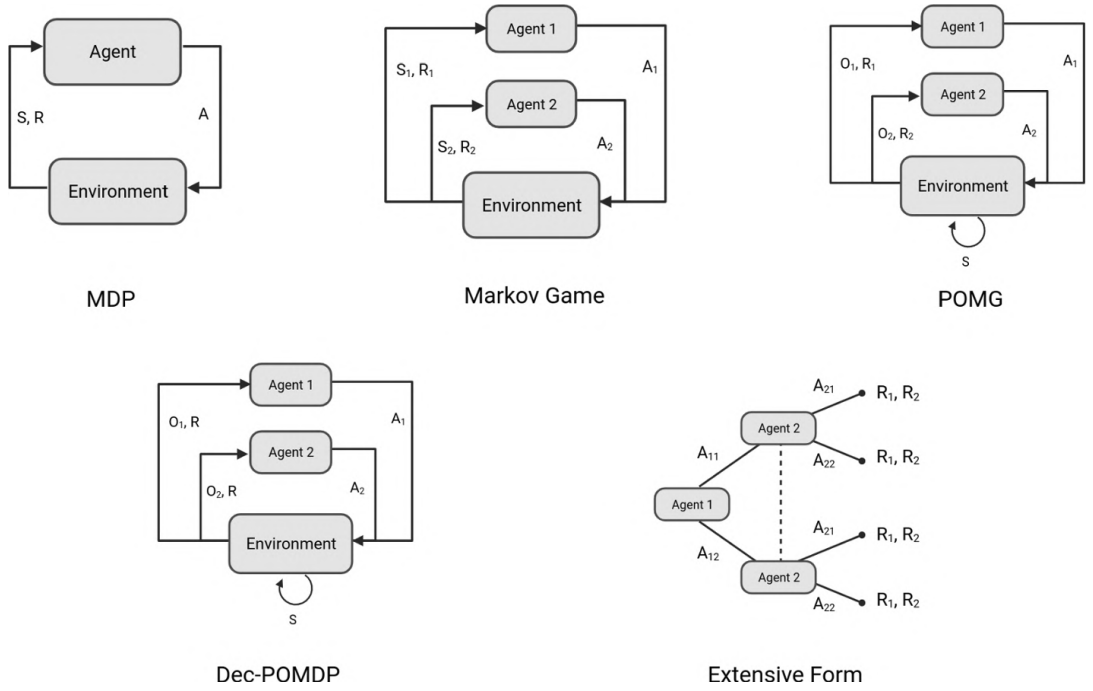


Figure 2.6: Taxonomy of MARL approaches. Extensive form refers to a problem formulation used when agents take turns sequentially. The O refers to observation, specifying that it does not comprehend all the information of the state S. This figure is taken from [62].

setting. However, convergence and stability can be problematic due to the changing environment and it is typically memory-unfriendly since n models have to learn in parallel. Another approach, *Joint Action Learning*, directly considers the interactions among agents and so a single joint action-value function (*Q-function*) is learned. In general, the second process has better performances, but can be computationally prohibitive [14].

An open research topic is the one of *centralized training and decentralized execution*: the main idea is that agents can access extra information during training, such as other agents' observations and rewards. Agents then execute their policy based on local observations, since they cannot access others' information. Their learning methods can be divided, as usual, into value-based and policy-gradient methods. Value-based methods focus on how to decouple centrally learned value functions and use them for decentralized execution; the most popular methods are *Value Decomposition Networks* [67] including QMIX [16].

2.3.2 Multi-Agent Actor-Critic

Policy-based actor-critic architectures use a centralized critic to train decentralized actors. A first example is given by the Counterfactual Multi-Agent (COMA) [68] which adopts a centralized critic, that has access to the stacked actions and observations for the Q-function approximation, but

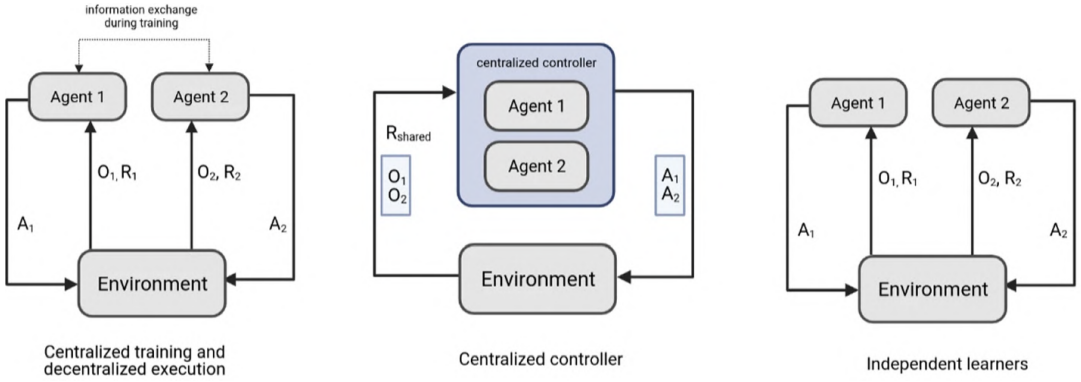


Figure 2.7: MARL training schemes. This figure is taken from [62].

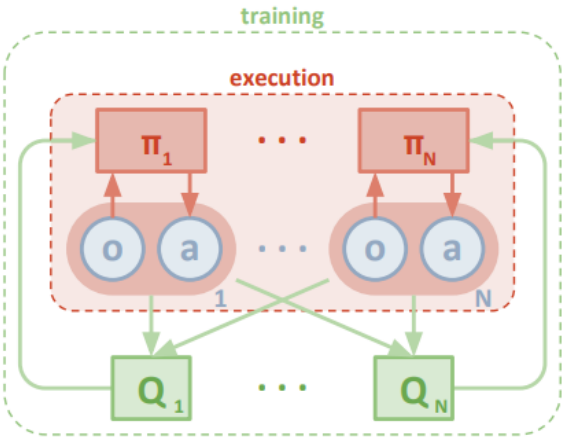


Figure 2.8: MARL Actor-Critic architecture structure. This figure is taken from [15].

decentralized actors for the policies, that depend on the single agent's action-observation sequence. Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [15] uses a centralized critic for each agent since they have different reward functions in mainly competitive environments. MADDPG can learn continuous policies, whereas COMA only discrete. Even if these methods help to reduce the overall complexity of the problem, obviously the input dimension increases exponentially with the number of agents. Hence, studies have tried to reduce this problem: Mean-Field Actor-Critic takes into account only the interaction with the neighboring agents using the mean-field theory [69].

Finally, a recent paper has shown the efficacy of applying single-agent actor-critic algorithms to multi-agent settings [17]. Considering a collaborative scenario with reward sharing, MAPPO can significantly outperform other MARL algorithms such as MADDPG, QMIX, and IPPO (which is the Independent Learning version of a multi-agent PPO). Given this result, I will show that this applies also in a different, collaborative, scenario where each agent has its own reward.

2.4 Curriculum Learning

Curriculum Learning is a ML technique introduced by Y. Bengio in the context of training artificial neural networks [70]. The idea is to improve and make easier the learning process by presenting the training data to a model in a progressive manner, moving from simple scenarios to more complex ones. This approach is inspired by the way humans tend to learn, starting with basic concepts before tackling more intricate topics.

In the general case, what changes over time is the difficulty of the data presented to the model for training. Initially, the model is exposed to easier samples that help it understand basic patterns. Then, the complexity of the examples is incrementally raised, allowing the model to learn progressively more intricate relationships and features. This gradual exposure to harder examples aims to prevent the model from getting stuck in suboptimal solutions and enables it to converge to a better overall solution while allowing the model to generalize better when facing different kinds of data [70].

Therefore, Curriculum Learning can lead to faster and better convergence and improved generalization. Furthermore, the organization of the curriculum can be guided by various factors, not only difficulty, but also diversity, or relevance to the task at hand. While facing successive tasks, the learning rate of the model should be lowered so as not to lose important patterns found in previous data.

It has been successfully applied in Deep Reinforcement Learning contexts, where a neural network is used to approximate the policy [71]. Thanks to curriculum learning, a wheeled multi-robot problem shows that the learned policy can be well generalized to new scenarios that do not appear in the entire training process [32].

3

Simulation environments

Contents

This chapter gives a brief description of the possible simulation environment to train quadrotors. The simulation environment is mandatory for this task to obtain training data from simulated sensors. It must support the training of multiple drones and be as similar as possible to reality. In section 1.1, the simulation environment provided by Robot Operating System (ROS) and Gazebo is presented. In section 1.2, an alternative given by Isaac Sim, a simulation environment by NVIDIA, is shown. Finally, section 1.3 defines the simulation environment that was employed for the training of drones, Microsoft's AirSim.

3.1 Robot Operating System (ROS) and Gazebo

Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications, while Gazebo is an open-source 3D robotics simulator. A set of packages, namely `gazebo_ros_pkgs`, provides the necessary interface to simulate a robot in Gazebo using ROS messages and services [72] [73].

In ROS, *publishers* and *subscribers* are the main building blocks to achieve communication. Publishers post messages to a channel, while subscribers listen to such channels to acquire messages.

These messages can have different kinds of data depending on their utility. Subscribers take these messages and execute a function to achieve something useful with that data. *Services* are used to achieve communication too; in this case, a client that uses a service communicates with servers: they publish a message and then wait for a response.
Using these simple starting blocks, ROS can be used to train RL agents. It also provides a package, `openai_ros`, that interfaces to OpenAI's *Gym*, an open-source Python library for developing and comparing RL algorithms by providing an API to communicate between learning algorithms and environments. However, the package is obsolete and did not follow the continuous evolution of the field given that it has not been updated since 2018. Moreover, it can not simulate multi-agent environments and a lot of work should be done to manually support the case.
On the other hand, ROS is the most efficient and effective way to create communication in robot implementations. *ArduPilot*[1], an open-source autopilot software capable of controlling almost any vehicle system that has been under development since 2010, can be easily implemented as a simulation-to-reality application using ROS and `MavRos`, a package that can convert ROS and MAVLink messages allowing ArduPilot vehicles to communicate with ROS.

3.2 NVIDIA Isaac Sim

NVIDIA Omniverse's *Isaac Sim*[2] is a robotics simulation toolkit that has essential features for building virtual robotic worlds and experiments. It supports navigation and manipulation applications through ROS and ROS2 and simulates sensor data from sensors such as RGB cameras and Lidar for various computer vision techniques such as segmentation, and object recognition.
As pros, it can interface to ROS and ArduPilot for robotic simulation, but also to the most famous RL framework, *Stable-baselines3*; it has included a variety of template robots for various applications, such as Jetbot, or Franka arm robot; furthermore, it is possible to import personalized robots with Unified Robot Description Format (URDF) files (an XML specification used in industry to model multibody systems) and to create new ones from scratch. It also supports multi-agent environments through the use of simulation extensions.
As cons, the coding documentation is not completely satisfactory; it does not natively provide a quadrotor interface but must be created from scratch; it is very time and computationally consuming, as it needs a high amount of memory together with high-range performing GPU. Another important problem is the impossibility of speeding up simulations: while training agents, this can

[1] ArduPilot Official Website
[2] Isaac Sim Official Website

be troublesome as a huge amount of episodes must be simulated.

3.3 Microsoft AirSim

Microsoft's *AirSim* is a simulator for drones, cars, and more vehicles, built on Unreal Engine. It also provides an experimental Unity release [48]. It is open-source, cross-platform software and supports hardware simulation with popular flight controllers such as PX4 autopilot [74] and ArduPilot (eventually interfacing ROS). It was specifically designed as a platform for AI research to experiment with deep learning, computer vision, and RL algorithms for autonomous vehicles.
It is easy to use and the simulation environment of Unreal Engine is helpful to create different maps in a simplified manner, providing elements more similar to the reality than what Gazebo and Isaac Sim can create. This last characteristic is fundamental to training the drones in case a real implementation is needed: the more the simulator can be compared to reality, the more accurate the real prediction will be.
AirSim can interface with most Machine Learning and Reinforcement Learning frameworks; it includes a ready-to-use API for quad-rotors and supports multi-agent systems; it allows to speed up the simulation to accelerate training. However, the project closed in 2022 since a new Microsoft flight simulator software will be published soon. Besides this, it has been chosen as the thesis simulator because of its complete documentation and its simplicity.

3.3.1 Unreal Engine

Unreal Engine[3] is a 3D computer graphics game engine developed by *Epic Games*. Initially developed for mostly video game development, it has been adopted by other industries, such as the film and television industry, and simulation platforms. The engine is written in *C++* and features a high degree of portability.
It provides a perfect platform to train RL agents: supports 2D, and 3D environments, is completely open-source, and is extremely customizable, i.e. it is possible to change the physics of the game, the collision presets, and the elements of an environment.
Creating different maps for RL-agents training is straightforward: it supports the creation of static and dynamic elements using a wide range of textures useful for computer vision purposes.

[3] https://www.unrealengine.com/

4

Methodology

Contents

This chapter introduces the project methodology and setup for training drones. Section 4.1 briefly describes the maps created with Unreal Engine for the training process, giving a general idea of the purpose of the thesis. In Section 4.2, the drone environment setup will be introduced, comprehending how drones are represented and how they move in the previously detailed maps. Section 4.3 defines the RL problem representations that include different observation spaces, action spaces, reward functions, and neural network function approximators; they depend on the approach used. Here, there will be a brief introduction to the algorithmic frameworks: *PettingZoo*, *Gymnasium*, *SuperSuit*, and *Stable-Baselines3*. Finally, Section 4.4 explains the Curriculum design choices that were produced to experiment with Reinforcement Learning training generalization.

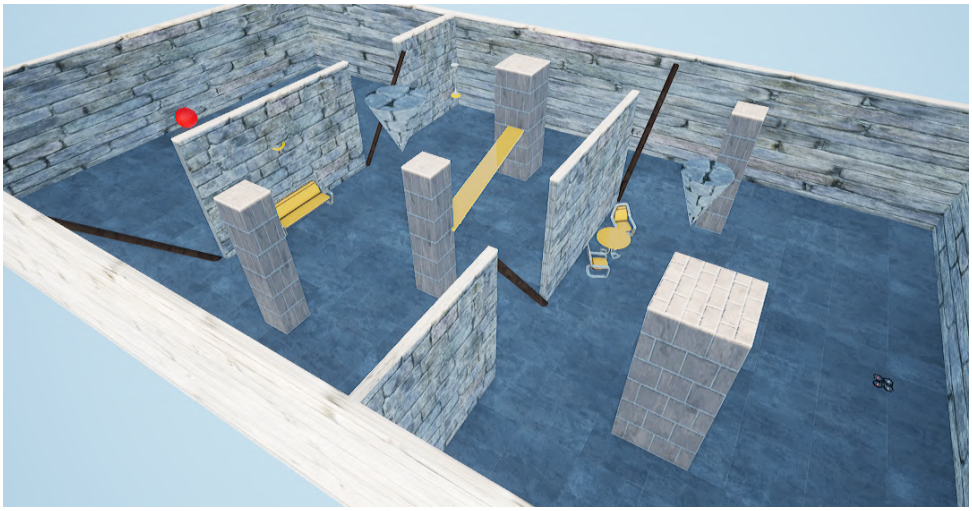


Figure 4.1: Baseline map including different obstacles.

4.1 Environment Modelling

Thanks to Unreal Engine, four handcrafted settings were created for the training phase: two static and two dynamic ones. All of them were represented as occluded rooms, so to avoid agents going outside a specified area; note that for evaluation proofs the same maps have been created with invisible roofs to deeply understand the movements of the entire swarms, without having any impact on the performances. Drones start from one side, while the objective is located oppositely: this modeling choice will be clearer when expressing the possible actions that can be taken in the algorithm.

4.1.1 Static Environments

Two static environments were created. The first one is a simple room, with different kinds of obstacles such as walls, pillars, concrete cones, shelves, and random oblique wooden pillars (Figure 4.1). It is a baseline map in which the drones were tested during training, so as to understand patterns, and be able to recognize their target, which is represented as a red ball-shaped object; a minimum distance threshold to the goal has been selected to avoid having to reach directly above it. The second static environment represents a deep forest, still contained inside a room so drones do not find tricks to avoid them using unaccepted movements outside the borders (Figure 4.2). This map has been considered as a comparison with the standard control theory paper map [41].

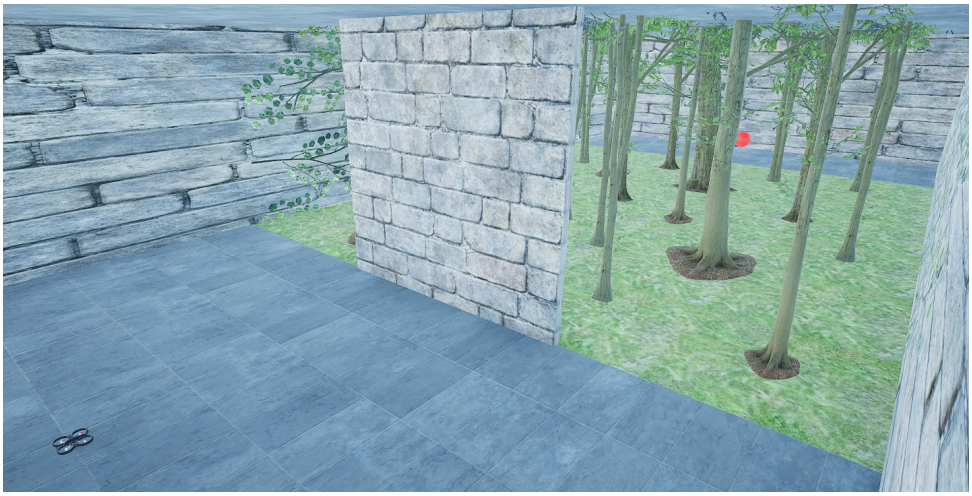


Figure 4.2: Deep forest map inspired by paper [41].

Figure 4.3: Pedestrian environment.

4.1.2 Dynamic Environments

For learning, dynamic environments were also created. One is composed of a pedestrian model: it includes several AI-moved NPCs (Non-Player Characters) roaming the map to simulate a busy pedestrian crossing (Figure 4.3). Their movement is defined by an Unreal function that makes them walk to a reachable fixed point at a given distance of the map in a continuous loop. Finally, the second dynamic map is the one used as a test map for Curriculum Learning: it includes all the different obstacles of the previous ones, such as randomly roaming NPCs, concrete pillars, and trees (Figure 4.4).

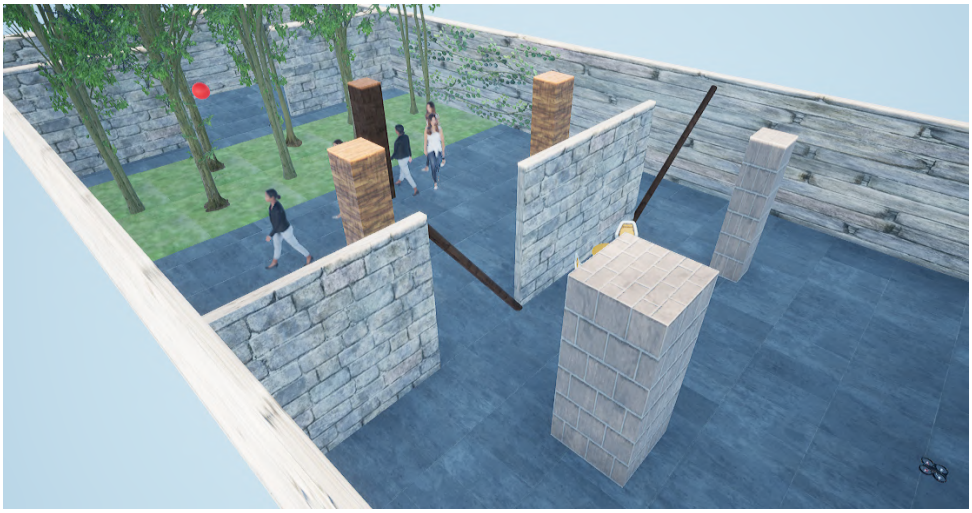


Figure 4.4: Test map which includes everything that drones had to learn.

4.2 Drone Modelling

While the results will be focused on the study of how multiple drones perform while working together collaboratively, it is crucial to understand the fundamentals of drone modeling. Thus, this includes also the understanding of the base case where the drone is only one: it works as a starting point to deeply recognize the improvements that have been made to get a better and broader result.

4.2.1 Single-Agent

The drone's body is modeled directly with the AirSim environment: it can modified to include any kind of sensor, in any position, that must be defined in a *JSON* file at start-up. In the entire project, every camera has been defined as RGB, with resolution 84×84 (inspired by the *DeepMind*'s Atari paper [1]) with a Field of View (FOV) of 120, to take into account also particular elements outside the standard 90 degrees view.

Drone behavior is wrapped in a class `gym.Env` that provides an interface with the training framework. This class is provided by *Gym* [75] (today, known as *Gymnasium*), an open-source Python library for communication between learning algorithms and environments, created by *OpenAI* and supported by a non-profit organization called *Farama Foundation*. This wrapper provides the necessary skeleton to interface in the trial-and-error process of any RL algorithm.

At the start of each episode, the drone spawns in a random y, z position that is parallel to its starting position ($x = 0$), taken as the origin of the map. This randomicity has been chosen to

prevent the drone from memorizing a specific pattern, but instead to let it find different possible pathways to the goal.
An episode is considered finished when the agent reaches the goal or collides with an obstacle.

4.2.2 Multi-Agent

The number of drones and their specification must be defined inside the *settings* file. In this case, we want all the drones equal to the one defined in the single-agent model.
As a difference, swarm instance can not be wrapped using the *Gym* package; instead, its multi-agent version framework is adopted: *PettingZoo* [76], a Python library for conducting research in MARL, created by *Farama Foundation* as an extension of *Gym*. The library provides two different APIs: one for Agent Environment Cycle (AEC) environments, specific for turn-based games, and one for Parallel environments, where agents must take actions in the same step, inspired by POMGs.
At the start of each episode, each drone starts again in a random y, z position that is parallel to its starting position ($x = 0$), but must keep a minimum distance between each other to prevent starting-time collisions.
An episode is considered finished when all agents reach the goal, collide with an obstacle, or with each other. Thus, if an agent crashes, it is removed from the episode (that is, its observation becomes a 0-matrix, and no actions will be provided) and the episode continues until its end.

4.3 Algorithm Implementation

The algorithmic approach that was implemented is the one of PPO described in Section 2.2.2. The implementation comes from an open-source Python library, *Stable-baselines3* [77], defined as a set of reliable implementations of RL algorithms in PyTorch [78]. While it offers a multitude of algorithms, ranging from on-policy and off-policy ones, such as *DQN, SAC, DDPG, TRPO, A2C*, *PPO* was chosen since it is applied in most of the related work papers and because it supports continuous actions.
Regarding the single-agent environment defined by *Gym*, it can directly interface with the algorithm framework, but the same is not possible with the multi-agent one shaped by *PettingZoo*. An additional interfacing *Farama Foundation* library is necessary: *SuperSuit*, a collection of functions that can wrap RL environments to do preprocessing to observations, rewards, and actions and can convert environments to different instances [79].

Other important multi-agent algorithmic frameworks are publicly available, but they are not exhaustively described and are still under development. The most important ones comprehend *RLlib* [80], *MARLlib* [81], and *Tianshou* [82].

Stable-baselines3 library supports multi-processing training, where agents train on n environments using n processes. The different environments act according to the same policy π and collect episodes to update and improve the policy at the same time. Anyway, it would need different instances of Unreal Engine to run at the same time, which is not possible at the time of the research. Details about the PPO training parameter settings for single and multi-agent environments can be found in Table 4.1. Anyway, hyperparameter tuning in RL is a very expensive process so most of the parameters were kept equal to the original paper [4].

Parameter	**Value**
Learning Rate	0.0001
Number of Steps	2048
Batch Size	256
Discount Factor (Gamma)	0.99
GAE Lambda	0.95
Clip Range	0.2
Clip Range VF	None
Normalize Advantage	True
Entropy Coefficient	0.0
Value Function Coefficient	0.5
Maximum Gradient Norm	0.5
Use SDE	False
SDE Sample Frequency	-1
Rollout Buffer Class	None
Rollout Buffer Parameters	None
Target KL	None
Statistics Window Size	100

Table 4.1: PPO Hyperparameters. For detailed explanation, refer to Section 2.2.2 or to the *Stable-Baselines3* official website.

4.3.1 Single-Agent

The single-agent implementation was considered as a starting product to check the reliability of the environments to interface with each other. The problem has been kept very simple and its search space is much smaller than the multi-agent case. All the elements composing the RL modeling are also less complicated just to consider it as a toy problem. It is worth citing it since the policy will converge smoothly at some point, assuring the satisfiability of the problem.

As defined in Chapter 2, what must be defined in a *DRL* problem formulation is strictly about the observation space, the action space, the reward function, and the policy function approximator.

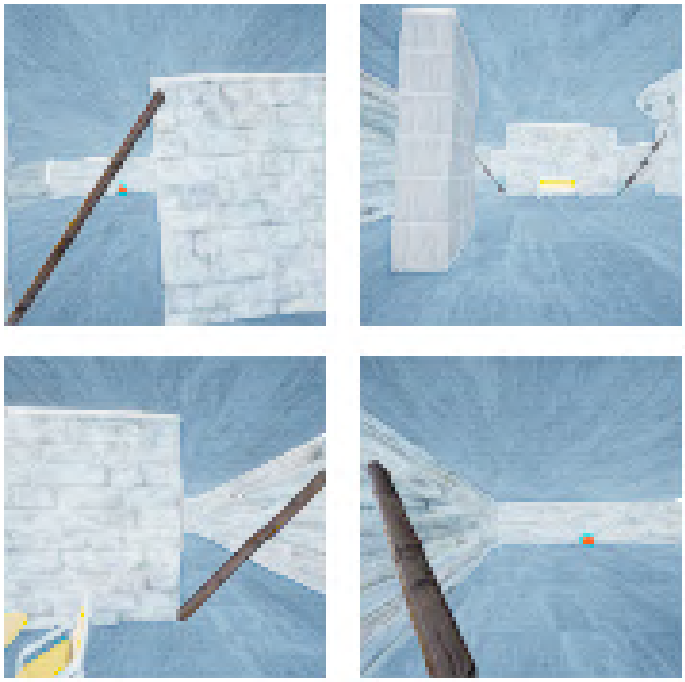


Figure 4.5: Four different training observation images on the baseline map.

Observation Space

The agent needs to get all relevant information about the current state to be able to fulfill the task successfully. The raw data that has been identified as relevant in the single-agent case is only about its camera. The camera has been chosen as RGB because of its easy and cheap industrial availability. An agent who can learn how to navigate only using an RGB camera would have outstanding implications in the research area.
The observation data o^t_{RGB} has thus been selected to be an $84 \times 84 \times 3$ matrix with values $[0, 255]$ (i.e. $o^t_{RGB} \in [0, 255]^{84\times84\times3}$); it will be normalized in the range $[0, 1]$ before being fed to the neural approximator. Image 4.5 shows an example of the perceived observations.

Action Space

In Section 1.4, the dynamics of a quad-rotor were defined: there are six degrees of freedom, three translational and three rotational. For the project, only translational degrees were considered. In

particular, the actions to be predicted strictly refer to the velocity V:

$$V = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{4.1}$$

More formally, at each step, the algorithm receives an observation plus a reward and generates a velocity V with regard to the drone's frame which lasts one time-step. Since it is a continuous action, the drone is further stopped with regard to the world's frame for a one-time step to prevent swaying.

The action values must be constrained to obtain a reliable approximation of the world. In this simple setting, v_x is fixed at 1 m/s (so, it always goes forward: the RGB camera pointing forward cannot recognize obstacles backward) while $v_y, v_z \in [-3.0, 3.0]$. Big changes in velocity can affect movement at training time, causing a sort of unnatural behavior. At evaluation time thought, the policy gives a smoothed representation of actions.

Reward Function

The reward function is an important point since it is taken as feedback by PPO to understand how chosen actions are good. Reward function shaping is challenging and critical for successful learning; many different reward functions have been investigated but only the one that will be presented can achieve satisfying results. Reward R_t is computed at each time step t, after taking the action A_t:

$$R_t = r_s + r_c + r_g \tag{4.2}$$

The reward is thus a sum of different reward functions:

- r_s denotes the reward shaped by how much the drone is approaching the goal:

$$r_s = \frac{d_i}{d_a} \tag{4.3}$$

  Where d_i is the Euclidean distance or norm between the initial random position of the drone and the goal; d_a is the Euclidean distance or norm between the position at time t of the drone and the goal.

- r_c is a constant-valued reward, given by whether the drone collided with an obstacle or not:

$$r_c = \begin{cases} -100 & \text{if agent collides} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

- r_g is a constant-valued reward, given by whether the drone reaches the goal or not:

$$r_g = \begin{cases} 100 & \text{if agent arrives} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Neural Approximator

The design of Neural Networks is a crucial point of RL: many kinds of layers, the number of layers, the layer sizes, and different activation functions, can be tried to find the best function approximation for the problem. However, it is a time-consuming process and the best way is to try models publicly available and already tested in different environments.
In Policy Gradient methods, a Neural Network is designed for both the Actor and the Critic of the architecture, only the output shape changes: the Actor, in charge of shaping the policy, decides the action so its output shape has the same shape of the number of actions to be predicted; the Critic, in charge of calculating the value function, must give an idea on how good the action predicted is. It is reasonable to share the same network, so to share the feature-extraction part of the neural network, and then fine-tune the models for their specific task.

The adopted model is inspired by the DQN paper, with minor modifications [50]. It consists of a series of three 2D convolutional layers with different sizes, kernel sizes, and strides followed each by a 2D batch normalization layer with Rectified Linear Unit (ReLU) as activation: the first convolutional has 32 as filter size, a kernel size of 8, and a stride of 2; the second has a bigger filter size of 64, 4 as kernel size and, again, stride 2; the last convolution has same filters as the previous, but smaller kernel and stride of a single unit. The result of the series is flatted and fed to a linear (or, fully connected) layer followed by a 1D batch normalization layer with ReLU. The output size has been chosen to be 512; the paper implementation is 256. Table 4.2 summarizes the network specifications.

Layer	Type	Activation	Size	Kernel	Stride	Padding
1	Conv2D		32	8	2	0
2	BatchNorm2D	ReLU	32			
3	Conv2D		64	4	2	0
4	BatchNorm2D	ReLU	64			
5	Conv2D		64	3	1	0
6	BatchNorm2D	ReLU	64			
7	Flatten					
8	Linear		512			
9	BatchNorm1D	ReLU	512			

Table 4.2: PPO Actor-Critic network.

After the CNN networks, the shared model consists of two linear layers with output size 64 and *Tanh* activation function. Moreover, the successive output layers differ in the resulting shape: the Actor model has an output shape of 2 (v_y and v_z); the Critic has an output shape of 1 (the value of the chosen action).

4.3.2 Multi-Agent

The multi-agent implementation can be considered as an extension of the single case. However, since the search space increases exponentially with the number of drones, a different representation of all the RL structures must be discovered to help the training converge to an approximate optimal solution. Communication between drones is achieved using a centralized critic structure, which provides a single action value to all the actions. The policy is, therefore, a shared one: it is learned using the observations, actions, and rewards of every agent in a unified manner, creating a model that adopts parameter sharing.

Observation Space

While an RGB camera is enough for stable learning in the single-agent case, in the multi-agent case it is not possible to obtain good learning in a satisfactory time using only observations $o^t_{RGB_i}$ received from the camera attached to each drone i. Thus, the observation space for each drone has to comprehend other information: good results have been obtained by adding the observed position $o^t_{P_i} = [x_i, y_i, z_i]$ at time t for each agent i.

As a reference, it could be useful to adopt a technique that involves stacked observations: each observation could include the last three consecutive states, to have a more meaningful representation of the environment; anyway, this resulted in poor performances.

The observation of each agent is then stacked together and fed to the Actor and Critic neural ap-

proximators. *Supersuit* interface does not support dictionaries of different observations and stacked observations, so the package had to be heavily modified manually to support it.

Action Space

The action space still refers to the three translational degrees of freedom of a quad-rotor, as in the single-agent case. Here, each agent is not constrained anymore to follow a fixed v_x velocity, but instead, this velocity must be predicted accordingly, so that agents can modify their velocity if they recognize other agents using the observation camera. This, the action space becomes:

$$V = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad v_y, v_z \in [-3.0, 3.0],\ v_x \in [0.0, 3.0] \tag{4.6}$$

v_x still can not be lower than 0, since the camera is not able to detect obstacles behind the agent.

Reward Function

In a multi-agent setting, the reward function is the one that shapes the purpose of the task. It indicates how agents have to collaborate or compete to achieve successfully their purpose. Moreover, it must take into account the behavior of different different agents and their overall performance concerning the environment and each other.
The reward function must be kept simple to give the possibility for the algorithm to find interesting patterns and correlations between actions, observations, and obtained rewards. Anyway, the multi-agent reward function is more complicated than its single counterpart.
For this problem, the approach of providing a different reward to each agent has been chosen. For each agent i, the reward R_i^t can be defined as:

$$R_i^t = r_s + r_c + r_g + r_v + r_t + r_{max} \tag{4.7}$$

The reward is, again, a sum of different reward functions:

- r_s is similar to the single-agent case and denotes the reward shaped by how much the drone

is approaching the goal:

$$r_s = \begin{cases} \frac{d_i}{d_a} \cdot v_x & \text{if } v_x > 0 \\ -3 & \textit{if } v_x = 0 \end{cases} \tag{4.8}$$

Where d_i is the Euclidean distance or norm between the initial random position of the drone and the goal; d_a is the Euclidean distance or norm between the position at time t of the drone and the goal. In this case, this reward has been chosen to be weighted using its forward velocity; in the case of a zero-velocity, not moving forward is discouraged with a slight negative reward.

- r_c is a constant-valued reward, given by whether the drone collided with an obstacle, another drone, or not:

$$r_c = \begin{cases} -100 & \text{if agent collides} \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

If a pair of agents crash with each other, this reward is given to both.

- r_g is a constant-valued reward, given by whether the drone reaches the goal or not:

$$r_g = \begin{cases} 100 & \text{if agent arrives} \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

- r_v is the reward given by the vicinity of different agents:

$$r_v = \begin{cases} -10 & \text{if } ||o^t_{P_i} - o^t_{P_j}|| \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

If the Euclidean distance or norm between the position of a pair of drones is less than a fixed threshold, then give both a small negative reward.

- r_t is a collaborative reward that indicates the performance of the swarm:

$$r_t = \begin{cases} 100 & \text{if all the agents arrive at the goal} \\ -100 & \text{if all the agents collide} \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

- r_{max} is a reward to prevent agents from getting stuck in a minimum. It is beneficial in the

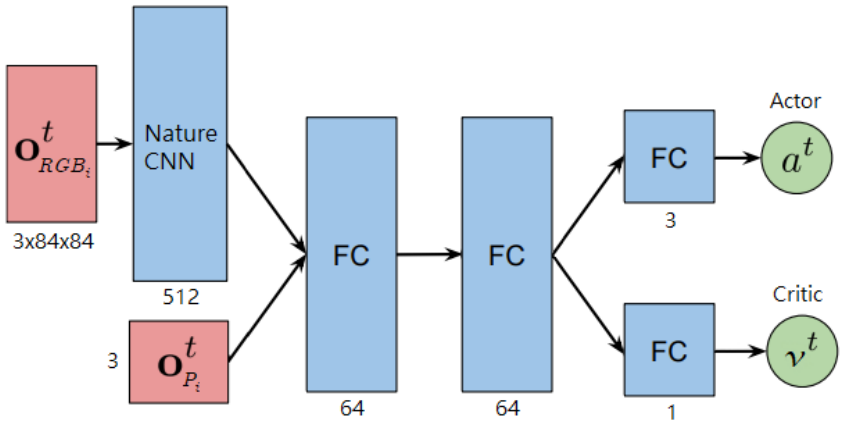


Figure 4.6: Summary of the chosen network for the multi-agent case. Stacked image observations are fed to the Nature CNN network described for the single-agent case; the resulting feature vector is concatenated with the agent's position. FC stands for fully connected, or linear, layer.

first epochs when drones are still exploring their possibilities:

$$r_{max} = \begin{cases} -200 & \text{if the current episode's step } > 100 \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

The agents' possibility of zero or near-zero forward velocity kept them swaying indefinitely at the same point. It is then worth fixing a maximum time step after which the episode ends by assigning a negative reward to all agents.

Neural Approximator

The observation space has been changed, but the neural approximator applied in the single-agent case does not work with data different than images. The feature extractor must be changed to get a reliable approximation: if the received observation is an image, just pass it through the convolutional network defined in the single-agent case; otherwise, flatten the observation (i.e. the position) and concatenate it to the flattened vector which is the output of the network in Table 4.2. The resulting output which has shape 515 (image feature extractor plus flattened position) is fed to the same linear layers described in the single-agent case; in this case, the Actor model has output with dimension 3 (v_x, v_y, v_z) while the Critic is still 1 (the value of the action). The resulting network can be summarized by Image 4.6.

Figure 4.7: Curriculum Learning design choice: the baseline map serves as a starting point to understand patterns and moving behavior; the NPC map and the forest map are used for generalization.

4.4 Curriculum Design

Curriculum Learning follows a scheme that aims to create smooth learning using presented data slightly more difficult than the previous one. This difficulty can refer to different characteristics of the data: environmental difficulty (according to the presence of objects), diversity, or relevance.
In this project, diversity has been chosen as the main asset to determine the data series provided to the swarm for efficient Curriculum Learning. The baseline map comes first, determining the majority of swarm learning. Then, the pedestrian dynamic map was chosen for the second training phase: this decision was driven by the similarity of textures presented to agents; anyway, learning how to avoid moving pedestrians is tricky and will cause the previously converged reward curve to decrease. As the last map before testing the policy, the forest was selected: the composition of the map is completely different than the previous ones with its complicated deep displacement of trees.
Baseline learning has a fixed learning rate of 0.0001; successive Curriculum Learning trials have lower learning rates to keep in memory what was learned in the past. Furthermore, the number of steps for training are lower than the one selected for the baseline. Figure 4.7 gives a clear idea of the setup; the model is trained with the reported number of steps, but, at the end of each phase, the weights that received the highest reward are kept.

5

Results and Discussion

Contents

This chapter presents and compares the same training setups for different-sized swarms. They will be evaluated in a quantitative as well as a qualitative way. During the quantitative evaluation, the training, as well as the test results, are presented and discussed. During the qualitative evaluation, the learned policy of the agents is discussed qualitatively by investigating example episodes from different environments, trying to give a possible explanation of the paths taken and if they can be considered sub-optimal. Then, the impact of the Curriculum Learning technique in the test set is analyzed. Finally, a possible real-world implementation is outlined, even if it is not the scope of this thesis.

Applying Reinforcement Learning for Navigation and Obstacle Avoidance of quad-rotors has little application available: this evaluation chapter serves as its proof of concept and starting point for further and more specialized research in the field.

5.1 Performance Analysis

In the performance analysis, I have adopted a methodology that involves initiating drones from random-fixed starting positions to rigorously test their performance across various potential start-

ing scenarios. This approach allowed me to ensure the robustness and adaptability of the system under different conditions, while ensuring replicability by fixing a chosen seed.
Each map was used both for training and testing individually, followed by a curriculum learning strategy applied on the designated test map.
To assess the impact of varying the number of drones, I compared both the time taken and the learning curves associated with each scenario. The quantitative analysis was supported by plots that highlighted key findings, including instances of early stopping and bug identification, providing insights into the reasons behind performance trends. Qualitatively, I aimed to visualize the drones' trajectories on a sort of 2-dimensional map of the environments, offering an intuitive understanding of their behavior. Through this visualization, I was able to analyze the trajectories, identify any sub optimal paths taken, and speculate on the underlying causes, thereby deepening my understanding of the system's operational dynamics.

5.1.1 Quantitative Evaluation

Case 1: 2 drones

In the scenario involving two agents constituting the swarms (Figure 5.1), all training episodes converge to an optimal policy. Specifically, in maps 1 and 3, convergence is attained around the predetermined maximum programmed steps, set at 2.5 million. However, in map 2, convergence occurs considerably earlier than the fixed maximum, requiring less than a day of training. Notably, in map 3, it is significant to highlight a limitation in training progression, attributed to the challenge of discerning the correct strategy for navigating obstacles comprised of trees. These obstacles are intricate to recognize using a low-resolution RGB camera, thereby possibly imposing a sort of halt in the middle of the training progress.

Case 2: 3 drones

In Figure 5.2, the training results for swarms consisting of three drones are highlighted, demonstrating remarkable outcomes. Across all three maps, the training algorithm managed to converge towards a near-optimal policy within approximately 2 days. It is noteworthy that when the mean reward from an evaluation episode exceeds 200, it indicates that all drones have successfully reached their objective. Interestingly, for maps 2 and 3, which are based on non-player character (NPC) interactions and forest environments respectively, the training process concluded prematurely: just

before reaching the predetermined total of 3 million steps for map 2, and remarkably at 1.5 million steps for map 3.. This again suggests that the model is capable of efficiently approximating an optimal policy in a consistent manner.

Case 3: 5 drones

Training swarms consisting of five drones proves to be more complex (Figure 5.3). Across the first 2 maps, the model eventually converges to a certain performance level (approximately between 150 and 200 as mean reward) after about 5 million steps and 4 days. However, this convergence suggests that, in most episodes, not all drones successfully reach the target together. A visual inspection of 20 evaluation episodes on maps 1 and 2 indicates that in 8 episodes, all five drones reach their destination, whereas in another 4 episodes, all drones fail, often colliding with each other (notably, drones remain visible on the map after a collision). In the remaining episodes, the number of drones that successfully arrive varies. On the other hand, in map 3, surprisingly, the training algorithm successfully managed to learn a policy which allows all the drones to reach the objective in all the evaluation episodes, already at 3 million simulation steps.

5.1.2 Qualitative Evaluation

Case 1: 2 drones

During the evaluation episodes of swarms composed of 2 drones, agents manage to collectively reach the goal 90% of the time in all maps (Figure 5.4). Notably, a bias is observed in the drone's path selection during these episodes. It seems that in each episode, drones consistently favor one path over another, potentially expediting goal attainment significantly. This bias may stem from the model's tendency to assign a positive reward whenever agents reach the objective, without adequately emphasizing exploration bonuses or time-related rewards. Furthermore, in the second map featuring moving non-player characters (NPCs), the drones display an impressive ability to hover over these NPCs (this is also observed in the other swarm size experiments). While navigating through the forest environment (here, drawing the trajectory is more challenging due to the cluttered environment created by trees), the drones adopt a line formation, strictly adhering to a single pathway deemed viable by the policy. This selective navigation suggests a potential influence of obstructed views due to tree foliage, leading to the exclusion of alternative routes. However, this behavior also highlights the drones' capacity to detect nearby agents and patiently await their

movement. On the other hand, it could also underscore the necessity for higher exploration during training, thus emphasizing the limitation of the early stopping mechanism.

Case 2: 3 drones

As illustrated in Figure 5.5, swarms consisting of three drones successfully identify and follow an optimal path towards the objective. When initiated on one side of the map, the drones efficiently navigate the shortest route to their target. This behavior contrasts with the results obtained with 2-drone swarms: with 3 drones, it appears that agents recognize that traveling together along the same path could lead to collisions more frequently, thus favoring splitting among different paths depending on their spawning positions. Moreover, this is the only result in map 3 that allows for different paths to be taken; line formation is also favored during the 5-drone optimal trajectory. Swarms composed of 3 agents thus appear to be the most successful experiments, clearly demonstrating how drones can explore different paths while still communicating with each other.

Case 3: 5 drones

The simulation of trajectories for five drones (Figure 5.6) highlights an increased complexity in achieving convergence with a larger number of agents. In fact, drones reach the objective as a group in only about 50% of the episodes. Although the drones continue to learn how to hover over NPCs in the second map, in the base map, a preference emerges for one side of the map to reach the objective, even if the drones initially take a different route. This tendency could again be attributed to insufficient exploration rates, which limit the algorithm's exploration of alternative routes. In map 3, the line formation emerges once more, as observed with swarms composed of 2 agents, although the most efficient behavior would involve splitting among the possible paths between trees.

5.2 Curriculum Learning Impact

While quantitatively speaking, Curriculum Learning seems to create a smooth learning curve while changing the training map, its qualitative result is far from outstanding. For any case previously described, the model seem to over-fit and tends to create a common phenomenon that can happen in any Machine Learning approach, called forgetting [83]. Forgetting refers to the phenomenon where a model loses or degrades its ability to recall previously learned information upon learning

new data. This issue is particularly prevalent in scenarios where models are sequentially trained on different tasks or data sets. As new information is incorporated, the model's parameters adjust to optimize performance on the recent data, which can inadvertently lead to a decrease in performance on the previously learned tasks.

Conversely, training on a single map appears to be advantageous for subsequent training sessions on different maps, acting as a form of warm start. This approach leverages the knowledge gained from the initial training environment to improve learning efficiency and model adaptability in new, unfamiliar environments. By utilizing the foundational skills and insights acquired from the first map, the model can more quickly and effectively adjust to the nuances of new maps. This strategy not only speeds up the learning process but also potentially enhances the model's overall performance, demonstrating the benefits of applying previously learned knowledge to new contexts.

Curriculum learning quantitative results in case of swarms composed of 3 agents are shown in Figure 5.7.

5.3 Simulation to reality (Sim2Real)

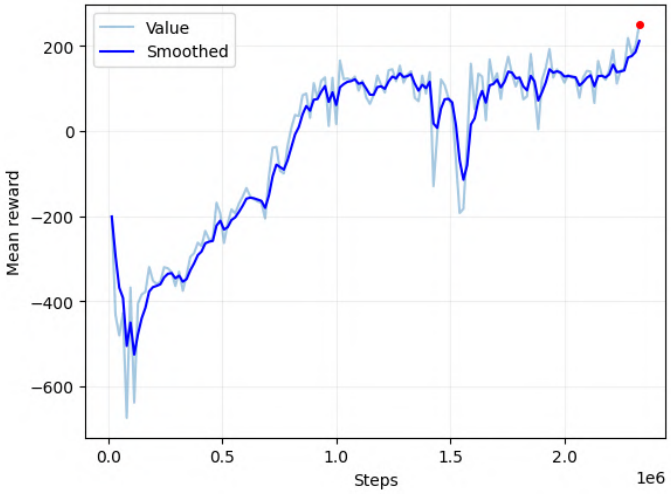
The process of moving from a simulation environment to real-world applications represents a significant challenge in robotics research. This challenge primarily arises because it is impractical to directly train robots in real-world settings, often due to the potential risks and safety concerns involved. However, for the advancements and developments made within simulation environments to be truly valuable, they must eventually be validated and applied in real-world scenarios.

A principal aspect of this transition involves using simulation tools like AirSim, which facilitate the creation of highly realistic and dynamic simulation environments for robotics research. AirSim is particularly noted for its support of ArduPilot's Copter and Rover vehicles, allowing for comprehensive simulation of these platforms. The integration of AirSim with these vehicles can be achieved through various setups, including running both on a single *Linux* system, or by running AirSim on *Windows 10* with ArduPilot operating within the Windows Subsystem for Linux (WSL). While AirSim is compatible with *MacOS*, the integration with ArduPilot on this OS remains untested and, therefore, a less explored avenue.

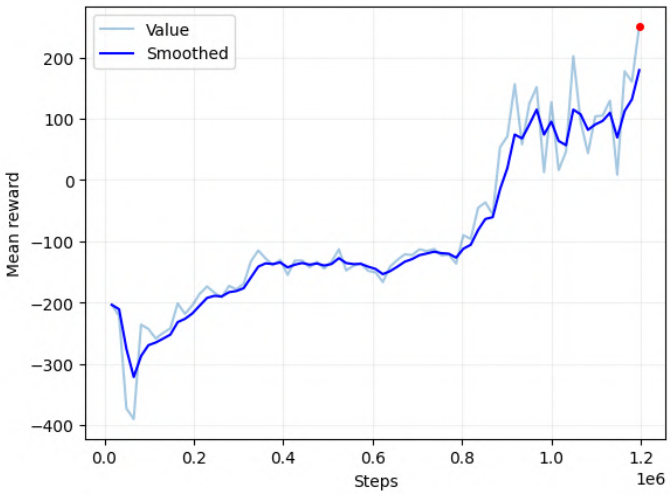
Furthermore, AirSim's capabilities extend to interaction with MavLink [84] and PX4 [74], two open-source frameworks that are instrumental in controlling actual drones. These interactions are facilitated through ROS2 messages, which can be employed in both Software in the Loop (SITL)

and Hardware in the Loop (HITL) setups. SITL simulates a vehicle's flight control hardware in code, allowing the simulation to run without any physical hardware. On the other hand, HITL involves real flight control hardware in the loop, providing a closer approximation to real-world operations. This dual capability significantly enhances the scope for testing and validating the simulated networks, ensuring that the transition from simulation to reality is as seamless and effective as possible.

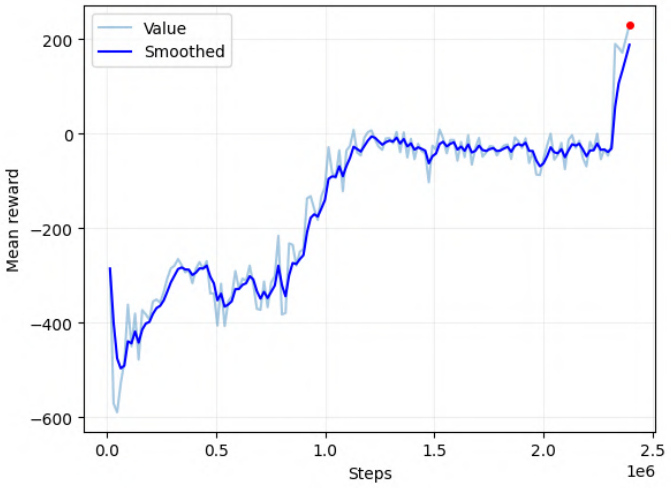
Given the complexity involved in creating and managing these sophisticated simulation environments, and their integration with real-world hardware, this aspect of robotics research falls outside the scope of the current study. Nonetheless, it underscores the importance of rigorous simulation frameworks like AirSim in bridging the gap between theoretical models and their practical, real-world applications in robotics.



(a) Mean reward training curve for two drones on map 1.

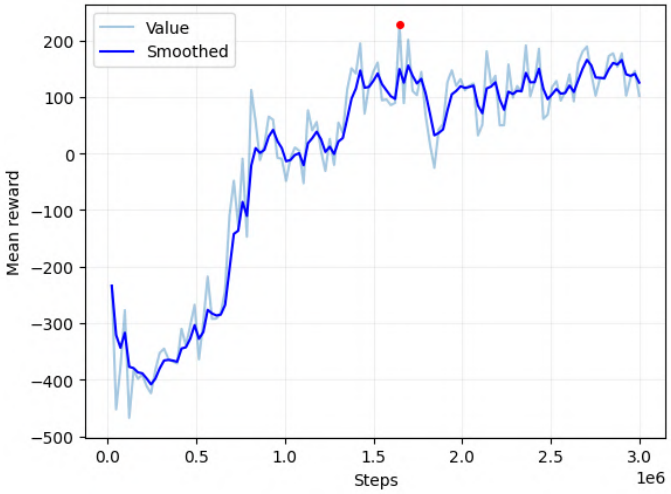


(b) Mean reward training curve for two drones on map 2.

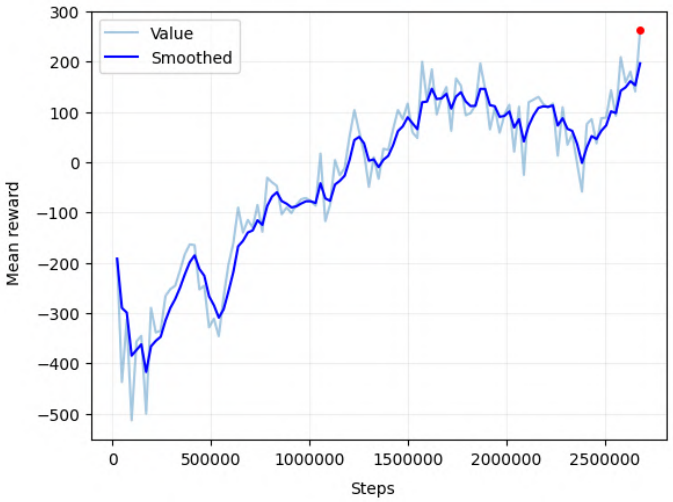


(c) Mean reward training curve for two drones on map 3.

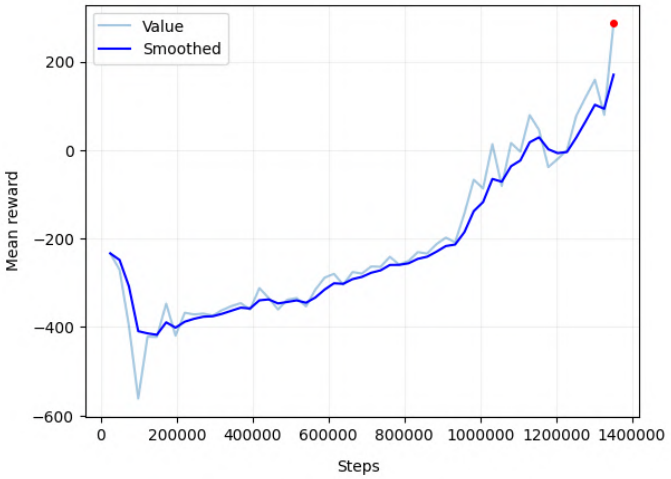
Figure 5.1: Mean reward training curves for swarms composed of 2 agents.



(a) Mean reward training curve for three drones on map 1.

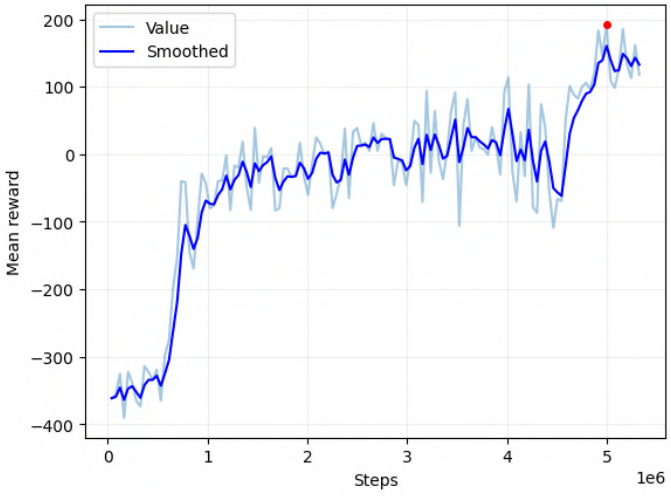


(b) Mean reward training curve for three drones on map 2.

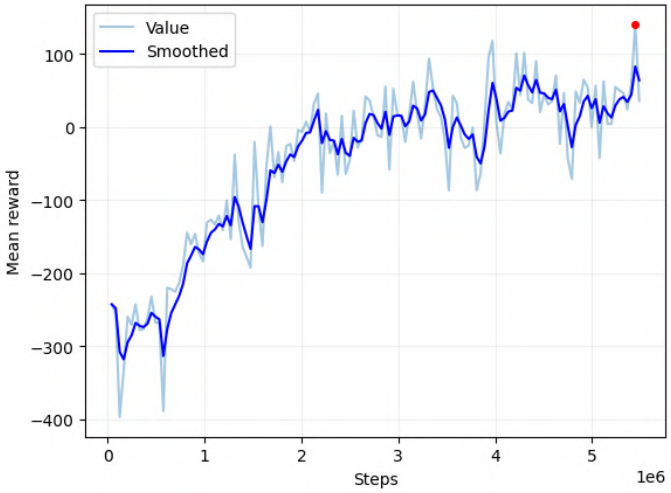


(c) Mean reward training curve for three drones on map 3.

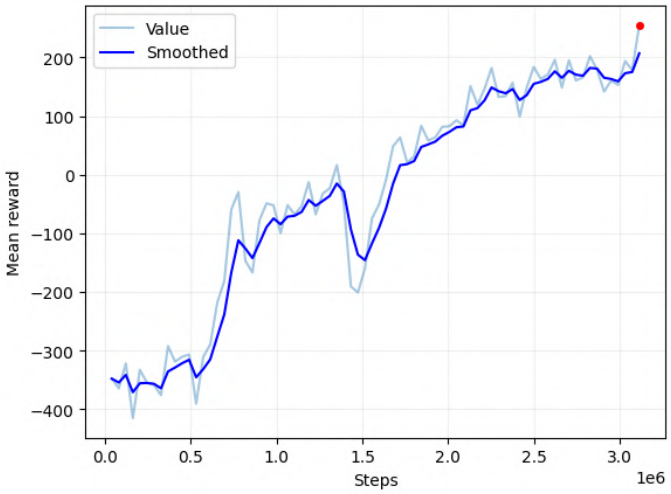
Figure 5.2: Mean reward training curves for swarms composed of 3 agents.



(a) Mean reward training curve for five drones on map 1.

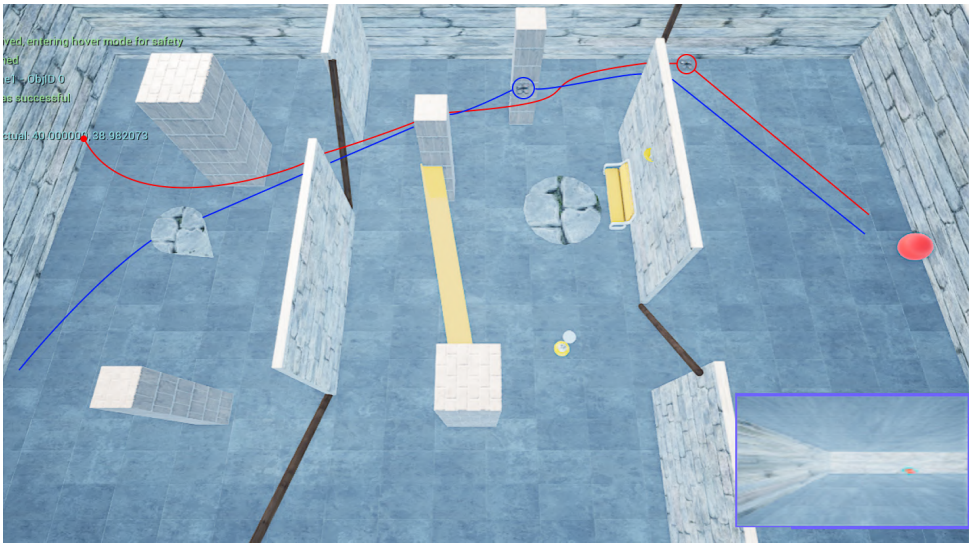


(b) Mean reward training curve for five drones on map 2.



(c) Mean reward training curve for five drones on map 3.

Figure 5.3: Mean reward training curves for swarms composed of 5 agents.

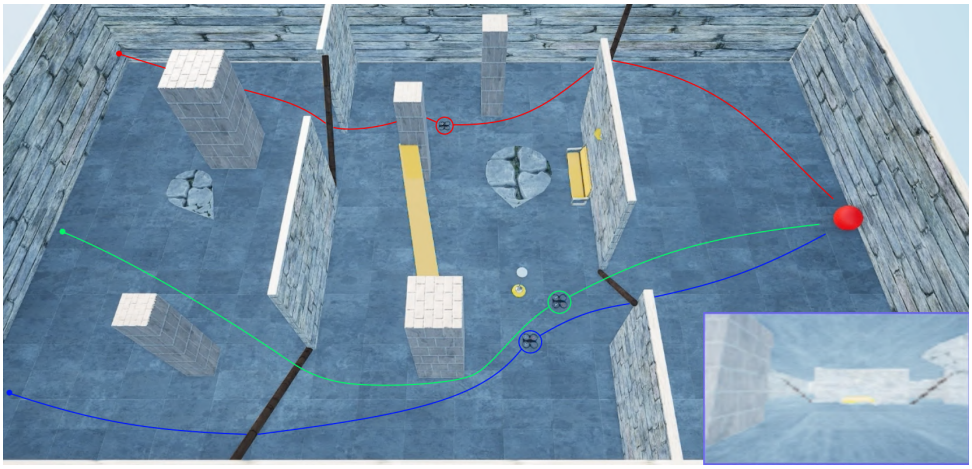


(a) Trajectory of one evaluation episode in case of 2 drones on map 1.

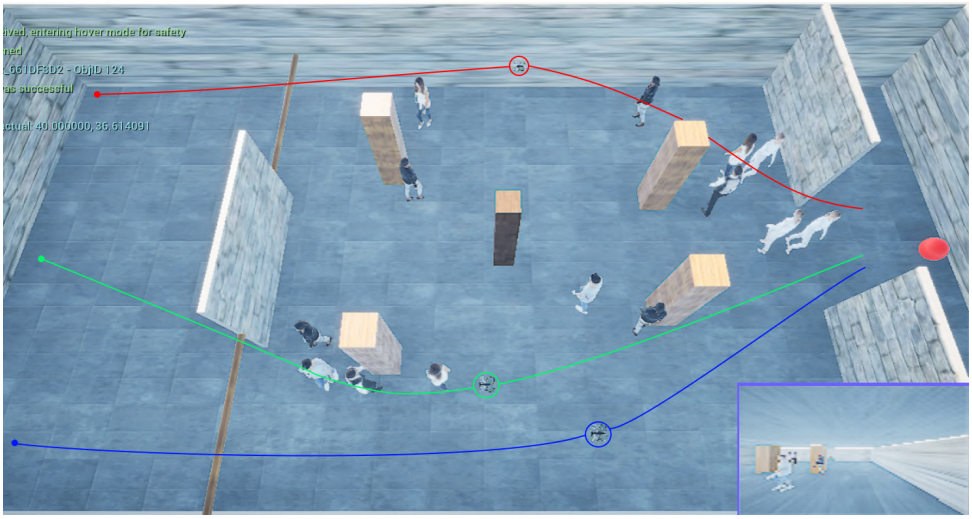
(b) Trajectory of one evaluation episode in case of 2 drones on map 2.

(c) Trajectory of one evaluation episode in case of 2 drones on map 3.

Figure 5.4: Trajectories of swarms composed of 2 agents.



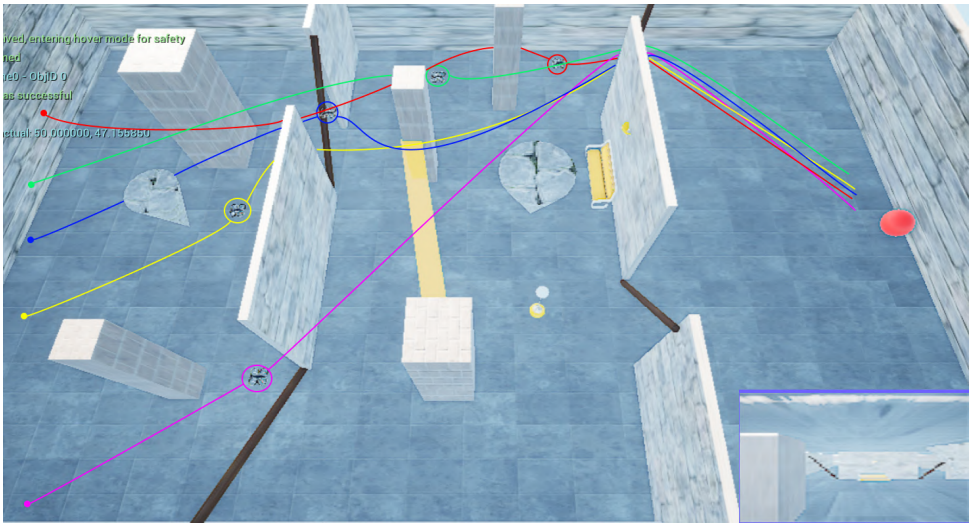
(a) Trajectory of one evaluation episode in case of 3 drones on map 1.



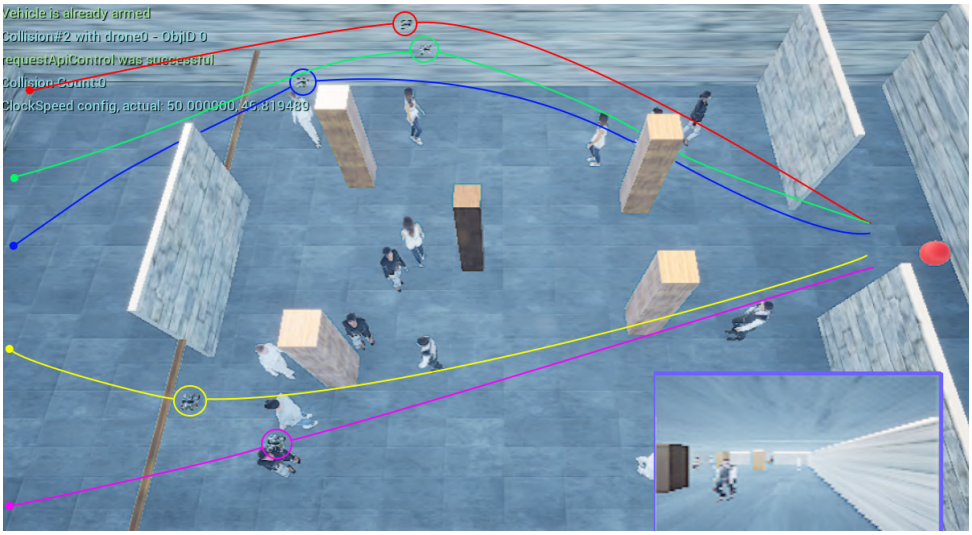
(b) Trajectory of one evaluation episode in case of 3 drones on map 2.

(c) Trajectory of one evaluation episode in case of 3 drones on map 3.

Figure 5.5: Trajectories of swarms composed of 3 agents.



(a) Trajectory of one evaluation episode in case of 5 drones on map 1.



(b) Trajectory of one evaluation episode in case of 5 drones on map 2.

(c) Trajectory of one evaluation episode in case of 5 drones on map 1.

Figure 5.6: Trajectories of swarms composed of 5 agents.

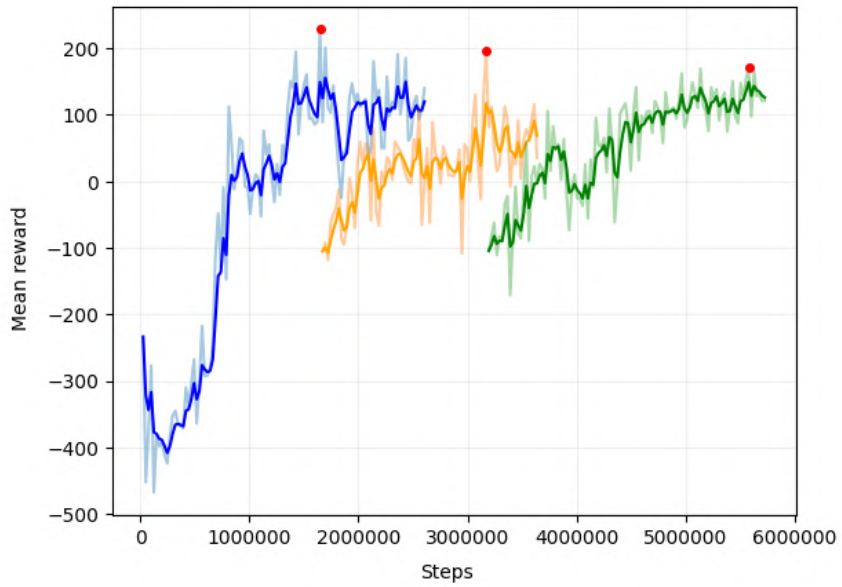


Figure 5.7: Curriculum Learning Mean Reward Graph in case of swarms composed of 3 agents: it is noteworthy how subsequent training sessions on different maps enhance the starting policy, making it superior to one with no prior training.

Conclusions and future work

This thesis has presented a comprehensive exploration into the application of Deep Reinforcement Learning (DRL) for the control of drone swarms in both static and dynamic environments, emphasizing obstacle avoidance. By implementing a novel DRL model, I have successfully demonstrated that drones are capable of autonomously learning optimal trajectories, thereby showcasing significant advancements in the field of autonomous navigation. The integration of a Proximal Policy Optimization (PPO) algorithm, complemented by a state encoding through convolutional neural networks (CNNs) and an effective reward function, has enabled the drones to exhibit emergent collective behaviors. These behaviors address crucial challenges such as agent loss and mutual avoidance, ultimately enhancing the efficiency and adaptability of drone swarms across various operational scenarios.

My findings indicate that the proposed DRL model not only achieves convergence and near-optimal trajectories in diverse environments but also offers a scalable solution for controlling drone swarms of varying sizes, if those are correctly handled. This is especially true considering the really simple state representation that has been chosen for testing.

This research contributes to the existing body of knowledge by providing a viable alternative to traditional supervised learning and classical control methods, possibly challenging the current state-of-the-art in drone navigation.

On the other hand, Curriculum Learning must be explored more to understand in which way it is possible to achieve learning and policy generalization without the need of restarting the training from scrach.

Despite the promising results, several areas have been identified for future improvement and research: a fundamental area for improvement includes addressing the computational efficiency and data requirements. The integration of more efficient learning algorithms, data reduction techniques and multiprocessing setups is essential to alleviate the computational demands and minimize the extensive training data necessary for model convergence.

Another critical avenue involves enhancing the sophistication of the state-action-reward representations (i.e. usage of different sensors as $LiDAR$ or a $NeRF$ network as state encoding) to improve

the model's adaptability. Investigating more complex representations could lead to a deeper understanding of the environment and decision-making processes, thereby facilitating more accurate and efficient navigation strategies.

The practical applicability of the proposed model in real-world scenarios, such as industrial cooperation, search and rescue operations, and environmental monitoring, is yet another crucial aspect for future exploration. Transitioning from simulated environments to real-world testing is vital for validating the effectiveness and reliability of the DRL model in operational settings, especially when real-world introduce noisy observations and signals.

Addressing the challenge of forgetting, observed during Curriculum Learning, is also imperative. Future work should focus on methods to mitigate the problem, ensuring that drones retain their performance capabilities across a variety of tasks and environments.

Exploring alternative learning paradigms, such as meta-learning or transfer learning (i.e. using as starting weights the ones provided by a supervised approach), presents an exciting opportunity for future research. These paradigms could offer new pathways to enhance the efficiency and generalization capabilities of the DRL model, enabling drones to adapt quickly to new environments based on prior knowledge and reducing the necessity for extensive retraining.

Lastly, expanding the model to accommodate multi-task learning could drastically increase the versatility and utility of drone swarms. Developing algorithms capable of optimizing multiple objectives simultaneously, such as energy efficiency, speed, and safety, would mark a significant step forward in real-time operational adaptability.

By pursuing these areas of research, future work can continue to build upon the foundation laid by this thesis, further exploring the vast potential of autonomous drone navigation and opening new horizons for the application of drone swarms in a myriad of operational contexts.

Ethical Considerations

Advancements in drone swarm technology have shown huge promises in various fields, from agriculture and disaster response to surveillance and entertainment. However, along with technological progress comes a set of ethical considerations that need careful examination. Acknowledging and addressing these concerns is imperative to ensure responsible research and deployment of drone swarms [85] [86].

- **Privacy and Surveillance**: one of the primary ethical concerns of drone swarms revolves around privacy and surveillance. Drone swarms can capture vast amounts of data, creating an increased potential for intentional intrusion into individuals' private spaces. Researchers must establish guidelines for data collection, storage, and usage to protect the privacy rights of individuals and communities.

- **Safety and Security**: the proliferation of drone swarms raises concerns regarding safety and security. Accidents, collisions, and system failures could have serious consequences, especially in densely populated areas. Implementing robust fail-safes, and establishing safety standards can mitigate potential hazards. Moreover, military drone swarms can raise ethical dilemmas such as extrajudicial killings.

- **Autonomy and Decision-Making**: drones within a swarm often operate autonomously, making real-time decisions based on algorithms and sensor data. Ethical questions arise concerning the level of autonomy these systems possess and the potential for human intervention. A balance between automated decision-making and human oversight is essential to ensure accountability and prevent unintended consequences.

- **Equity and Accessibility**: ensuring equal access to drone swarm technology is a crucial ethical consideration. It is imperative to guard against existing social and economic disparities. Efforts should be made to democratize access, promote inclusivity, and prevent the technology from becoming a tool of exclusion or inequality.

- **Legal and Regulatory Compliance**: navigating the complex legal and regulatory landscape surrounding drone swarm technology is vital. Researchers and developers must stay

abreast of evolving laws and guidelines governing drone operations. Adherence to these regulations is not only a legal obligation but also an ethical imperative to ensure the responsible and lawful use of the technology.

In conclusion, the ethical considerations surrounding drone swarm research demand meticulous attention. By addressing these concerns, researchers can promote an environment of responsible innovation and deployment. Collaborative efforts among academia, industry, and regulatory bodies are essential to balance technological advancement and ethical stewardship.

Bibliography

[1] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] C. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, Mar. 2012. DOI: 10.1109/TCIAIG.2012.2186810.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] M. Hessel, J. Modayil, H. Van Hasselt, *et al.*, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[6] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, pp. 229–256, 1992. DOI: 10.1007/BF00992696. [Online]. Available: https://doi.org/10.1007/BF00992696.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, PMLR, 2015, pp. 1889–1897.

[9] V. Mnih, A. P. Badia, M. Mirza, *et al.*, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[10] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*, PMLR, 2018, pp. 1861–1870.

[12] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*, PMLR, 2018, pp. 1587–1596.

[13] R. Sutton and A. Barto, *Reinforcement Learning, second edition: An Introduction* (Adaptive Computation and Machine Learning series). MIT Press, 2018, ISBN: 9780262039246. [Online]. Available: https://books.google.co.jp/books?id=5s-MEAAAQBAJ.

[14] M. Tan, "Multi-agent reinforcement learning: Independent versus cooperative agents," in *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993*, P. E. Utgoff, Ed., Morgan Kaufmann, 1993, pp. 330–337. DOI: 10.1016/b978-1-55860-307-3.50049-6. [Online]. Available: https://doi.org/10.1016/b978-1-55860-307-3.50049-6.

[15] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems*, vol. 30, 2017.

[16] T. Rashid, M. Samvelyan, C. S. De Witt, G. Farquhar, J. Foerster, and S. Whiteson, "Monotonic value function factorisation for deep multi-agent reinforcement learning," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7234–7284, 2020.

[17] C. Yu, A. Velu, E. Vinitsky, *et al.*, "The surprising effectiveness of ppo in cooperative multi-agent games," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 611–24 624, 2022.

[18] P. Abbeel, A. Coates, M. Quigley, and A. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in Neural Information Processing Systems*, B. Schölkopf, J. Platt, and T. Hoffman, Eds., vol. 19, MIT Press, 2006.

[19] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain," *Science robotics*, vol. 5, no. 47, eabc5986, 2020.

[20] J. Peters, S. Vijayakumar, and S. Schaal, "Reinforcement learning for humanoid robotics," *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pp. 1–20, Jan. 2003.

[21] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: Lessons we have learned," *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 698–721, 2021.

[22] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2017, pp. 3389–3396.

[23] H. Surmann, C. Jestel, R. Marchel, F. Musberg, H. Elhadj, and M. Ardani, "Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments," *arXiv preprint arXiv:2005.13857*, 2020.

[24] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, Sep. 2013. DOI: 10.1177/0278364913495721.

[25] O. M. Andrychowicz, B. Baker, M. Chociej, *et al.*, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.

[26] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[27] M. Bojarski, D. Del Testa, D. Dworakowski, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[28] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[29] A. Kendall, J. Hawke, D. Janz, *et al.*, "Learning to drive in a day," in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8248–8254.

[30] M. Hüttenrauch, S. Adrian, G. Neumann, *et al.*, "Deep reinforcement learning for swarm systems," *Journal of Machine Learning Research*, vol. 20, no. 54, pp. 1–31, 2019.

[31] M. Hüttenrauch, A. Šošić, and G. Neumann, "Guided deep reinforcement learning for swarm systems," *arXiv preprint arXiv:1709.06011*, 2017.

[32] P. Long, T. Fan, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning," in *2018 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2018, pp. 6252–6259.

[33] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 7512–7519.

[34] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[35] A. Rodriguez-Ramos, C. Sampedro, H. Bavle, P. D. L. Puente, and P. Campoy, "A deep reinforcement learning strategy for uav autonomous landing on a moving platform," *Journal of Intelligent and Robotic Systems*, 2019.

[36] R. Polvara, M. Patacchiola, S. Sharma, *et al.*, "Autonomous quadrotor landing using deep reinforcement learning," *arXiv preprint arXiv:1709.03339*, 2017.

[37] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2021, pp. 1205–1212.

[38] D. Scaramuzza and F. Fraundorfer, "Visual odometry part i: The first 30 years and fundamentals," 4, vol. 18, 2011, pp. 80–92. DOI: 10.1109/MRA.2011.943233.

[39] S. Wang, R. Clark, H. Wen, and N. Trigoni, "Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks," in *2017 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2017, pp. 2043–2050.

[40] M. Memmel, R. Bachmann, and A. Zamir, "Modality-invariant visual odometry for embodied vision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 21 549–21 559.

[41] A. Loquercio, E. Kaufmann, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza, "Learning high-speed flight in the wild," *Science Robotics*, vol. 6, no. 59, eabg5810, 2021.

[42] X. Zhou, J. Zhu, H. Zhou, C. Xu, and F. Gao, "Ego-swarm: A fully autonomous and decentralized quadrotor swarm system in cluttered environments," in *2021 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2021, pp. 4101–4107.

[43] X. Zhou, X. Wen, Z. Wang, *et al.*, "Swarm of micro flying robots in the wild," *Science Robotics*, vol. 7, no. 66, eabm5954, 2022. DOI: 10.1126/scirobotics.abm5954. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm5954.

[44] S. Wolfram, D. Peter, and P. R. Montague, "A neural substrate of prediction and reward," *Science*, vol. 275, no. 5306, pp. 1593–1599, Mar. 1997, ISSN: 0036-8075. DOI: 10.1126/science.275.5306.1593. [Online]. Available: https://cir.nii.ac.jp/crid/1364233268518617216.

[45] P. P. Read, Montague, P. Dayan, *et al.*, "A framework for mesencephalic dopamine systems based on predictive hebbian learning," in *Journal of Neuroscience*, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:224172.

[46] G. Dichter, C. Damiano, and J. Allen, "Reward circuitry dysfunction in psychiatric and neurodevelopmental disorders and genetic syndromes: Animal models and clinical findings," *Journal of neurodevelopmental disorders*, vol. 4, p. 19, Jul. 2012. DOI: 10.1186/1866-1955-4-19.

[47] G. Hoffmann, H. Huang, S. Waslander, and C. Tomlin, "Quadrotor helicopter flight dynamics and control: Theory and experiment," Aug. 2007. DOI: 10.2514/6.2007-6461.

[48] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*, Springer, 2018, pp. 621–635.

[49] R. J. Williams and L. C. Baird, "Analysis of some incremental variants of policy iteration: First steps toward understanding actor-cr," 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:18786951.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[51] J. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997. DOI: 10.1109/9.580874.

[52] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[53] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[54] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1995–2003.

[55] D. Joel, Y. Niv, and E. Ruppin, "Actor–critic models of the basal ganglia: New anatomical and computational perspectives," *Neural Networks*, vol. 15, no. 4, pp. 535–547, 2002, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(02)00047-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608002000473.

[56] Ö. Şimşek, S. Algorta, and A. Kothiyal, "Why most decisions are easy in tetris—and perhaps in other sequential decision problems, as well," in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings

of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1757–1765.

[57] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

[58] G. Owen, *Game Theory*. Emerald Group Publishing Limited, 2013, ISBN: 9781781905074. [Online]. Available: https://books.google.co.jp/books?id=OfnLkgEACAAJ.

[59] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ser. ICML'94, New Brunswick, NJ, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 157–163, ISBN: 1558603352.

[60] T. W. Sandholm and R. H. Crites, "Multiagent reinforcement learning in the iterated prisoner's dilemma," *Biosystems*, vol. 37, no. 1, pp. 147–166, 1996, ISSN: 0303-2647. DOI: https://doi.org/10.1016/0303-2647(95)01551-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0303264795015515.

[61] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The complexity of decentralized control of markov decision processes," *Mathematics of operations research*, vol. 27, no. 4, pp. 819–840, 2002.

[62] A. Wong, T. Bäck, A. V. Kononova, and A. Plaat, "Deep multiagent reinforcement learning: Challenges and directions," *arXiv preprint arXiv:2106.15691*, 2021.

[63] E. A. Hansen, D. S. Bernstein, and S. Zilberstein, "Dynamic programming for partially observable stochastic games," in *AAAI*, vol. 4, 2004, pp. 709–715.

[64] P. Palanisamy, "Multi-agent connected autonomous driving using deep reinforcement learning," in *2020 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2020, pp. 1–7.

[65] J. Z. Leibo, E. Hughes, M. Lanctot, and T. Graepel, "Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research," *arXiv preprint arXiv:1903.00742*, 2019.

[66] J. Hu and M. P. Wellman, "Nash q-learning for general-sum stochastic games," vol. 4, pp. 1039–1069, Dec. 2003, ISSN: 1532-4435.

[67] P. Sunehag, G. Lever, A. Gruslys, *et al.*, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[68] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[69] Y. Yang, R. Luo, M. Li, M. Zhou, W. Zhang, and J. Wang, "Mean field multi-agent reinforcement learning," in *International conference on machine learning*, PMLR, 2018, pp. 5571–5580.

[70] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09, Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 41–48, ISBN: 9781605585161. DOI: 10.1145/1553374.1553380. [Online]. Available: https://doi.org/10.1145/1553374.1553380.

[71] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, and P. Stone, "Curriculum learning for reinforcement learning domains: A framework and survey," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7382–7431, 2020.

[72] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

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

[74] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074.

[75] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[76] J. Terry, B. Black, N. Grammel, *et al.*, "Pettingzoo: Gym for multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 15 032–15 043, 2021.

[77] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[78] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[79] J. K. Terry, B. Black, and A. Hari, "Supersuit: Simple microwrappers for reinforcement learning environments," *arXiv preprint arXiv:2008.08932*, 2020.

[80] E. Liang, R. Liaw, R. Nishihara, *et al.*, "Rllib: Abstractions for distributed reinforcement learning," in *International conference on machine learning*, PMLR, 2018, pp. 3053–3062.

[81] S. Hu, Y. Zhong, M. Gao, *et al.*, "Marllib: Extending rllib for multi-agent reinforcement learning," *arXiv preprint arXiv:2210.13708*, 2022.

[82] J. Weng, H. Chen, D. Yan, *et al.*, "Tianshou: A highly modularized deep reinforcement learning library," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 12 275–12 280, 2022.

[83] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," *arXiv preprint arXiv:1312.6211*, 2013.

[84] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, "Micro air vehicle link (mavlink) in a nutshell: A survey," *IEEE Access*, vol. 7, pp. 87 658–87 680, 2019.

[85] D. B. Resnik and K. C. Elliott, "Using drones to study human beings: Ethical and regulatory issues," *Science and Engineering Ethics*, vol. 25, no. 3, pp. 707–718, 2019. DOI: 10.1007/s11948-018-0032-6.

[86] N. Wang, M. Christen, and M. Hunt, "Ethical considerations associated with "humanitarian drones": A scoping literature review," *Science and Engineering Ethics*, vol. 27, pp. 1–21, Aug. 2021. DOI: 10.1007/s11948-021-00327-4.