POLITECNICO DI TORINO

MASTER's Degree in DATA SCIENCE AND ENGINEERING

MASTER's Degree Thesis

Model-Free Multi-Agent Reinforcement Learning Approach in NeurIPS LuxAI S3 Competition

Supervisors

Prof. Daniele APILETTI

Dr. Simone MONACO

Dr. Daniele REGE CAMBRIN

Candidate

Paolo RIZZO

Academic Year 2024-2025

Abstract

This thesis investigates the application of Multi-Agent Reinforcement Learning (MARL) to the development of a robust and adaptive agent able to interact with a partially observable and continuously evolving environment, while competing against other agents in order to achieve winning conditions.
With the widespread adoption of deep learning, Reinforcement Learning (RL) has gained lots of popularity in the last decade, scaling to previously intractable problems, such as playing complicated games from pixel observations, sustaining conversations with humans and autonomous driving. However, there is still a wide range of domains inaccessible to RL due to the high computational cost of training or unfeasibility of agent convergence for complex problems. Therefore, the NeurIPS (Conference on Neural Information Processing Systems) LuxAI competition has become a significant event within the scientific community, serving as a platform for advancing research at the intersection of artificial intelligence, robotics, and human-robot interaction.
The season 3 competition revolves around testing the limits of agents when it comes to adapting to a game with changing dynamics. In particular, the player agent competes against an opponent agent in several matches, controlling multiple sub-agents and performing a continuous trade-off between exploration of a random environment with partial observability and exploitation of the current information to maximize the target objective.
The thesis first provides an overview of the main challenges of MARL paradigm, like non-stationarity, equilibrium selection, credit assignment and the scaling to many agents. Then, it follows the comparison of state-of-the-art algorithms and explanation of the architecture used. In addition, it's underlined that the model is developed as agnostic and trained with self-play, meaning that no previous knowledge is instilled and strategies are learnt indipendently, in contrast to traditional rule-based models which leverage on human heuristics to reason and take action. Finally, the thesis evaluates the performances of the model and shows the position reached in the competition ranking.

Acknowledgements

to...

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In the last decade, reinforcement learning (RL) has established itself as a powerful technique for solving complex decision-making problems in dynamic environments. Traditionally, RL focuses on the learning process of a single agent interacting with an environment. However, many real-world problems involve multiple agents interacting with each other and with the environment simultaneously. These settings, known as Multi-Agent Reinforcement Learning (MARL), present unique challenges and opportunities, offering a more complex yet highly relevant framework for addressing problems that occur in multi-agent systems.
A multi-agent system (MAS) consists of multiple autonomous entities, or agents, that operate in a shared environment, often with conflicting goals, cooperation requirements, or competition. These agents must make decisions based on local observations while considering the actions and strategies of other agents, which makes the problem of learning in such systems inherently more complex. MARL is the extension of traditional RL that specifically deals with scenarios where multiple agents learn concurrently and potentially interact with one another. The agents may pursue individual goals or work together to achieve a shared objective, depending on the problem domain.

The application of MARL to real-world problems is particularly valuable in domains where the behavior of multiple agents impacts the outcome of the system. For example, in autonomous vehicle systems, multiple vehicles must navigate through roads, interacting with each other to ensure safety and efficient traffic flow. Similarly, in robotics, multi-robot systems often need to collaborate to accomplish tasks such as search-and-rescue missions, warehouse management, or manufacturing automation. Other applications span healthcare, communication networks, energy grids, and supply chains, all of which benefit from the coordinated decision-making of multiple agents working together or in competition.

Despite of the great potential for solving these complex, real-world challenges, it also presents significant hurdles. The primary challenge is the non-stationarity of multi-agent environments. In traditional RL, the environment is typically static from the perspective of the agent, but in MARL, the actions of other agents dynamically change the environment, creating a constantly evolving state space. Furthermore, MARL requires high computational power for the optimization of each agent, more than single RL paradigm, questioning directly the trade-off performance/scalability.

To foster research progress and development of innovative solutions when addressing MARL challenges, NeurIPS (Conference on Neural Information Processing Systems) promotes a series of competitions, under the name of LuxAI, whose pillars are real-time decision making in dynamic environments, resource management and interaction of agents through collaboration and competition. In particular, the third season, LuxAI S3, has been taken in exam for the sake of the thesis and it poses the attention on the adaptability of two teams in complex dynamic multi-agent environment. The competition is structured by 1vs1 games, where one player's agent controls 16 sub-agents and competes autonomously against another player's agent in a environment with changing dynamics, while balancing cooperative strategies and greedy behaviors to pursue a victory condition.

Putting aside the ludic purpose, the thesis focus is to address the main challenges of MARL paradigm and developing a stable architecture which faces adaptively the continuous evolution of environment dynamics. More specifically, the aim is to propose a solution which can overcome rule-based solutions, i.e. solutions which implement human-crafted heuristics, while maintaining an agnostic fashion, using only Self-Play knowledge without additional external knowledge. Furthermore, the architecture is developed implementing two popular algorithms, DQN and A2C, upon the concept of Homogeneity of Agents, which relaxes the MARL problem complexity and make the learning process computationally faster and scalable to many agents. Since the proposed solution is labeled as Model-Free, meaning that it doesn't require the transition probability information of an environment to be implemented, it can be shifted to other domains which respect the homogeneity condition.

The thesis consists of other four chapters. Chapter 2 dives into a technical digression of RL and MARL, presenting their strength points and current limitations, showing as well the state-of-the-art algorithms in cooperative/competitive environments. Chapter 3 explains the details of LuxAI S3 competition, frameworks and methodology adopted during solution development. Chapter 4 shows the results obtained and the competition ranking. Chapter 5 is the conclusion, with a brief recap and discussion over possible improvements to conclude the dissertation.

Chapter 2

Background

This chapter covers the state-of-the-art and the theory beneath the methodologies described in the next chapter. The discussion gives a brief introduction of Machine Learning and its main branches. It follows a deep dissertation of Reinforcement Learning and, for extension, Multi-Agent Reinforcement Learning, underlining on the one side the strenghts with respect to traditional Machine Learning and Deep Learning techniques and, on the other side, the weaknesses and open problems. Finally, an overlook of main neural network architectures is proposed to better comprehend policy approximation.

2.1 Machine Learning

"How can computers learn to solve problems without being explicitly programmed?". *"Machine learning"* was the answer Arthur Samuel in 1959, the IBM engineer who firstly coined the term[1].
This simple answer embodies the capacity of artificial intelligence to perform tasks without explicit instructions by human-programmer.

At its core, machine learning is the study and development of statistical algorithms that can learn from data and generalize to unseen data, taking automated decisions and making judgments. In particular, the goal of algorithm modeling can normally assume two types of nature: 1) accurately predict some future quantity of interest, given some observed data and 2) to discover unusual or interesting pattern in the data. To achieve these goals, the model must rely on three important pillars of the mathematical sciences[2].
Function approximation. Since data is the cornerstone of machine learning, the first step is to understand which relationships there are between variables and the most natural way is through a map or a function. This function is not

known in most of the cases but it can be approximated if given enough data and computational power.
Optimization. Given a class of mathematical methods, the aim is to find the most suited model in that class. This step tailors the best fitting function to the observed data, leveraging on optimization algorithms and efficient programming.
Probability. The observed data is interpreted as random variables which are ingested by the model to output a response; this realization, under probability law, identifies the accuracy of the model itself and prospects how reactive the model will be in formulating new prediction in the future.

Despite the multitudes and countless forms of machine learning, it's possible to exemplify the taxonomy in three main branches: Supervised Learning, Unsupervised Learning and Reinforcement Learning.

2.1.1 Supervised Learning

Given an input or *feature* vector x, a supervised model will generate an output or *response* variable y. The ability of the model to output correct prediction depends on a process called training, where the algorithm searches for a function f able to encompass all the relationships between feature and response.

In detail, the main actors of the learning process are[3]:

- Feature vector $X = (x_0, ..., x_n) \in R^n$ belonging to a *domain set D*
- Label set $Y = \{y_0, ..., y_n\}$
- Training data $S = ((x_0, y_0), ..., (x_n, y_n))$
- Prediction rule or hypothesis $f : X \to Y$
- Loss function $L(y, \hat{y})$ which compares the expected output y and the prediction $\hat{y}$

Depending on the cardinality of the label set Y, the problem addressed by a supervised model can be *Classification* or *Regression.* Both approaches require labeled data for training but differ in their objectives: classification aims to find decision boundaries that separate discrete classes, whereas regression focuses on finding the best-fitting line to predict numerical outcomes in a continuous support. It's obvious that the variety of problems covered by this branch of learning is practically unlimited. Detecting if an email is spam and evaluating if a picture represents a cat or dog are basic examples of classification. Predicting weather temperature and forecasting stock prices are, instead, all examples of regression tasks.

As in Figure 2.1, training data serves as an instructor by giving the algorithm instances to work with. In general, we suppose that all samples from training set are identically and independently distributed (*i.i.d. assumption*), according to the distribution of domain set D; this assumption shall hold since training set aims to represent as closely as possible the domain set and it shall reflect its nature without introducing additional biases. Once training set is built, the algorithm runs iteratively, comparing prediction and expected labels, and adjusts the internal parameters of hypothesis function f to minimize the error of loss function L. The algorithm is then evaluated on a validation set to verify if the model is effectively able to generalize on unseen data or, instead, has reached the condition of *overfitting*, meaning that it cannot adequately capture the underlying structure of data different from training set. Here, a hyperparameter optimization, or hyperparameter tuning, determines the set of hyperparameters that yields an optimal response of model and minimize the loss function. The final step is to test the model on the test set which is completely independent of the training set.

Depending on the complexity and nature of the task, countless typologies of algorithms may serve the scope. Tree-based models (Decision Tree, Random Forest), Halfspace predictors (SVM), Linear Regressors, Deep Neural Networks are small indicators of the vast magnitude of solutions for supervised learning problems.

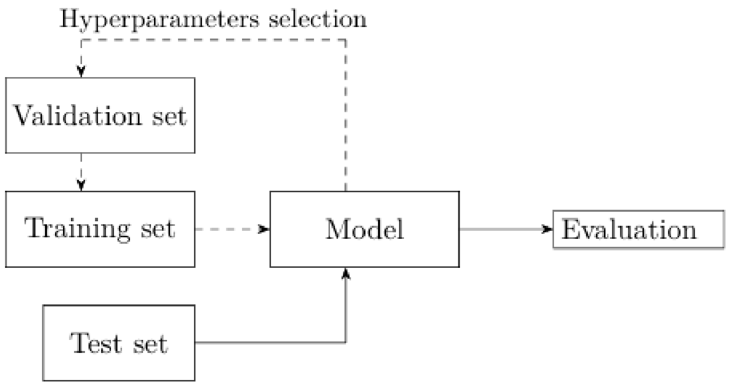


Figure 2.1: Generic example of Supervised Learning workflow. The model is trained on training set and evaluated on validation set; following the results of validation, the selection of best hyperparameters is conducted, also called hyperparameter tuning. Once the model has reached its best performances or simply converges, it's tested on the test set. Source:[3]

2.2 Reinforcement Learning

Compared to supervised and unsupervised learning, Reinforcement Learning represents a unique shape of machine learning, due to a completely different approach in modeling and handling problems. The main actor in this case is an agent which learns to make decisions by interacting with an environment, aiming to maximize cumulative reward over time. The learning process regarding which action suits the most is dynamic and, foremost, takes into account not only the immediate impact of the action but also the long term affect on the environment. Thus, trial-and-error and delayed reward are the two main peculiar characteristics of reinforcement learning. In the last decades, various fields has benefited from it, such as robotics, autonomous driving, finance, game theory and many others are currently experimenting, like natural language processing with the new Reasoning Language Models (RLM), a more powerful version of LLMs.
The following sections will give a deep understanding of the fundamentals of reinforcement learning and they take lots of inspiration from [4] for descriptive paragraphs, since it's arguably the most authoritative source for this topic.

2.2.1 Nomenclature and Elements of RL

The purpose of Reinforcement Learning is to make the agent learn how to act and behave beneficially, with the focus on getting the highest possible reward from the environment. Every problem of this kind can be shaped through these following components[4]:

- **Environment**: the world which the problem takes place.
- **State** s_t: the representation of environment at certain time or step t.
- **Agent**: the entity which interacts with the environment. In complex environments, there may be several agents or a combination of one agent and multiple sub-agents.
- **Action** a_t: interaction of the agent with the environment. It's chosen from a set which can be discrete or continuous, depending on the cardinality.
- **Observation** o_t: snapshot of environment seen by the agent at certain time or step t. Not all information of the environment are passed to the agent and that's the main difference between state and observation. If state and observation overlap, it's called *complete observation*, otherwise *partial observation.*

- **Reward** r_t: scalar value sent to the agent after every update of the environment. It's customized by the environment or by the programmer and embodies the goodness of an action. The metaphor *"carrot and stick"* gives a clear idea of the reward functioning: good behaviors receives treats (positive reward), bad behaviors receives punishments (negative rewards).

- **Trajectory** τ: the path of the agent through the state space up until the horizon H. It's a sequence over continuous time steps, from t to $t+h$, and it's expressed as concatenation of state-action-reward.

$$\tau_t = s_t, a_t, r_t, ..., s_{t+h}, a_{t+h}, r_{t+h} \tag{2.1}$$

- **Return** g_t: cumulative reward according to the trajectory. The agent's purpose is to maximize the long term reward feedback thus it shall take into account not a small immediate effect but rather a greater benefit later: such long term benefit is calculated as the discounted sum rewards during time. The discounted factor γ underlines how valuable future rewards are take into account in the immediate present: the higher the factor, the greater the impact.

$$g_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad \gamma \in [0,1) \tag{2.2}$$

- **Policy** π: learning agent's way of behaving at a given time. It's the core element in RL and a mapping function that dictates which action a_t is performed given an input state s_t. It may be deterministic or stochastic. In the latter case, the objective of RL problem will be to find the optimal policy π^* that returns the greater cumulative reward in future timestamps;

$$a_t = \pi_{deterministic}(s_t) \tag{2.3}$$

$$\pi_{stochastic}(a_t \mid s_t) = \mathbb{P}[a_t \mid s_t] \tag{2.4}$$

- **State Value function** $V^\pi(s)$: it estimates the goodness of a state s according to the policy π. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.

$$V^\pi(s_t) = \mathbb{E}_{\tau\sim\pi}\left[g_t \mid s = s_t\right] \tag{2.5}$$

- **Action Value function** $Q^\pi(s, a)$: it estimates the goodness of an action a taken in state s according to the policy π. The value of an action is the total amount of reward an agent can expect to accumulate over the future, starting from that state after taking that action.

$$Q^\pi(s_t, a_t) = \mathbb{E}_{\tau\sim\pi}\left[g_t \mid s = s_t, a = a_t\right] \tag{2.6}$$

- **Model**: it mimics the behavior of the environment and allows inferences to be made about how the environment will behave. Generally, it's represented by a transition probability matrix where transition probabilities are previously given as prior knowledge. If it occurs, the solution developed will be under the category of *Model-Based* algorithms, otherwise *Model-Free* algorithms.

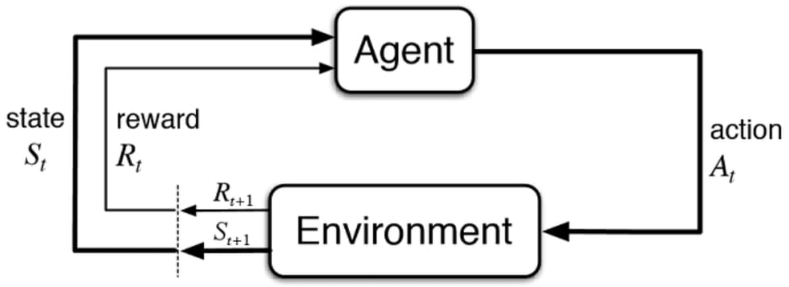


Figure 2.2: Agent-Environment interaction. When agent makes action a_t at state s_t, the environment is updated and returns a new state s_{t+1} and a reward r_{t+1}. Source: [4].

2.2.2 Markov Decision Process

Markov Decision Process (MDP), also called a stochastic dynamic program or stochastic control problem, is a model for sequential decision making when outcomes are uncertain[5]. Even though it was introduced for Operational Research in first instance, it has some properties which frame perfectly every Reinforcement Learning problem. On the one side, the objective of MDP is to find a good function π that allows the decision-maker to take an action a in state s, in the same way as formula (2.3). On the other side, every state s_{t+1} gets all information from the previous state s_t and doesn't recall the history of past states, since the previous one already embeds it. Thus, it holds

$$\mathbb{P}[s_{t+1} \mid s_1, ..., s_t] = \mathbb{P}[s_{t+1} \mid s_t] \tag{2.7}$$

A MDP and, for extension, Reinforcement Learning paradigm can be modeled as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ where:

- $\mathcal{S}$ is the set of all states or *state space*; it may be discrete or continuous.
- $\mathcal{A}$ is the set of actions or *action space*; it may be discrete or continuous.

- $\mathcal{P}$ is the transition probability from state s to state s' after action a

$$\mathcal{P}_a(s,s') = \mathbb{P}(s_{t+1} \in S' \mid s_t = s, a_t = a) = \int_{S'} \mathcal{P}_a(s,s')\, ds' \tag{2.8}$$

- $\mathcal{R}$ is the expected reward function

$$\mathcal{R}_a(s,s') = \mathbb{E}[r' \mid s_t = s, a_t = a] \tag{2.9}$$

- γ is the discounted reward factor $\gamma \in [0,1)$

As clarification, even though the $\mathcal{S}$ may be continuous, it's assumed that MDP moves states at discrete time, as a countable-infinite sequence, and, for the sake of the thesis, there will always be a *termination state* which will end the MDP.

2.2.3 Bellman Equations and Policy Optimization

Once the mathematical framework is set, the focus is shifted on the pursuit of goodness of actions a_i and policy π. Since the objective is to maximize the goodness of the policy, which directly influences action selection, *Bellman Equations* may be leveraged to reach optimality. In fact, an optimal policy has the property that whatever the initial state and decision are, future actions must constitute an optimal policy with regard to the state resulting from the first decision[6]. Thus, both (2.5) and (2.6) may be re-written as:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[g_t \mid s = s_t\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma r_{t+2} + ... \mid s = s_t\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma g_{t+1} \mid s = s_t\right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s)(\mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)[\gamma \mathbb{E}[g_{t+1} \mid s_{t+1} = s']]) \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s)(\mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)[\gamma V^\pi(s')]) \end{aligned} \tag{2.10}$$

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi\left[g_t \mid s = s_t, a = a_t\right] \\ &= \mathbb{E}_\pi\left[r_{t+1} + \gamma g_{t+1} \mid s = s_t, a = a_t\right] \\ &= \mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s)[\gamma \mathbb{E}[g_{t+1} \mid s_{t+1} = s', a_{t+1} = a']] \\ &= \mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s)[\gamma Q^\pi(s',a')] \end{aligned} \tag{2.11}$$

In order to reach optimality for policy π^*, one step more gives *Bellman Optimal Equations*:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{a}(\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a) V^{\pi}(s')) \\ Q^*(s,a) &= \max_{\pi} Q^{\pi}(s,a) \\ &= \mathcal{R}(s,a) + \max_{s'}(\gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s) Q^{\pi}(s',a')) \end{aligned} \tag{2.12}$$

And thus it's possible to rewrite Q^* in relation to V^* as:

$$Q^*(s,a) = \mathcal{R}(s,a) + V^*(s') \tag{2.13}$$

Explicitly solving Bellman Optimal Equations is one way to find the optimal policy. However, it's a method rarely adopted since the expected terms in both equations would imply, on the one side, an extremely long search in time perspective among all possible states and, on the other side, a huge cost in terms of computational power. Considering enough states $|\mathcal{S}| = m$ such that m^m transitions exponentially explode, the transition probability matrix alone would represent a non-negligible constraint for both memory allocation and matrix calculation. Alternatively, settling policy approximation may relax the problem complexity and achieve approximated optimal results comparable with the real optimal ones. As already explained in 2.1, function approximation makes possible to generalize RL problems, allowing to mimic the real behavior of policy function but introducing some drawbacks like non-stationarity, bootstrapping and delayed targets which normally don't appear in Supervised Learning[4].

Despite of the solution used, it's worthy to underline some distinction when approaching RL paradigm.
As already introduced in the Nomenclature section 2.2.1, a model of the environment may be provided and, more specifically, a transition probability matrix. This prior knowledge about the environment assumes a finite cardinality of states $|\mathcal{S}|$ and the existence of deterministic optimal policy $a = \pi(s)$. Algorithms using the transition probability matrix to find such policy are called *Model-Based* algorithms. Otherwise, they are called *Model-Free* algorithms. Since, this kind of prior knowledge is rarely given in real-world applications, Model-Based solution are difficult to implement.
The second discriminant factor is the objective function. Policy π may be seen as a mapping function which assigns probabilities to actions and the agent follows this probability distribution to behave in the best way. If the focus of the algorithm is to optimize directly the policy function, the solution is labeled as *Policy-based*. In contrast, if the target is instead the State Value Function V^{π}, the policy is

optimized only indirectly and the solution is labeled as *Value-based*.
The last parameter is when the learning process is carried out. The MDP framework defines a continuous chain between state-action exchange and the optimization of objective function takes it into account. If the learning process considers only the current sample or batch of state-action tuples, the algorithm is called *On-Policy*; otherwise, if it takes into account previous samples or batches analyzed by other policies, then it's labeled as *Off-Policy*.

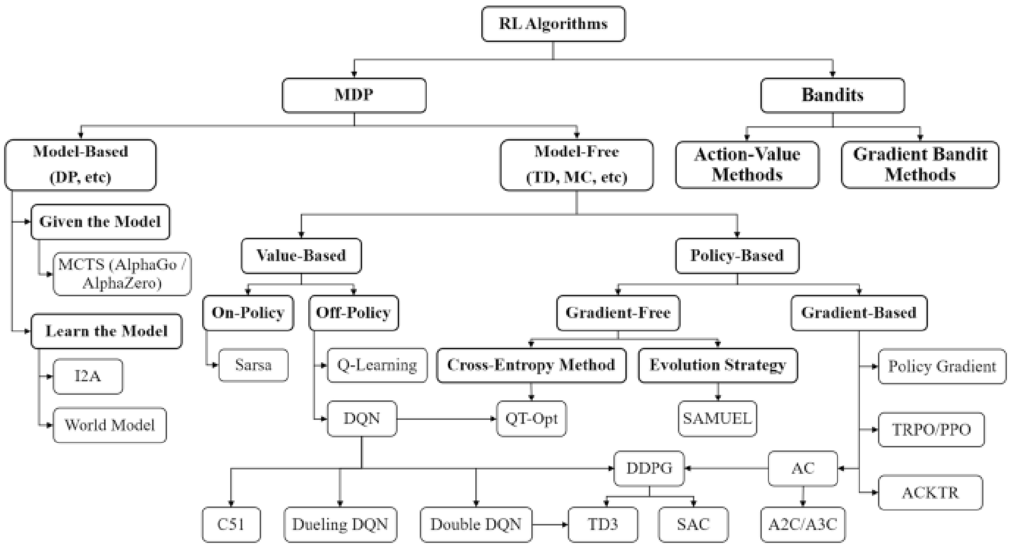


Figure 2.3: (Non-exhaustive) Taxonomy of algorithms in modern RL. Source: (.)

2.2.4 Dynamic Programming

Dynamic Programming (DP) is a method for solving problems by breaking them down into simpler sub-problems and solving each sub-problem only once, storing its solution for future use. This technique helps to avoid the redundant computation of overlapping sub-problems, leading to significant time and space efficiency. The intuition behind DP is that if sub-problems can be nested recursively inside larger problems, a relationship may be found between the values of the sub-problems themselves and the value of the larger problem; thus, if it's possible to obtain optimal solutions for sub-problems, consequently creating an optimal sub-structure, there is an optimal solution for the larger problem as well [6]. DP is largely implemented in Model-based situations, thanks to the effectiveness of computation but, as expected from more complex problems, it's rarely used when generalization and function approximation is needed.

In dynamic programming, several algorithms are leveraged. For instance, *Policy Iteration* is made of two steps. The first step is called *Policy Evaluation* or *Policy*

Prediction since the objective is to build the deterministic policy as at 2.3. The policy is initialized with some arbitral values and is updated through an iterative process until the accuracy is high enough and the error is below an arbitral parameter θ.
The second step is called *Policy Improvement* since initial policy may not be the optimal one and thus it's worth changing it. In order to discover if the current policy is the best fit, an action a is selected in state s such that $a \neq \pi(s)$. If $Q^{\pi'}(s,a)$ generated by the new policy π' is greater than $Q^{\pi}(s,a)$ according to old policy π, it's reasonable to assume that the new policy π' achieves betters results. In fact:

$$If \quad Q^{\pi}(s,\pi'(s)) \geq V^{\pi}(s) \quad Then \quad V^{\pi'}(s) \geq V^{\pi}(s) \tag{2.14}$$

Combining these two steps, a continuous cycle of evaluation and improvement is created and when the policy converges, a stable approximated policy and value function are obtained. A snippet of the algorithm is shown in Algorithm 1.

Algorithm 1 Policy Iteration for estimating $\pi \approx \pi^*$

```
1. Initialization
Input policy $\pi$ and set parameter $\theta > 0$ for accuracy threshold
Initialize $V(s)$ for $\forall s \in \mathcal{S}$, except $V(terminal) = 0$

2. Policy Evaluation
while $\Delta < \theta$ do:
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ do:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a \mid s)(\mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)[\gamma V^{\pi}(s')])$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
end while
TD learning is a combination of Monte Carlo ideas and dynamic programming
(DP) ideas.

3. Policy Improvement
$policyStable \leftarrow true$
for each $s \in \mathcal{S}$ do:
    $oldAction \leftarrow \pi(s)$
    $\pi(s) \leftarrow arg \max_a(\mathcal{R}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)[\gamma V^{\pi}(s')])$
    If $oldAction \neq \pi(s)$, then $policyStable \leftarrow false$
end for
If $policyStable$, then stop and return $V \approx V^*, \pi \approx \pi^*$; else, go to 2
```

Other dynamic programming methods may be used. For example, *Value Iteration* is focused on the improvement of value function V rather than policy π itself. It still preserves the cycle of evaluation and improvement of Policy Iteration fashion but the former phase updates the state value function, influencing indirectly the optimal policy.

Furthermore, the term *Generalized Policy Iteration* (GPI) refers to the general idea of allowing policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function[4]. This implies that the Bellman optimality equations 2.12 hold and consequently policy and value function are optimal. Almost all RL algorithms are classified as GPI.

2.2.5 Temporal-Difference learning

Temporal Difference (TD) learning is a class of Model-Free algorithms used to learn value functions directly from experience. It is a hybrid approach that combines elements from both Monte Carlo methods and Dynamic Programming (DP), offering the benefits of both while avoiding some of their limitations. In particular, these methods sample randomly from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods. One the one side, it doesn't require model of the environment, overcoming the main limitation of dynamic programming and generalizing over high state space; one the other side, it's well suited for continuous tasks and doesn't depend on a terminal state to update the policy, addressing more effectively MC lack of control over long episode series. In short, TD uses data collected from the environment and bootstraps to learn action value functions.

The general update rule for nonstationary environment is formalized as:

$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha[g_t - V(s_t)] \\ V(s_t) &\leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \end{aligned} \tag{2.15}$$

To clarify, g_t is the expected return at time t and α is a constant step-size parameter. In MC methods, which in this case are called *constant-α* MC, normally wait the terminal state before updating $V(s_t)$ and getting g_t. TD, instead, focuses on optimizing $r_{t+1} + \gamma V(s_{t+1})$ and the error derived from the difference with $V(s_t)$. Thus, a new objective function is introduced with TD error δ_t as target:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.16}$$

Two main algorithms are representative of TD learning: SARSA and Q-Learning.

State-Action-Reward-State-Action (SARSA) is a On-Policy TD algorithm which continually estimates Q^π for the policy π and at the same time change π toward greediness with respect to Q^π. If the previous transitions were made from state to state with the attempt of learning State-Value function, SARSA considers instead transitions from state-action to state-action with the attempt of learning Action-Value function, focusing on the error generated adjusted by the learning rate α.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.17}$$

Algorithm 2 SARSA (On-Policy TD control) for estimating $Q \approx Q^*$

```
Input $\alpha \in$ [0,1], small $\epsilon > 0$
Initialize $Q(s,a)$ for $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$ except $Q(terminal, \cdot) = 0$
for each episode do:
    Initialize $s_t$
    Choose $a_t$ from $s_t$ using policy $\pi$ derived from $Q^\pi$ (e.g., $\epsilon$-greedy)
    for each step episode do:
        Take action $a_t$, observe $r_{t+1}$ and $s_{t+1}$
        Choose $a_{t+1}$ from $s_{t+1}$ using policy $\pi$ derived from $Q^\pi$ (e.g., $\epsilon$-greedy)
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
        $s_t \leftarrow s_{t+1}$
        $a_t \leftarrow a_{t+1}$
        If $s_t$ is terminal, break the for-loop
    end for
end for
```

The ϵ-greedy policy is a simple method to balance exploration and exploitation of actions with a random parameter ϵ. Exploration allows to improve knowledge for long-term benefit, since it discovers new trajectories and brings different returns; exploitation uses current knowledge for short-term benefit, choosing the best action given a state, with the aim of obtaining the best possible return. An example of ϵ-greedy function is:

$$a_t = \begin{cases} arg\max_a Q^\pi(s_t, a_t) & with\ probability\ (1-\epsilon) \\ random(a_t) & with\ probability\ (\epsilon) \end{cases} \tag{2.18}$$

Q-learning[7] is a Off-Policy TD algorithm and it represents one of the early breakouts of RL world. This algorithm has influenced many following algorithms and still have a huge impact on Deep Learning RL, becoming a pillar for baseline solution development. It can also be viewed as a method of asynchronous dynamic programming. Similarly to SARSA, Q-learning updates an estimate of the optimal state-action value function Q^π based on a ϵ-greedy function. However, while SARSA updates the Q-values following the ϵ-greedy policy directly while Q-Learning updates the Q-values following a greedy policy which maximize action selection as in 2.18.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.19}$$

Algorithm 3 Q-Learning (Off-Policy TD control) for estimating $\pi \approx \pi^*$

```
Input $\alpha \in$ [0,1], small $\epsilon > 0$
Initialize $Q(s, a)$ for $\forall s \in \mathcal{S}$, $\forall a \in \mathcal{A}$ except $Q(terminal, \cdot) = 0$
for each episode do:
    Initialize $s_t$
    Choose $a_t$ from $s_t$ using policy $\pi$ derived from $Q^\pi$ (e.g., $\epsilon$-greedy)
    for each step episode do:
        Take action $a_t$, observe $r_{t+1}$ and $s_{t+1}$
        Choose $a_{t+1}$ from $s_{t+1}$ using policy $\pi$ derived from $Q^\pi$ (e.g., $\epsilon$-greedy)
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
        $s_t \leftarrow s_{t+1}$
        $a_t \leftarrow a_{t+1}$
        If $s_t$ is terminal, break the for-loop
    end for
end for
```

2.3 Deep Learning

Traditional Reinforcement Learning solutions seen in (2.2.4, 2.2.5) have become obsolete in the last decades due to lack of generalization when handling more complex problem, especially when addressing challenges from real world. When facing these new challenges and the relative approximation burden, it's taken for granted that the computational machinery necessary to express complex behaviors requires highly varying mathematical functions, i.e., mathematical functions that are highly non-linear in terms of raw sensory inputs, and display a very large number of variations across the domain of interest[8]. As matter of fact, the raw input of complex learning system is a high dimensional entity, composed by many observed variables, which are related by unknown intricate statistical relationships. In this regard, *Artificial Neural Networks* (ANN) have reached a huge consensus in scientific community as the best performing architecture for approximating an unknown function, or policy in RL, being able to capture those obscure variable relationships. Formally, Universal Approximation Theorem[9] states that:

Theorem 1 (Universal Approximation Theorem)
Given a family of neural networks, for each function f from a certain function space, there exists a sequence of neural networks $\phi_1, \phi_2, \ldots, \phi_n$ from the family, such that $\phi_{1\ldots n} \to f$ according to some criterion.

This section will give a brief idea of *Deep Learning*, a class of Machine Learning algorithms which leverage neural network architectures, organized as hierarchy of layers, to transform input data into a progressively more abstract and composite representation. This digression is forced in order to better understand the evolution of Reinforcement Learning, i.e. *Deep Reinforcement Learning*.

2.3.1 Feed-Forward Neural Network

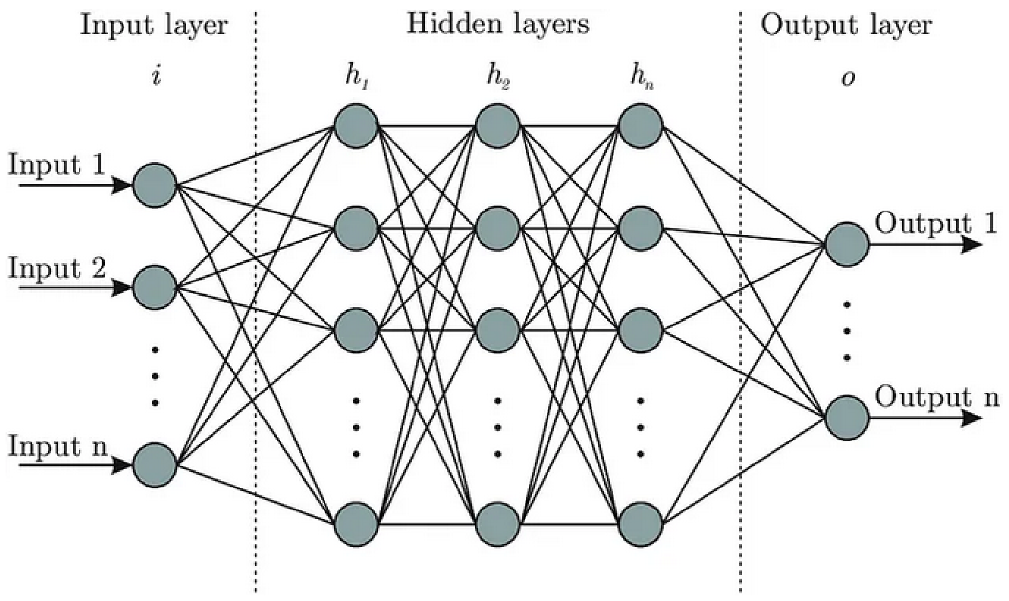


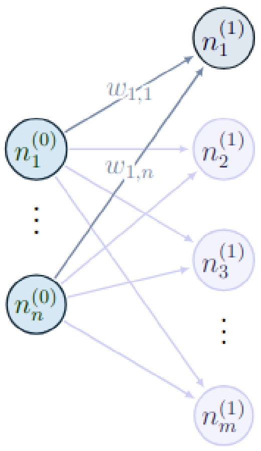
Figure 2.4: Representation of general Feed-Forward Neural Network. The input data enters from input layer i, then is manipulated by a series of hidden layers $h_1, \ldots, h_n$ and finally the network outputs the result from output layer o. Source: (.)

The first relevant contribution regarding neural network architecture shall be conducted to Frank Rosenblatt when he proposed the idea of *Multi-Layer Perceptron*[10]. Taking inspiration from brain structure, the network is a Direct Acyclic Graph (DAG) where layers of nodes or neurons are connected to each other with edges or weights and input data, after passing through this series of layers, is transformed into a shaped output. Obviously, the original idea has been outdated but the main concept still influence newer structures and a generalization may be seen in 2.4.

The main elements of Feed-Forward NN are:

- **Input Layer**: it takes the input data. The input data is normally shaped as a tensor, a multi-dimensional vector which embeds feature information, since it can concatenate multiple elements of the dataset.
- **Neuron**: it represents the NN-feature. Neurons receive and send information through continuous layers.
- **Weights and Biases**: they transform the information passed by the neurons. For instance, a neuron x_j of layer l_n will receive information equal to $x_j = \sum w_i x_i + b_j$, where w_i is the weight associated to neuron x_i of layer l_{n-1} and b_j is the bias.
- **Activation Function**: it gives the non-linear property to each layer. Layer output is transformed as $f = \sigma(w, x)$, where σ is the non-linear function. *ReLU, Tanh, Sigmoid* are all examples of activation functions.
- **Hidden Layers**: they stack neurons. Wide architectures contain layers with high number of neurons, while deep architectures contain high number of layers. Information aggregation of hidden layer neurons is explained in 2.5.
- **Output Layer**: it gives the final output of the network.

Designing neural network becomes fundamental to approximate efficiently the unknown function. When modeling *Fully Connected Layers*, i.e. layers where each neuron gives contribution to all neurons of next layer, the central question is to decide whether build a wide or deep network[11].
One the one side, deeper networks can capture a off-course range of patterns, aiding the displaying of complicated connections in data, and are inclined to create knowledge hierarchical likenesses when getting in input abstract features; however, they suffer vanishing or exploding gradient descent during the optimization process, since it's iterated through many layers, and are susceptible to overfitting if not designed properly, memorizing training set structure instead of finding a generalization rule. On the other side, wider networks can process multiple features in parallel, accelerating training and inference times, and are usually more robust to outliers or input perturbations, thanks to the great number of neurons; despite of this, the



$$= \sigma\left(w_{1,1}n_1^{(0)} + w_{1,2}n_2^{(0)} + \ldots + w_{1,n}n_n^{(0)} + b_1^{(0)}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}n_i^{(0)} + b_1^{(0)}\right)$$

$$\begin{pmatrix} n_1^{(1)} \\ n_2^{(1)} \\ \vdots \\ n_m^{(1)} \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \ldots & w_{m,n} \end{pmatrix} \begin{pmatrix} n_1^{(0)} \\ n_2^{(0)} \\ \vdots \\ n_n^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_m^{(0)} \end{pmatrix}\right]$$

$$\mathbf{n}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{n}^{(0)} + \mathbf{b}^{(0)}\right)$$

Figure 2.5: Mathematical representation of a neuron information aggregation. Each neuron of layer l_1 is the result of the multiplication between weight matrix and neurons from previous layer l_0, increased by a bias and transformed by activation function σ. Source: (.)

core drawback is the curse of dimensionality, with high memory and computing power consumption, and non-proportional benefit returns as the number of neurons increases.

2.3.2 Convolutional Neural Network

In 1989, Yann LeCun and its team proposed an interesting architecture designed to recognize handwritten zip code[12], *LeNet*. The perk of this innovative neural network was the sequential combination of convolutional layers which were able to extract different levels of feature from input data, thank to kernels or filters analysis. Moreover, the input data was passed as tensor in 2+ dimensions, scaling more easily the input memorization, instead of a huge 1-dimensional array, and thus facilitating feature extraction. Since this approach could directly make use of the spatial relationships, it outperformed MLP-based NN in tasks where spatial hierarchy and the local structure of data were important, such as image processing. In the following years, it gained popularity and countless updates of its structure, becoming the baseline algorithm in many fields, especially Computer Vision and Natural Language Processing.

A CNN presents additional elements than Feed-Forward NN:

- **Convolutional Layer**: the core building block of a CNN. It applies a set of

filters, or kernels, slides across the input image, performing a mathematical operation called convolution. This operation detects local patterns such as edges, textures, or more complex shapes. Each filter is designed to detect a specific feature. As the filter moves over the image, it generates a feature map that highlights areas where the feature is present; doing so, the same neuron can recognize learned features even if they appear in different locations of the image. In particular, the kernels are applied across the width and height of the input data and compute dot products between the values in the filter and the ones in the input at any position; after the convolution, activation function is applied. It's common practice to use $ReLU$ function to introduce non-linearity. By setting all negative values to zero, it helps the model learn complex patterns and not just linear relationships.
Given input data tensor of channels C_0, width W_0, height H_0 and convolutional parameters of stride S, padding P, kernel size K, dilation D, new channels C_1, the output tensor will have new dimensions as:

$$\begin{aligned} Channels_{out} &= C_1 \\ Width_{out} &= \lfloor \frac{W_0 + 2 \times P - D(K-1) - 1}{S} + 1 \rfloor \\ Height_{out} &= \lfloor \frac{H_0 + 2 \times P - D(K-1) - 1}{S} + 1 \rfloor \end{aligned} \tag{2.20}$$

- **Pooling Layer**: it decreases the spatial dimensionality of input tensor. It helps to reduce the spatial dimensions of the feature maps and thus computational load, contributing to control overfitting. The pooling operation typically works by selecting the aggregated value from a region of the feature map, making the output more abstract and less sensitive to small changes in the input image. According to the aggregation performed, the layer may be *MaxPooling* for max function or *AvgPooling* for averaging function.

- **Residual Block**: it's an optional element and it was introduced the first time in *ResNet*[13]. It creates a "*residual connection*" or "*skip connection*" in a sub-network with a certain number of stacked layers, allowing the model to skip one or more layers and enabling the flow of gradients directly through the network during training. This architecture stabilizes the training of extremely deep networks, i.e. hundreds of layers, without suffering from degradation in performance, as it facilitates convergence of optimization process.

2.3.3 Optimization

When input data has passed through all layers of neural network, an output is given. According to a **Loss Function** $\mathcal{L}(y, \hat{y})$, it's possible to evaluate the goodness of the

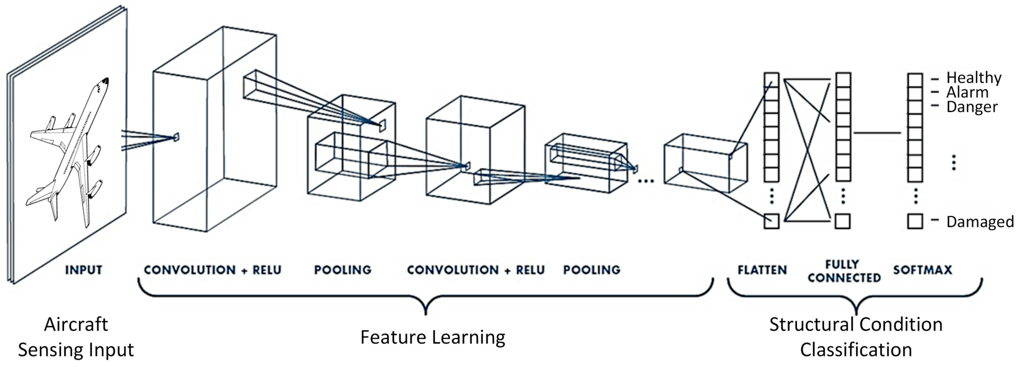


Figure 2.6: Representation of general Convolutional Neural Network for image classification. Convolutional layers extract features, depending on the number of kernels used. Pooling layers decrease width of tensors. Finally, Fully-Connected layers perform classification task, after all convolutional processing. Source: (.)

output $\hat{y} = f(x)$ and the accuracy of the approximated function f. In supervised learning, $\mathcal{L}$ will be an indicator of how far the predicted value $\hat{y}$ is from the expected label y. In reinforcement learning, there won't be a proper label to be evaluated but rather a comparison between predicted and expected $V(s)$ and $Q(s,a)$.
Countless loss functions exist in scientific literature but, for the sake of the thesis, only three are listed: *Mean Absolute Error* (MAE) or *L1 Loss*, *Mean Squared Error* (MSE) or *Squared L2 norm*, *Huber Loss.*

$$MAE = \frac{\sum_{i=1}^{n} |y_i - f(x_i)|}{n} \tag{2.21}$$

$$MSE = \frac{\sum_{i=1}^{n} (y_i - f(x_i))^2}{n} \tag{2.22}$$

$$HuberLoss = \begin{cases} \frac{1}{2}(y_i - f(x_i))^2 & if\ |y_i - f(x_i)| < \delta \\ \delta(|y_i - f(x_i)| - \frac{1}{2}\delta) & otherwise \end{cases} \tag{2.23}$$

As clarification, the loss formulation may be applied over all training set or progressively on batches of dataset; due to proportional increase of computational power and memory requirements, the latter choice is preferred.

Once the loss is obtained the network shall put in practice some kind of optimization, in order to minimize further errors. The parameters θ of a neural network, also called weights of the network, are the cornerstone of target function and their variations change the function behavior. It's now implicit that the better the configuration of such parameters, the better the accuracy and the lower the loss.

The most appreciated way of optimizing a neural network is through **Gradient Descent**[14]. The core idea of this technique is to compute the gradient of the loss function in relation to the model parameters and then change network parameters in the opposite direction of the gradient. Namely, the gradient, or slope, is a vector made up of the partial derivatives of the loss function with respect to each parameter. Given an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}$, the gradient is obtained as:

$$\nabla_\theta J(\theta) = (\frac{\partial J(\theta)}{\partial \theta_1}, \ldots, \frac{\partial J(\theta)}{\partial \theta_n})^T \tag{2.24}$$

When applying gradient descent on huge training sets, a trade-off between the accuracy of the parameter update and the time it takes to perform an update, depending on how much data we use to compute the gradient of the objective function. In case all dataset is taking into account it will be called *Batch Gradient Descent* (BGD), otherwise, if the update is performed on each training sample (x_i, y_i), it will be called *Stochastic Gradient Descent* (SGD).

$$\theta = \begin{cases} \theta - \eta \cdot \nabla_\theta J(\theta) & if\ BGD \\ \theta - \eta \cdot \nabla_\theta J(\theta; x_i; y_i) & if\ SGD \end{cases} \tag{2.25}$$

Another common baseline for optimization of NN is *Adam*[15]. It's an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. It leverages on the momentum m for accelerating the gradient descent process by incorporating an exponentially weighted moving average of past gradients. This helps smoothing out the trajectory of the optimization, reducing convergence oscillations.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\frac{\partial J(\theta)}{\partial \theta_t} \tag{2.26}$$

Moreover, it adds an exponentially weighted moving average of squared gradients, which helps overcome the problem of diminishing learning rates.

$$\theta_{t+1} = \theta_t - \frac{\alpha_t}{\sqrt{v_t + \epsilon}} \frac{\partial J(\theta)}{\partial \theta_t} \tag{2.27}$$

It's generally preferred to previous gradient descent methods since: with dynamic learning rates, it avoids oscillations and gets past local minima efficiently; it helps preventing early-stage instability thanks to bias correction; it has more efficient performances with hyperparameter tuning than the static one of SGD.

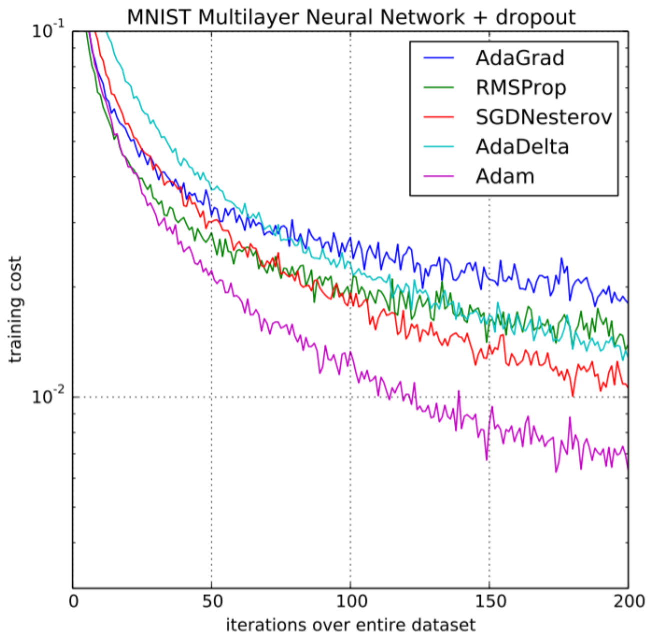


Figure 2.7: Performance comparison of different optimizers on training cost. Source: [15]

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning is the direct evolution of RL, which is bolstered by the expressive capabilities of deep neural networks. As already explained in previous sections, traditional RL algorithms lack generalization and deployability when facing complex challenges or real-world scenarios, addressing several issues like rapidly-changing dynamics, intractable and computationally expensive state and action spaces, and noisy reward signals[16]. These traditional methods are also called "*tabular methods*" because state-value and action-value functions are represented with large tables, where each entry corresponded to the value of a particular state or state-action couple. When the joint state-action space becomes too high to be stored in feasible memory, the implementation of neural networks as approximation functions gives an alternative approach to the solution development. Considering a $V^\pi(s)$ and $Q^\pi(s,a)$ as functions to be approximated and θ as the vector of neural network parameters, the focus of DRL becomes to find the proper optimal approximation such as:

$$\begin{gathered} V(s;\theta) \approx V^\pi(s) \\ Q(s,a;\theta) \approx Q^\pi(s,a) \end{gathered} \tag{2.28}$$

Practically, all DRL algorithms are under the Model-Free label, since the focus is generalization, and, following the taxonomy at 2.3, two main branches characterize the plethora of possible solutions: Value-based one which is translated in *Deep Q-Learning* and Policy-based one which is generally extended into *Policy Gradient.*

2.4.1 Deep Q-Learning

Recalling Algorithm 3, the aim of Q-Learning is seeking for the most-suited Q-function $Q^{\pi}(s,a)$ and store the action-values into a Q-table. Deep Q-Learning keeps the same objective with a Q-function $Q(s,a;\theta)$ and maintains Off-policy property, meaning that policy π^* is optimized taking into account not only the current samples of training set but also previous batches from different policies. Since there isn't anymore a Q-Table to store all state-action values, an experience replay buffer $\mathcal{D}$ is introduced; thus, when training the network, random samples from the replay memory are used instead of the most recent trajectory. Moreover, the trade-off between exploration and exploitation is similarly handled by ϵ-greedy policy.
The most promising algorithm of this category is *Deep Q-Network* (DQN).

2.4.2 Policy Gradient

Value-based algorithms learn a parameterized value-function and the agent follows a policy that is directly derived from this value-function. However, it can be desirable to directly learn a policy as a separate parameterized function, eventually represented by any function approximation technique. If the approximation is performed leveraging on gradient updates, the algorithm is labeled as Policy Gradient. More specifically, if given a policy π_θ with parameters θ and $J(\pi_\theta)$ is the expected finite-horizon discounted return of the policy, the gradient $J(\pi_\theta)$ computed will be:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{N} \nabla_\theta \log \pi_\theta(s_t,a_t;\theta) Q_\pi(s,a;\theta)\right] \tag{2.29}$$

And, if the direction of the gradient improves the expected return, the weight parameters are updated through *Gradient Ascent* and not *Gradient Descent*, differently from deep learning optimization techniques which consider loss backpropagation instead:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.30}$$

Directly representing the policy of an RL agent has two key advantages[4].
First, in environments with discrete actions, a parameterized policy can represent any probabilistic policy, which leads to significantly more flexibility in its action

selection compared to value-based RL algorithms. A value-based RL agent following an ϵ-greedy policy is more restricted in its policy representation depending on the current value of ϵ and its greedy action. Second, by representing a policy as a separate learnable function, we can represent policies for continuous action spaces. In environments with continuous actions, an agent selects a single or several continuous values (typically within a certain interval) as its actions.
Among Policy Gradient solutions, it's worth to underline *Actor-Critic* methods. It's a family of policy gradient algorithms that train a parameterized policy, called the *Actor*, and a value function, called the *Critic*, alongside each other. The actor makes choices by selecting actions according to the current policy. Its role is to explore the action space in order to maximize the expected cumulative rewards. By continuously refining the policy, the actor adjusts to the changing environment. The critic assesses the actions performed by the actor. It estimates the value or quality of these actions by offering feedback on their effectiveness. The critic plays a crucial role in steering the actor towards actions that yield higher expected returns, thus enhancing the overall learning process.

2.4.3 Popular algorithms

DQN[17] was presented in 2013 as an algorithm able to play several Atari 2600 games from the Arcade Learning Environment. It successfully learned to control policies from high-dimensional sensory input, taking raw pixels from the games and outputting the value-function to estimate future rewards, and it has even surpassed human expert in three of those games.
The main innovation is the implementation of two similar neural networks, *Behavior network* $Q(s, a; \theta)$ and *Target network* $Q^*(s, a; \bar{\theta})$. When computing the loss $\mathcal{L}(\theta)$, the expected next Q-value for s_{t+1} follows the target network while the predicted next Q-value is according the current behavior network. In this way, there will be convergence as soon as the two networks will output the same Q-values. Since it's Off-Policy method, the behavior network uses random samples from the buffer $\mathcal{D}$. The update of parameters is done as:

$$\theta_{k+1} \leftarrow \theta_k + \alpha(r_{t+1} + \gamma \max_a Q^*(s_{t+1}, a_{t+1}; \bar{\theta}_{k+1}) - Q(s_t, a_t; \theta_k))\nabla_{\theta_k} J(\theta_k) \quad (2.31)$$

In order to prevent oscillation in learning process, only the behavior network is updated continuously while the target network is updated every C episodes or softly according to small values of τ.

A2C[18] stands for *Advantage Actor Critic* and it's a On-policy, synchronous, deterministic actor-critic algorithm that updates the policy and value function simultaneously. On the one side, the neural network function of the actor outputs a deterministic policy, $\pi(s, a; \phi)$, which maps states to actions. On the other side, the

Algorithm 4 Deep Q-Network (DQN)

```
Initialize behavior network $Q$ with random parameters $\theta$
Initialize target network $Q^*$ with parameters $\bar{\theta} = \theta$
Initialize an empty replay buffer $\mathcal{D}$ to capacity $N$

**for** each episode **do**:
    Initialize sequence with state $s_1$
    **for** each step episode **do**:
        With probability $\epsilon$, choose random action $a_t$
        otherwise, with probability $(1 - \epsilon)$, $a_t = \max_a Q^*(s_t, a; \bar{\theta})$
        Apply action $a_t$; observe reward $r_{t+1}$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ into buffer $\mathcal{D}$
        Sample random mini-batch of $B$ transitions $(s_k, a_k, r_{k+1}, s_{k+1})$ from $\mathcal{D}$
        Set $y_k = \begin{cases} r_{k+1} & \textit{if terminal } s_{k+1} \\ r_{k+1} + \gamma \max_a Q^*(s_{t+1}, a_{t+1}; \bar{\theta}) & \textit{if non terminal } s_{k+1} \end{cases}$
        Compute $\mathcal{L}(\theta) = y_k - Q(s_t, a; \theta)$
        Update parameters $\theta$ by minimizing the loss $\mathcal{L}(\theta)$
    **end for**
    Every $C$ episodes, update target network parameters $\bar{\theta}$
    Alternatively, every episode, softly $\tau$-update target network parameters $\bar{\theta}$
**end for**
```

neural network of the critic approximates the value function, $V(s;\theta)$ and estimates the prediction of expected cumulative reward from a given state. As the name suggests, given the Q-value $Q(s,a;\phi)$ of an action chosen by the actor and the state-value $V(s;\theta)$ estimated by the critic, the *Advantage* is obtained as:

$$Adv(s,a;\phi;\theta) = Q(s,a;\phi) - V(s;\theta) \tag{2.32}$$

The advantage can therefore be understood as quantifying how much higher the expected returns are when applying the specific action a compared to following the actor policy π in state s. The advantage will be positive or negative if, respectively, it has been reached a higher or lower return in comparison to the expected return of the state-value. The innovation of A2C regards, in fact, the optimization of actor policy which is not performed considering actor network alone but combining the critic network as well, unified by the advantage function.

Algorithm 5 Advantage Actor Critic (A2C)

```
Initialize actor network function $\pi$ with random parameters $\phi$
Initialize critic network $V$ with parameters $\theta$

for each episode do:
    Initialize sequence with state $s_1$
    for each step episode do:
        Sample action $a_t \sim \pi(\cdot|s_t;\phi)$
        Apply action $a_t$; observe reward $r_{t+1}$ and new state $s_{t+1}$
        if $s_{t+1}$ is terminal then:
            Advantage $Adv(s_t,a_t) \leftarrow r_{t+1} - V(s_t;\theta)$
            Critic target $y_t \leftarrow r_{t+1}$
        else:
            Advantage $Adv(s_t,a_t) \leftarrow r_{t+1} + \gamma V(s_{t+1};\theta) - V(s_t;\theta)$
            Critic target $y_t \leftarrow r_{t+1} + \gamma V(s_{t+1};\theta)$
        end if
        Compute actor loss $\mathcal{L}(\phi) = -Adv(s_t,a_t)\log\pi(a_t|s_t;\phi)$
        Compute critic loss $\mathcal{L}(\theta) = y_t - V(s_t;\theta)$
        Update parameters $\phi$ by minimizing the actor loss $\mathcal{L}(\phi)$
        Update parameters $\theta$ by minimizing the critic loss $\mathcal{L}(\theta)$
    end for
end for
```

PPO[19] is a On-Policy algorithm, built upon the concept of *Trust Policy Region*[20], but with key innovations that make it more computationally efficient.

Policy updates are normally computed using gradient directly and, depending on the learning rate, it might lead to significant changes of the policy and could reduce the expected performance of the policy. The risk of large changes in the policy resulting from a single gradient optimization step can be minimized by using trust regions. Essentially, trust regions define a boundary in the policy parameter space where changes to the policy are limited, ensuring that the resulting policy will not cause a significant drop in performance. This allows to "trust" that the policy, with such parameter adjustments, will maintain its effectiveness.
Moreover, PPO introduces the importance sampling weight $\rho(s_t, a_t)$ and the clipping technique, in order to restrict how much the policy can change during each update, ensuring that the policy does not diverge too far from the previous policy.

Algorithm 6 Proximal Policy Optimization (PPO)

```
Initialize actor network function $\pi$ with random parameters $\phi$
Initialize critic network $V$ with parameters $\theta$

for each episode do:
    Initialize sequence with state $s_1$
    for each step episode do:
        Sample action $a_t \sim \pi(\cdot|s_t;\phi)$
        Apply action $a_t$; observe reward $r_{t+1}$ and new state $s_{t+1}$
        $\pi_\beta(a_t|s_t) \leftarrow \pi(a_t|s_t;\phi)$
        for epoch $e = 1, \dots, N_e$ do:
            $\rho(s_t, a_t) \leftarrow \frac{\pi(a_t|s_t;\phi)}{\pi_\beta(a_t|s_t)}$
            if $s_{t+1}$ is terminal then:
                Advantage $Adv(s_t, a_t) \leftarrow r_{t+1} - V(s_t;\theta)$
                Critic target $y_t \leftarrow r_{t+1}$
            else:
                Advantage $Adv(s_t, a_t) \leftarrow r_{t+1} + \gamma V(s_{t+1};\theta) - V(s_t;\theta)$
                Critic target $y_t \leftarrow r_{t+1} + \gamma V(s_{t+1};\theta)$
            end if
            Actor loss $\mathcal{L}(\phi) = -\min\begin{pmatrix} \rho(s_t, a_t)Adv(s_t, a_t) \\ \text{clip}(\rho(s_t, a_t), 1-\epsilon, 1+\epsilon)Adv(s_t, a_t) \end{pmatrix}$
            Critic loss $\mathcal{L}(\theta) = y_t - V(s_t;\theta)$
            Update parameters $\phi$ by minimizing the actor loss $\mathcal{L}(\phi)$
            Update parameters $\theta$ by minimizing the critic loss $\mathcal{L}(\theta)$
        end for
    end for
end for
```

2.5 Multi-Agent Reinforcement Learning

This final section describes the main topic of the thesis: *Multi-Agent Reinforcement Learning*. Intuitively, it represents the sub-field of RL that studies the interaction of multiple agents in the same environment. It addresses far more issues than a single-agent situation, facing complex dynamics due to agent behaviors. In fact, agents may share the same goal, for instance like autonomous robots cooperating to map disaster sites and locate survivors, or conflict goals, such as agents trading goods in a virtual market in which each agent seeks to maximize its own gains. This view opens many new challenges regarding the optimization of the policy because the dichotomy between egoistic rewards and cooperative rewards leads to new formulations of the solutions.
Many of the following sub-sections take inspiration from [21] for descriptive paragraphs, since it's one of the lately updated sources on the topic.

2.5.1 Nomenclature in Multi-Agent systems

Multi-Agent Reinforcement Learning (MARL) algorithms focus on learning optimal policies for a set of agents in a multi-agent system and, as in the single-agent counterpart, the policies are learned through a continuous exchange of trial-and-error to maximize the agents' cumulative rewards. However, despite of the proven modeling through MDPs in single-agent stochastic environments, MARL environments require a different representation: in fact, state dynamics and expected reward changes violate the core stationarity assumption of an MDP. Recalling the definition of *Stationarity*[22]:

Theorem 2 (Stationarity Theorem)
Formally, let $\{X_t\}$ be a stochastic process and let $F_X(x_{t_1+\tau}, \ldots, x_{t_n+\tau})$ represent the cumulative distribution function of the unconditional joint distribution of $\{X_t\}$ at times $t_1 + \tau, \ldots, t_n + \tau$. Then, $\{X_t\}$ is said to be strictly stationary, strongly stationary or strict-sense stationary if:

$$F_X(x_{t_1+\tau}, \ldots, x_{t_n+\tau}) = F_X(x_{t_1}, \ldots, x_{t_n}) \quad for\ all\ \tau, t_1, \ldots, t_n \in \mathbb{R} \tag{2.33}$$

Since τ does not affect $F_X(\cdot)$, $F_X(\cdot)$ is independent of time.

In this sense, the cumulative reward and behavior of the agents should have mean and variance constant in time but, since they are not independent to each other, the stationarity assumption doesn't hold; in addition, in real-world scenarios, the environment noise can't be assumed as a random variable with fixed mean-variance, contributing to the rejection of stationarity hypothesis.

Returning to MARL specifics, the combination of agents' actions $\{a_i\}$ at time t in state s_t is called *Joint Action* and it modifies the state according to environment dynamics. Consequently, the environment is updated and new observations and rewards are given to each agent; both observation and reward are shaped depending on the single agent's action. Figure 2.8 gives an idea of this interaction. Keeping

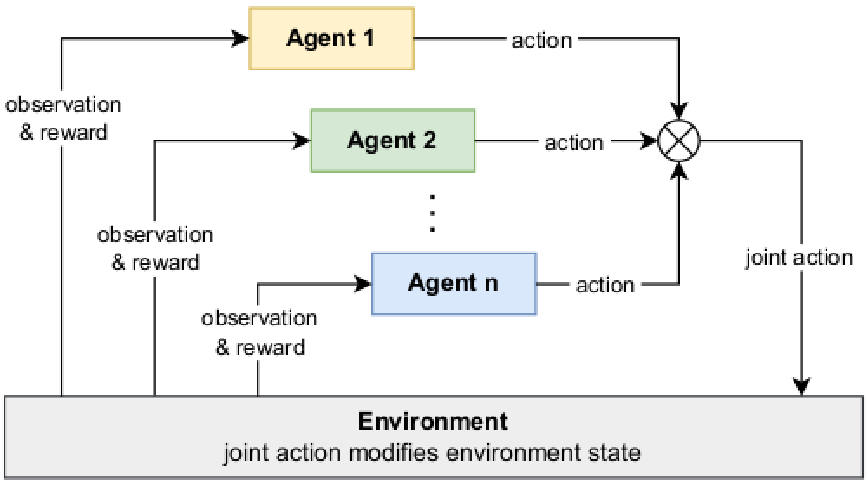


Figure 2.8: Agents-Environment interaction. When the joint action of all agents modifies the environment, it returns different observations and rewards to agents, directly influenced by the single agent's action. Source: [21]

the tuple-based structure of MDP, a MARL system with $\mathcal{N}$ agents can be expressed as:

$$< \mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_N, \mathcal{P}, \{\mathcal{R}^i\}_N, \gamma > \tag{2.34}$$

where the new terms are is the joint action space $\{\mathcal{A}^i\}_N$ and the set of reward functions for each agent $\{\mathcal{R}^i\}_N$.

According to agents' dynamics, Multi-Agent systems may labeled in three categories.

- **Cooperative**: if agents learn to collaborate in order to achieve a shared goal while maximizing a collective reward. Each agent's actions contribute to the overall success of the group, with the reward structure designed to encourage teamwork.

- **Competitive**: if agents participate in adversarial or competitive-cooperative scenarios where they aim to maximize their own rewards while minimizing those of their opponents, typical of zero-sum games.

- **Mixed-Interest**: if agents navigate cooperative-competitive dynamics, where they have both aligned and conflicting goals.

Finally, the last criteria to distinguish Multi-Agent systems is *Observability*. Taking for granted that *observation* $\neq$ *state*, the agents can either know other agents' actions/rewards or not. In case they don't have this knowledge, they can have some form communication to share information. A summary of all these possibilities is shown in Figure 2.9.

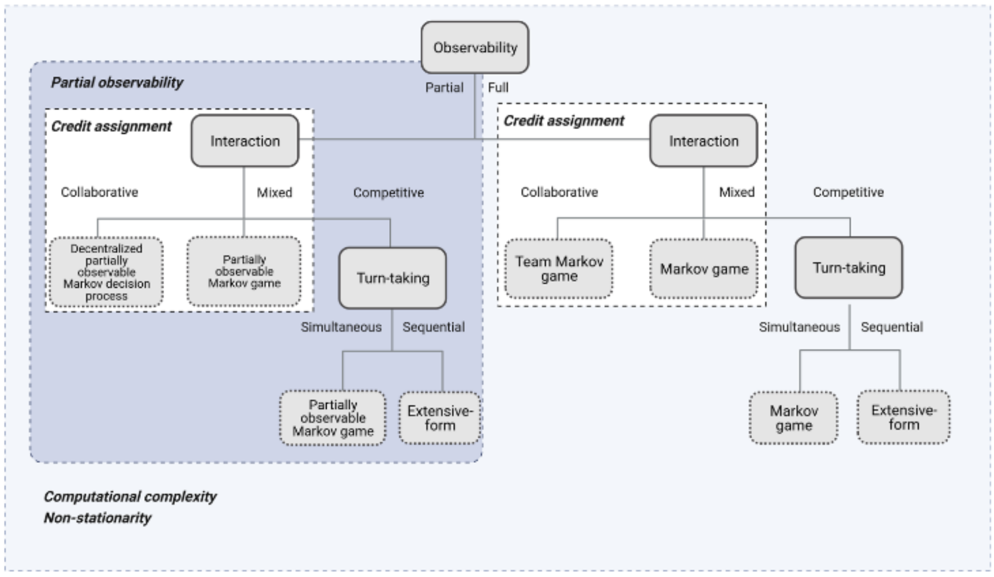


Figure 2.9: Classification of MARL systems, based on the environments constraints and agents' dynamics. Source: (.)

2.5.2 Open challenges

MARL comes with numerous challenges and addressing them properly is a prerequisite for the development of effective learning approaches. Despite promising results in the literature, computational complexity, non-stationarity, partial observability and credit assignment remain largely unsolved[23]. What makes even more struggling is that these obstacles do not occur in isolation but, in contrast, the solution development usually deals with one or more at the same time. In fact, the optimal policy convergence or simply general policy convergence is not guaranteed for every combination of algorithm-environment.

Computational Complexity is a term referring to computation power, time and memory requirements required to collect enough sampling data, learn a parameterized policy function and optimize it to approximate the optimal target function. The data collection is the first tackling point of RL paradigm because there isn't a pre-built dataset of inputs and labels, like in Supervised Learning. The exploration-exploitation framework is instead the shared rule of RL and the only way to collect as much state-action-state transitions as possible, limiting the sample efficiency.
Then, policy formulation requires long training epochs to converge, due to parameters optimization and sample generation. Once the policy is updated and probability distribution changed, new samples have to be generated by the updated policy. Therefore, the optimization faces a non-static context, where achieving the best objective value on the current samples is not the goal, and exploration is necessary to find better samples[24]. On complex or continuous-space problems, slow learning may become, in the worst-case, even unfeasible to master; the worst case is also more likely to happen if multiple agents interact in the same system.

Non-Stationarity refers to the environment evolution whose statistical properties are changing through time. This property may have different causes, such as stationary distribution of environment specifics that the agents cannot influence nor cannot observe, non-stationary settings that are influenced directly by the agent's behavior or a combination of both[25].
TD error δ_t in 2.16 set a good starting point in policy formulation, considering non-stationarity as a *moving target problem*, where state-values depend on subsequent actions which can change as the policy π changes over time and the error propagates in continuous time steps[21]. However, in MARL, multiple agents learn from the interaction to each other, resulting in a continual co-adaptation and re-formulation of policy function, leading to cyclic dynamics where a learning agent tailors his behavior regarding other agents' conducts which, in turn, will do the same with respect to the learning agent itself. Thus, it's implicit that statistical properties of the system, like mean and variance of different policies, are not constant in time since they are influenced by both exogenous and endogenous factors.

Partial Observability means that agents must take actions based on local observations, since they are not able to access the global state, creating a system where agents have incomplete and asymmetric information. Similarly, a learning agent may not see other agents' actions and rewards, making more complicated to understand how much its own action has effectively influenced the environment or, in contrast, it was mainly due to external changes.
Communication is a well promising mechanism to coordinate the dynamics of several agents, broadening their views of the environment, and supporting their collaborations[26]. The type of message sent can assume various forms, propagated

to all agents or only to sub-groups. However, there is no unique solution to solve partial observability and, in many cases, it should be tailored specifically for the environment.

Credit Assignment is probably the most challenging problem of the whole RL world. On the one side, there is the *Temporal Credit Assignment*, based on the long term consequences of single agent's actions, evaluating if distant and more beneficial dynamics are more important than greedy immediate behaviors; on the other side, there is the *Multi-Agent Credit Assignment*, questioning which agent is impacting the most to the shared welfare and how to proportionally reward the remaining agents.
The former credit assignment is not standard but tailored to the environment and agents which is applied to. Taking into account the action-value function as the measure of qualitative influence of the policy improvement is surely one of the main drivers of credit formulation[27]. Low quality evaluations can lead the policy to diverge from the optimal one, slowing down the progress[4]; while, high quality evaluations supply robust, precise and trustworthy signals that speed up convergence. Giving a-priori formulation is nonetheless misleading, especially in real world scenarios where a complex blend of several hierarchical tasks assign different priorities and different measures of optimality.
The latter credit assignment needs a proper evaluation of the Joint Action space $\{A^i\}_N$; in particular, the reward can be shaped on the direct or indirect assessment of agents' contributions. *Explicit methods* provide techniques to assign agent contributions that are at least provably locally optima[28]. *Implicit methods*, instead, do not intentionally attribute the individual actions against a certain baseline, and prefer learning a value decomposition from the shared reward signal into the individual component value functions[28].

2.5.3 Training and execution mode

One last point needs to be clarified: whether the agents are conscious of the other allies/opponents or not and, as well, whether this knowledge is shared or not. During training phase, agents may be restricted to leverage on its own local information only, in case of decentralized approach, or might access the global information like a continuous exchange of collection and delivery of respective data.
The execution aspect contributes to define the type of MARL algorithm since the joint action space may be interpreted as a joint set made of multiple sub-action spaces, in case of decentralized approach, or as a big action space composed by every possible combination of agents' actions, in case of centralized approach.

Centralized Training and Centralized Execution (CTCE) is the first set-up for solution development and arguably the easiest to implement. Having a central structure, both in learning process and execution of actions, can be interpreted as main "super-agent" which controls all the aspects of multiple sub-agents. Centrally shared information may include the agents' local observation histories, learned world and agent models, value functions, or even the agents' policies themselves[21]. In this way, it's possible to reduce a MARL game to a RL problem by using the joint-observation history to train a single central policy over the joint-action space. This is highly beneficial in games with partial observability, creating a more complete observation of the state by combining the joint-observations and reducing the vector of actions as a single action formed by the combination of agents' actions. Intuitively, this approach cannot be applied if action space is continuous or, in discrete case, grows exponentially with the number of agents; for example, having $n = 5$ agents with action space $|A| = 10$, the number of possible combination reaches $|\{A_n\}| = 10^5$, making it unfeasible even for the most powerful neural networks. Moreover, the "super-agent" shall shape a unique reward function for all agents and, if the number grows quickly, it becomes unfeasible to assign merit to a specific agent's behavior.

Decentralized Training and Decentralized Execution (DTDE) is the complete opposite of CTCE. The main idea, from the single agent perspective, is to consider other agents as a non-stationary part of the environment dynamics, so that the single agent behaves independently and trains its own policy in a completely local way, like in normal RL game[21]. The main benefit is the scalability of this solution, since the number of agents doesn't affect the local computation and local optimization. However, due to the lack of shared global knowledge, training can be heavily afflicted by non-stationarity caused by the concurrent training of all agents. Furthermore, it's unlikely that agents can distinguish stochastic changes in the environment as a consequence of other agents' actions and the natural evolution of environment transitions. Thereby, the global training will suffer of instable learning and poor convergence of each policy.

Central Training and Decentralized Execution (CTDE) is the middle ground of previous approaches and it tries to combine the benefits of both solutions, bypassing the respective critical drawbacks. It's the preferred solution in MARL problems since it enables conditioning approximate value functions on privileged information in a computationally tractable manner[21]. A popular application is within the Actor-Critic paradigm, where a multiple independent actors provide joint-observations to the single central critic which leverages on them to learn the advantage function.

2.5.4 Possible solutions

Independent Deep Q-Network (IDQN)[29] is a DTDE solution where agents train their own Q-value function and use their own memory buffer without sharing knowledge. It keeps the same structure of the classical DQN but the buffer storage should be handled with more attention, weighting older experience with less importance; in fact, it might be useful to apply importance sampling to address the changing policies of other agents and, thereby, correct the non-stationarity of the data distribution [30].
A snippet of the algorithm can be seen at 7.

Algorithm 7 Independent Deep Q-Network (IDQN)

```
Initialize $n$ behavior network $Q$ with random parameters $\theta_1, \dots, \theta_n$
Initialize $n$ target network $Q^*$ with parameters $\bar{\theta}_{1\dots n} = \theta_{1\dots n}$
Initialize an empty replay buffer for each agent $\mathcal{D}_1, \dots, \mathcal{D}_n$ to capacity $M$

for each episode do:
    Initialize sequence with state $s_1$
    for each step episode do:
        for agent $i = 1, \dots, N$ do:
            With probability $\epsilon$, choose random action $a_t^i$
            otherwise, with probability $(1-\epsilon)$, $a_t^i = \max\limits_{a^i} Q_i^*(s_t, a^i; \bar{\theta}_i)$
        end for
        Apply actions $a_t^i$; observe reward $r_{t+1}^i$ and new state $s_{t+1}^i$
        Store transition $(s_t, a_t^i, r_{t+1}^i, s_{t+1})$ into buffer $\mathcal{D}_1, \dots, \mathcal{D}_N$
        for agent $i = 1, \dots, N$ do:
            Sample random batches of transitions $(s_k, a_k^i, r_{k+1}^i, s_{k+1})$ from $\mathcal{D}_i$
            Set $y_k^i = \begin{cases} r_{k+1}^i & if\ terminal\ s_{k+1} \\ r_{k+1}^i + \gamma \max\limits_{a^i} Q_i^*(s_{t+1}, a_{t+1}^i; \bar{\theta}_i) & if\ non\ terminal\ s_{k+1} \end{cases}$
            Compute $\mathcal{L}(\theta_i) = y_k^i - Q_i(s_t, a^i; \theta_i)$
            Update parameters $\theta_i$ by minimizing the loss $\mathcal{L}(\theta_i)$
        end for
    end for
    for agent $i = 1, \dots, N$ do:
        Every $C$ episodes, update target network parameters $\bar{\theta}_i$
        Alternatively, softly $\tau$-update target network parameters $\bar{\theta}_i$
    end for
end for
```

Multi-Agent Deep Deterministic Policy Gradient (MADDPG)[31] is a CTDE solution and extends the idea of Actor-Critic paradigm to MARL systems. The upgrade consists in training one centralized critic over n decentralized agents. Each agent has its own continuous deterministic policy, accessing to local information only, while the critic learns the advantage function leveraging on a common memory buffer which stores state-action transitions from all actors experiences.
In the original paper, MADDPG is evaluated on cooperative communication scenario and the contribution of centralized critic towards the actors has achieved a good convergence rate balanced with convergence speed. However, in competitive settings, non-stationarity still represents a tackling issue and actors normally overfit the other actors' behaviors. In order to overcome this problem, the paper has proposed training a collection of K sub-policies for each actor. At the beginning of each episode, the actors choose from the pool of sub-policies and send state-action transitions to the memory replay buffer associated to the sub-policy previously selected.

Homogeneous Agents Solutions[21] is the last approach to be described and it's not a specific algorithm but more a "relaxation" of MARL problem. Since the number of agents directly affects the scalability of the algorithm, the sample efficiency and the training speed, the nature of the agents should be questioned if this makes a point. Homogeneous agents can be thought as identical or nearly identical copies of each other, and they interact with the environment in similar ways. Consequently, if they share similar characteristics, it would be allowed to interchange freely their own policies or, in the best case, share one single policy valid for all of them.

Theorem 3 (Weakly homogeneous agents)
A multi-agent environment has weakly homogeneous agents if for any joint policy $\pi = (\pi_1, \ldots, \pi_n)$ and permutation between agents $\sigma : I \to I$, the following holds:

$$U_i(\pi) = U_{\sigma(i)}(< \pi_{\sigma(1)}, \ldots, \pi_{\sigma(n)} >), \quad \forall i \in I \tag{2.37}$$

Theorem 4 (Strongly homogeneous agents)
A multi-agent environment has strongly homogeneous agents if the optimal joint policy consists of identical individual policies, formally:

- *The environment has weakly homogeneous agents.*
- *The optimal joint policy $\pi^* = (\pi_1, \ldots, \pi_n)$ consists of identical policies $\pi_1 = \cdots = \pi_n$.*

In case of environments with strongly homogeneous agents, the optimal joint policy can be represented as a joint policy consisting of identical policies for all agents[21]. Any trajectory performed by those agents, in the form of state-action transitions, is

Algorithm 8 Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

```
Initialize $n$ actor networks $\pi$ with random parameters $\theta_1, \dots, \theta_n$
Initialize critic network $V$ with parameters $\bar{\theta}$
Initialize an empty replay buffer $\mathcal{D}$ to capacity $M$

for each episode do:
    Initialize sequence with state $s_1$
    for each step episode do:
        For each agent $i$, $a_i = \pi_{\theta_i}(s_t)$ w.r.t. the current policy and exploration
        Apply actions $a = \{a_1, \dots, a_n\}$; observe reward $r^i_{t+1}$ and state $s_{t+1}$
        Store transition $(s_t, a^i_t, r^i_{t+1}, s_{t+1})$ into buffer $\mathcal{D}$
        $s_t \leftarrow s_{t+1}$
        for agent $i = 1, \dots, n$ do:
            Sample random batches of transitions $(s_k, a^i_k, r^i_{k+1}, s_{k+1})$ from $\mathcal{D}_i$
            Set $y_t \leftarrow r^i_{t+1} + \gamma V(s_{t+1}; \bar{\theta})$
            Update critic minimizing loss $\mathcal{L}(\bar{\theta}) = y_t - V(s_t; \bar{\theta})$
            Update actor using policy gradient

                $$\nabla_{\theta_i} J(\theta_i) \approx \frac{1}{S} \sum_t \nabla_{\theta_i} \pi_{\theta_i}(s_t) \nabla_{a_i} Q^\pi_i(s_t, a^1_t, \dots, a^n_t) \quad (2.35)$$

        end for
        Softly $\tau$-update target network parameters for each agent $i$:

                $$\theta^i_{t+1} \leftarrow \tau \theta^i_t + (1 - \tau)\theta^i_{t+1} \quad (2.36)$$

    end for
end for
```

then used to update shared parameters in parallel. One the one side, the first benefit is that the number of parameters remains constant, regardless the number of agents. One the other side, the shared parameters are updated using the transitions from the same memory buffer, resulting in a more diverse and larger set of experience samples for training.

2.6 Related Works

A great inspiration for this thesis has come from the past LuxAI competitions. The first season[32] has highlighted how a collective intelligence leads to better results than fully decentralized learning process, encouraging agents to self-organize and acting emergent behaviors to accomplish the tasks. According to the authors, LuxAI environment with cooperative-competitive dynamics emphasizes the development of strategies which are based on multi-agent co-evolution, rather than hand-crafted collaboration mechanisms.
Despite of this optimistic premise, the winners of previous competitions had been agent leveraging on rule-based solutions and not RL-based ones. As huge drawback, this solutions cannot be replicated or adapted to LuxAI S3 because they were tailored for specific tasks, completely different from the current ones. Moreover, past seasons' environments were designed with perfect information about the game state, meaning each agent had the complete information of global map and knowledge of opponent's behavior [32] [33]. In LuxAI S3 [34], instead, the environment is shaped as *Partially Observable Stochastic Game* (POSG), meaning that agents have limited vision of the state observation and asymmetric knowledge of the game dynamics, and, in addition, the map is not static but changes the set-up after every game. Hence the need of questioning which development is more suited in this competition between rule-based or RL-based.

The dilemma of the superiority between rule-based algorithms, developed according to tailored heuristics, and RL-based algorithms has been clarified when RL-based chess engine AlphaZero [35] overcame the best rule-based chess engine Stockfish of that time. In the following years, many other RL-based engines has become the state-of-the-art for more complex games [36] [37], including video-games [38] [39], establishing the superiority of AI over human-crafted heuristics. Applying this vision to multi-agent environments as well, many scenarios have used MARL algorithms to overcome the challenges of cooperative and competitive games. Independent DQN [29] was firstly introduced to train two agents for playing double ping-pong games, finding the optimal strategy to keep the ball in the game as long as possible; then, it rapidly became a state-of-the-art algorithm in many benchmark libraries but it fails to learn to cooperate with allies and compete with enemies in more complex environments [40]. Actor-Critic based solutions are effective in decentralized approaches too, like Independent A2C [41] applied to traffic flow, which demonstrated robustness and sample efficiency through the collaboration of local agent with the close neighbors; however, this collaboration doesn't consider the totality of the agents and thus inapplicable in environment where the full cooperation is required. CTDE seems to be the optimal way to tackle this issue and Actor-Critic algorithms are again well-performing. MADDPG [31] has well addressed non-stationarity using centralized critic and decentralized actors, without suffering the scalability as the number of agents was growing. Similarly, Multi-Agent PPO [42] has reached optimal performances in competitive-cooperative boardgames and videogames, without any domain-specific algorithmic modifications.

Even though most of these algorithms have been trained using Self-play, i.e. exploiting only experience generated brand new during training process, other RL agents has reached interesting performances with Imitation Learning (IL), a process where expected behaviors are learned by imitating an expert's behaviors, provided through demonstrations [43]. This branch of learning is nevertheless under the assumption that imitated dynamics are following an optimal policy and, for complex multi-agents environment, this is not always guaranteed. Since LuxAI S3 is a brand new competition, no benchmark exists and the experts' performances cannot be exploited to boost the training.

Regarding credit assignment in cooperative scenarios, the need of shaping reward based on both local and global perspective is intuitively the best way to accomplish a global task: in [44] [45], a huge network of traffic lights builds a communication system to share information and avoids traffic jams, combining information from single intersection with close neighbors knowledge. Similarly, in competitive scenarios like MOBA (Multiplayer Online Battle Arena) games [46], crafting team-based rewards for each agent foster cooperative dynamics to achieve victory against opponent's agents. However, sparse reward functions are the main cause of misleading behaviors, creating situations where the a few agents effectively contributes to the objective while the majority benefits from their work, acting as "lazy agents" [47] and damaging learning process.

Finally, non-stationarity remains the elephant in the room and no MARL algorithm is able to generalize for all environments or guarantee convergence. Nevertheless, if the condition of homogeneity are satisfied, the problem can be relaxed and facilitating the solution development. In [48], shared policy network for all agents has achieved effective results, leveraging on continuous knowledge acquisition from shared state-action-state transitions.

Chapter 3

Materials and Methods

This chapter outlines the specifics of the competition and the solutions adopted to address the challenge. A panoramic of LuxAI is firstly given, examining both characteristics and boundaries. Then, an overview of MARL constraints translated to the competition points out the theoretical limitations and gives the basis for the solution development. Finally, the methodology shows how the solution has been built in practice.

3.1 LuxAI

The LuxAI Competition is an annual event focused on artificial intelligence (AI) and robotics, aimed at pushing the boundaries of autonomous systems in a competitive setting. Typically held as part of academic and industry conferences or tech events, the competition brings together researchers, engineers, and developers to showcase innovative solutions in the fields of robotics, AI, and machine learning. While it has evolved over time, the competition consistently challenges participants to create robots that can perform complex tasks autonomously, using cutting-edge AI techniques.
At the heart of the LuxAI Competition is the notion of autonomous intelligence, i.e. how machines can learn, adapt, and interact with their environment in real-time. Participants usually work with state-of-the-art robot platforms, which are equipped with various sensors, actuators, and onboard computational resources. These robots are tasked with completing specific challenges that involve tasks like navigation, object manipulation, and decision-making based on dynamic and unpredictable environments.
LuxAI S3[34] is the third season and poses the attention on the adaptability of two teams in complex dynamic multi-agent environment. In particular, each

agent competes in $1vs1$ games where the environment presents a fixed set of game mechanics and hyper-parameters for each mechanic.
The incipit of the competition states[1]:

> With the help 600+ space organizations, Mars has been terraformed successfully. Colonies have been established successfully by multiple space organizations thanks to the rapidly growing lichen fields on the planet. The introduction of an atmosphere has enabled colonists to start thinking about the future, beyond Mars. Mysteriously, new deep-space telescopes launched from Mars revealed some ancient architectures floating beyond the solar system, hidden in a midst of asteroids and nebula gas. Perhaps they were relics of a previous sentient species?
>
> Seeking to learn more about the secrets of the universe, new expeditions of ships were set out into deep space to explore these ancient relics and study them. What will they discover, which expedition will be remembered for the rest of history for unlocking the secrets of the relics?

3.1.1 Rules

Every $1vs1$ game consists of 5 matches where 2 teams face each other and the winner is the agent that wins more matches by collecting more points; matches terminate with either a victory or a loss. One match counts 101 steps, for a total of 505 steps per game.
There are no constraints regarding the nature of the agents, allowing both rule-based agents built with heuristics and AI agents.
The ranking system is *TrueSkill*[49]. The players' score is defined by the player's value μ and confidence score σ. After each match, the Trueskill algorithm updates each players μ, σ values according to the match result.
The competition starts on 9^{th} December 2024 and it's possible to submit agent's architecture until 10^{th} March 2025. After this date, two weeks are used to run further evaluations and on 24^{th} march 2025 the final leaderboard is published.

3.1.2 Environment

The map is a $2D$ grid of size (24×24) generated randomly; the environment will be different in different games but it will be the same in all 5 matches of the single game. Many elements define the environment:

[1]`https://www.kaggle.com/competitions/lux-ai-season-3/overview`

- **Units**: blue and red ships for, respectively, player 0 and player 1. A new unit spawns every n seconds, according to a hyper-parameter, and they spawn in top-left corner for player 0, bottom-right corner for player 1.
  They can move in 4 directions (up, down, left, right) or staying still. Units can overlap with other friendly units and have initial $energy = 100$, which can change in range $energy \in [0{,}400]$.
  They can also perform the "sap action" which points a tile on the map and, if there is an opponent's unit, the opponent's units looses energy according to an hyper-parameter. Each of the 6 actions have a cost in terms of energy and, if the energy drops to 0, the unit is eliminated from the game and will respawn in the respective corner.
  Single unit has a vision range which enables it to see the immediate square around it, with radius equal to a hyper-parameter. For example, if the $radius = 2$, the vision range will be a 5×5-square with the unit at the center.

- **Relic nodes**: yellow tiles that enables units to go near it and collect points. The unit gains points only if it stays on a relic tile with yellow border while the yellow relic tile doesn't give any point. If multiple units are stacked onto the relic tile, they still get only 1 points per step. Relic positions are fixed during all the game but they may not be revealed totally since the beginning; this means that in the first match only one relic sites is disclosed per half-map, in the second match at most two relic sites per half-map and so on up to a maximum of three relic sites per half-map.

- **Asteroids**: impassable black tiles which stop anything from moving spawning onto them. They can be moved by the environment,

- **Nebulas**: passable purple tiles which block vision of units. The "visual reduction" hyper-parameter is different for every game, randomized between [0,3]. The unit's vision is decreased by this factor, limiting what the unit can see, and it's even possible to reduce it to 0, making it unable to see itself but still being able to move. Moreover, nebula tiles can reduce unit's energy.

- **Empty tile**: tiles without anything. Units and nodes can be placed/move onto these tiles.

The environment is built symmetrically along the anti-diagonal, meaning that some elements, like asteroids, relics and nebulas, will have symmetric positions. In addition, nebulas and asteroids move symmetrically along the anti-diagonal while relics remain still.
When two or more units from opposing teams find in close quarters, there is an energy collision: the units with the highest energy wins the clash and remain while the losing units loose all energy and are eliminated from the game, waiting to

respwan.

Some hyper-parameters are randomly generated at the beginning of a game and determines the dynamics of the environment and units' interactions. Here are listed the main ones:

```
    env_params_ranges = dict(
        unit_move_cost=list(range(1, 6)),
        unit_sensor_range=[1, 2, 3, 4],
        nebula_tile_vision_reduction=list(range(0, 8)),
        nebula_tile_energy_reduction=[0, 1, 2, 3, 5, 25],
        unit_sap_cost=list(range(30, 51)),
        unit_sap_range=list(range(3, 8)),
        unit_sap_dropoff_factor=[0.25, 0.5, 1],
        # map randomizations
        nebula_tile_drift_speed=[-0.05, -0.025, 0.025, 0.05],
)
```

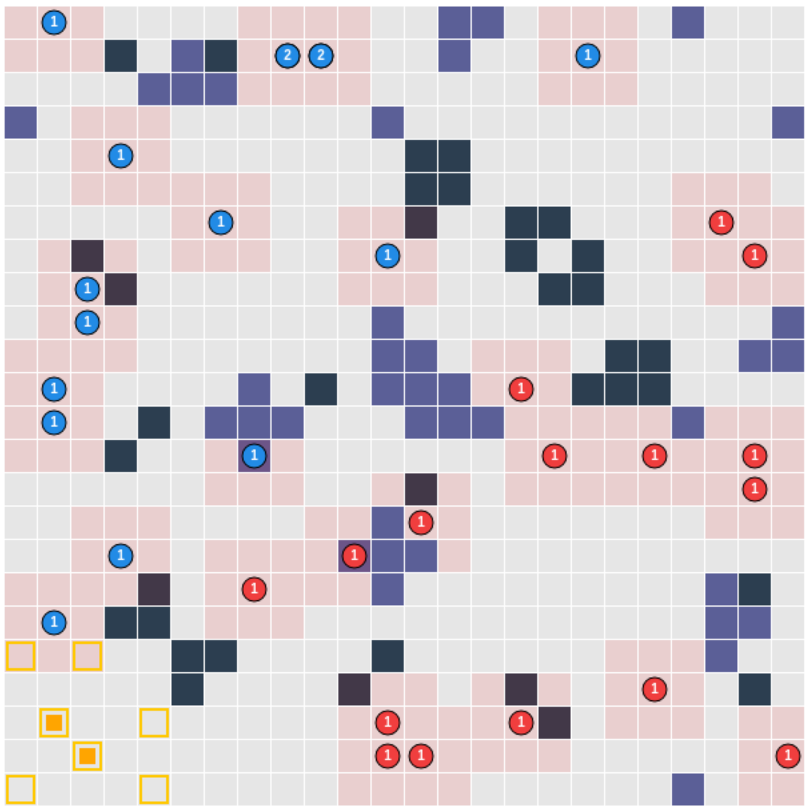


Figure 3.1: Example of LuxAI S3 environment.

3.2 Frameworks

LuxAI S3[2] is the source code of the competition.
`src` folder contains the environment and runner files to simulate games. `kits` folder holds two starter kits for building the agent, written respectively in `Python` and `Javascript`; the solution of this thesis has been developed extending the `Python` version. In order to to facilitate the vision of games' dynamics, a visualizer is available at *Visualizer*[3], while the ranking is accessible at *Ranking*[4].

OpenAI Gym[5] is a toolkit for RL research and one of the most authoritative frameworks for benchmarking, offering several single-agent and multi-agent environments and implementing state-of-the-art algorithms.In 2024, Gym has been "upgraded" to **Gymnasium**[6] and this version is the one used in the code.
LuxAI leverages on Gymnasium's wrapper classes, making easier the interaction agent-environment and facilitating the coding through interface functions that are common for every RL-based environment.

PyTorch[7] is a machine learning library and a pillar in neural network design. Written in `Python`, PyTorch performs tensor (i.e. n-dimensional arrays) manipulation and supports over 200 different mathematical operations. Among the feature, *dynamic graph computation* allows to change network behavior on the fly, rather than waiting for all the code to be executed, and *automatic differentiation* enables creation, training and optimization of deep neural networks. It works both on CPU and GPU. The architecture pipeline is built on this framework.

W&B[8] (Weights and Biases) is a machine learning development platform that allows programmers keep an eye on coding workflow and visualize the results through plots. The main benefit of this tool is that the tracking of training process is in real-time, acted through a logging system which connects the local machine and remote W&B server. Moreover, it provides plots and graphics, automating the visualization of hyper-parameters and outputs evolution in time.

[2]`https://github.com/Lux-AI-Challenge/Lux-Design-S3`

[3]`https://s3vis.lux-ai.org/`

[4]`https://www.kaggle.com/competitions/lux-ai-season-3/leaderboard`

[5]`https://www.gymlibrary.dev/index.html`

[6]`https://gymnasium.farama.org/index.html`

[7]`https://pytorch.org/`

[8]`https://wandb.ai/site/`

3.3 Methods

After the introduction of theoretical background notions and the current solutions in scientific literature made in Chapter 2, this section gives a deeper view on the contributions of the thesis work, outlining the problem formalization and defining the solution development. The overall goal has been to create autonomous agents able to play LuxAI S3 competition standalone, without any information distilled from human knowledge, and make them compete against other agent with aim of reaching the best ranking position.
Putting aside the ludic purpose, the thesis focus has been to attempt overcoming the main challenges of MARL paradigm and developing a stable architecture which could face adaptively the continuous evolution of environment dynamics. Here, the structured list of the objectives of this thesis:

- Building a scalable agent's architecture, able to control several sub-agents without the need of outstanding computational power and trained in feasible computational time.
- Addressing non-stationarity and make the agent aware of environment dynamics.
- Developing the policy as agnostic, i.e. without human-based directives or external heuristics.
- Fostering collaborative behaviors to reach the winning condition, tailoring with precision the credit assignment.
- Evaluate performances and behaviors of the algorithms, compared to other agents' results in competition ranking.

3.3.1 Observations, Actions, Rewards

LuxAI S3 competition falls into the category of multi-agent competitive games, as two opponent teams need to face in order to pursuit a winning condition. According to environment specifics, the state s_t and the observation o_t are not overlapping and thus the general framework to model the problem is *Partially Observable Stochastic Game* (POSG). Specifically, each agent possess different information about the state and decision-making system is acted upon this assumption; the actions performed by $\mathcal{N}$ sub-agents are combined into a Joint Action space $\{A^i\}_N$. Consequently, the credit assignment system is tailored on the contribution of the joint action space

and it uses different reward functions $\{\mathcal{R}^i\}_N$ to evaluate those interactions, which are properly discounted by a factor γ when calculating the expected return of state t. Since no transition probability matrix is given, the policy function is stochastic, leveraging on approximation techniques to estimate the optimal policy.
To sum up, each agent can formalize step t of the game as the tuple:

$$< \mathcal{N}, \mathcal{O}, \{\mathcal{A}^i\}_N, \mathcal{P}, \{\mathcal{R}^i\}_N, \gamma > \tag{3.1}$$

Observation $\mathcal{O}$ is the knowledge of environment state that is shared to the agent. As seen in 3.2, incomplete vision of the environment is available and this depends on how the units are distributed over the map. In particular, the environment provides a `dict` containing: position and energy of units, both player's and opponents; a units mask to identify which units are active; a sensor mask to show which map tiles are visible by the player's units; a map of tiles feature, showing if the tiles are empty, nebulas or asteroids; a relics mask for relic site visibility, even though only the yellow relic tile is shown and not the yellow-border tiles that give points; some statistics like points, counter of match/game steps.

```
obs = dict(
    "units": {"position": Array(T, N, 2), "energy": Array(T, N, 1)},
    "units_mask": Array(T, N),
    "sensor_mask": Array(W, H),
    "map_feature": {"energy": Array(W, H), "tile_type": Array(W, H)},
    "relic_nodes_mask": Array(R),
    "relic_nodes": Array(R, 2),
    "team_points": Array(T),
    "team_wins": Array(T),
    "steps": int,
    "match_steps": int
  )
```

In particular, $T = 2$ is the number of teams, $N = 16$ is the number of units for each player's agent, width $W = 24$ and height $H = 24$ are the dimensions of the map, R = [1,3] is the number of relics.

In order to compact all this knowledge and give it to the deep neural network, it has been embedded into a custom *PyTorch* tensor which represents the input knowledge of the agent. More specifically, a 6@24 × 24 tensor depicts the 24 × 24 map using 6 custom channels, meaning that every map tile is identified by 6 features which can be either a positive number if the feature is present or 0 otherwise.

1. *Exploration channel*: 1 if the tile has already been explored, 0 otherwise.

2. *Relic channel*: 1 if the tile is a yellow tile relic, 0 otherwise.
3. *Nebula channel*: 1 if the tile is a nebula, 0 otherwise.
4. *Asteroid channel*: 1 if the tile is a asteroid, 0 otherwise.
5. *Player's unit channel*: $n/16$ if the tile is occupied by n player's unit, 0 otherwise.
6. *Opponent's unit channel*: $n/16$ if the tile is occupied by n opponent's unit, 0 otherwise.

The idea is to create a kind of one-hot-encoding tensor, making the values bounded between [0,1], limiting the variance. As last, one more channel is introduced to discriminate which unit is considered; since this tensor is the input for the neural network and the output is an action, there shall be an indicator of which sub-agent is actually taken into account. That's why the additional channel is unit's energy which is very unlikely to be equal to another unit's energy.

7. *Unit's energy channel*: $e/400$ if the tile is occupied the unit considered, 0 otherwise.

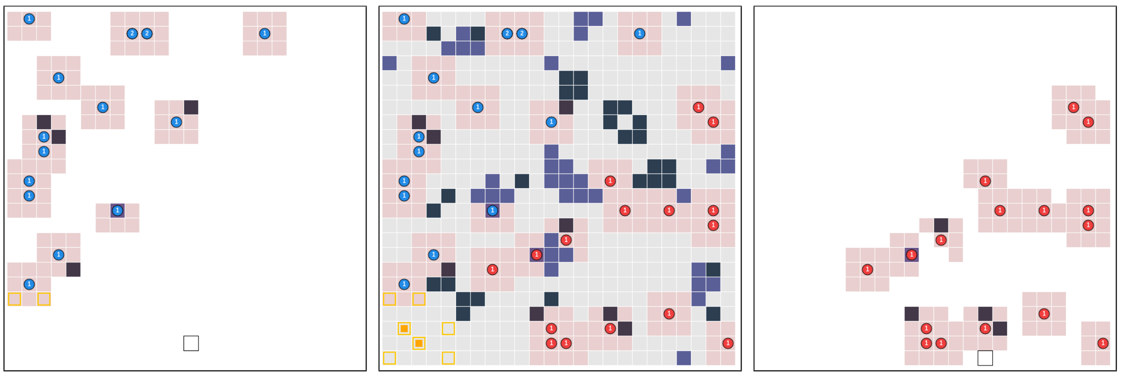


Figure 3.2: Comparison of player 0's vision (left), global state observation (center) and player 1's vision (right).

Joint Action space $\{A^i\}_N$ is the combination of the discrete actions that units can do. The possible actions are: 0. staying still, move 1. up, 2. right, 3. down, 4. left, 5. sap. For this last action, a position is also required, identified by row R and column C. The action is so coded as `Array(1,3)` where the first value is the action type and the other two ones are either positions for sap action or `0,0`.

Reward function $\{\mathcal{R}^i\}_N$ is the most important element in Reinforcement Learning since it's the driving factor of agent's learning, guiding it to a better

directions if the agent is behaving wrongly or encouraging it if it discovers the right trajectory of actions. In [27], the authors propose a formulation which centralizes the role of time in credit assignment; in fact, temporal contiguity may viewed as a proxy for causal influence, since the likelihood of action is directly proportional to how many future steps are effectively impacting the expected return.
In this sense, a global reward can be formulated, taking into account the match-steps. On the one side, the final objective is obviously to collect as many points as possible but, on the other side, at the beginning of a match, the most important task is know where the relic sites are and so the exploration has the priority.

$$\mathcal{R}_{global} = (1 - W_t)E_t + W_t \frac{1}{16}\frac{\delta\Delta_P}{\delta t} \tag{3.2}$$

where W_t is the ratio $\frac{step_t}{101}$, E_t is the exploration ratio $\frac{visible\ tiles}{24^2\ tiles}$, $\frac{1}{16}\frac{\delta\Delta_P}{\delta t}$ is the difference of points between the player and opponent, only for the step t and regularized over the 16 units.
In [47], the introduction of scaled sparse reward has brought to misleading behaviors, making agents lazy. In fact, due to a shared reward that is assigned equally to all agents, it's very likely that only a few agents actively contribute to the target goal while the majority acts without interest and benefits of others' effort. In this sense, hand-crafted dense rewards may encourage the adoption of proactive behaviors, evaluating with scalar values the single actions.

$$\mathcal{R}_{single} = B_{action} \cdot r^+ + (1 - B_{action}) \cdot r^- \tag{3.3}$$

where the $B_{action} = \{0,1\}$ is a crafted boolean evaluation variable of the action; if the action is evaluated as good, $B_{action} = 1$ and the scalar value r^+ is positive, otherwise the scalar value is negative r^- and $B_{action} = 0$. Examples of bad behaviors can be: trying to go outside map boundaries, entering into nebula tiles, bumping into asteroids or perform sap action to empty tiles. Examples of proactive behaviors can be: discover new tiles through exploration and collecting points.
In [44], the authors give a solution to avoid traffic flow jams and combine the local reward of the single traffic intersection with the global reward of the whole city traffic flow. The underneath idea is that credit assignment is more complex in multi-agent environments where explicit cooperation is needed, so a joint combination of reward functions may help the learning process. Inspired by this, a third option can be formulated mixing the rewards of single agents with the global perspective reward.

$$\mathcal{R}_{mix} = \mathcal{R}_{global} + \mathcal{R}_{single} \tag{3.4}$$

3.3.2 Algorithm selection

In chapter 2, several single-RL and MARL algorithms are proposed, offering a wide perspective on the different techniques used in current literature. In order to select

the most suited one, it's worth recalling the main obstacles in LuxAI S3.
The number of units is an issue to the scalability of the algorithm. Algorithms like IDQN[29] and MADDPG[31] have achieved notable results in multi-agent environment but they requires the training of different neural networks for each one of the agents. In this way, 16 parallel network optimizations are needed per player, without considering that some algorithms like Actor-Critic based ones have more than one policy to be optimized, plus a global-level optimization for CTDE which have centralized training. It's evident that, even with a HPC cluster, the computational time would be too long to perform dozens of experiments. Moreover, the units are not always present in the game, due to elimination or simply for not spawning in the beginning, fostering the fluctuation of policy convergence.
According to Theorems 3 and 4, since there can be a permutation of individual policies as the agents are identical to each others, the optimal joint policy can be represented as a joint policy consisting of identical policies for all agents. Hence, the LuxAI can be relaxed to Homogeneous MARL.

In [50], a performance benchmark between DQN, A2C and PPO is proposed. For the sake of the thesis, a comparison between Off-Policy algorithms, like DQN, and On-Policy algorithms, like A2C and PPO, is proposed as well. On the one side, the authors show how DQN achieves high rewards in shorter durations, suiting best for quick adaptation and time-efficient learning. On the other side, PPO and A2C experienced more pronounced fluctuations in their learning trajectories, needing more training time to converge; despite the overall better performances of PPO over A2C, the paper shows that the computational time of PPO training is way higher than A2C and that's why A2C is chosen to represent On-Policy algorithms.

Having taken in consideration this premises, the algorithms implemented are a novel combination of homogeneous agents paradigm, DQN and A2C: *Homogeneous Deep Q-Network* (HDQN) and *Homogeneous Advantage Actor-Critic* (HA2C). These two algorithms are both CTDE-based, meaning that training and optimization are carried on by a central super-agent, while the execution of the actions is performed locally by the single sub-agents/units, in a decentralized fashion. One the one side, this approach avoids the computational complexity of CTCE's execution, which would have created a huge single action space, built as the combination of all actions of all agents and thus $|\{A^i\}_N| = 6_{unit1} \cdot 6_{unit2} \cdot [\dots] \cdot 6_{unit16} = 6^{16}$ exponentially exploding. One the other side, it overcomes the limitation of DTDE training, which optimize every agent independently, making grow the computational cost for each policy and lengthening the overall training phase time.

The snapshot of **HDQN** can see in Algorithm 9.
The behavior network is the mapping function which takes in input a state s_t,

predicts the $Q(s_t, a_t)$ values for each action a and has a set of parameters θ; the target network is the mapping function which represents the expected $Q^*(s_{t+1}, a_{t+1})$ values of the next state s_{t+1} and has set of parameters $\bar{\theta}$.
When computing the loss $\mathcal{L}(\theta)$, the predicted $Q(s_t, a_t)$ value is compared with the sum of the reward r_{t+1} and expected next $Q^*(s_{t+1}, a_{t+1})$ value. In this way, there will be convergence as soon as the two networks will output the same Q-values.
Since it's Off-Policy method, the behavior network uses random samples from the buffer $\mathcal{D}$. The behavior network optimization is performed as:

$$\theta_{k+1} \leftarrow \theta_k + \alpha(r_{t+1} + \gamma \max_a Q^*(s_{t+1}, a_{t+1}; \bar{\theta}_{k+1}) - Q(s_t, a_t; \theta_k))\nabla_{\theta_k} J(\theta_k) \tag{3.5}$$

The target network is updated every C episodes or, in order to prevent oscillations in learning process, softly according to small values of τ after every episode:

$$\bar{\theta} \leftarrow \theta \quad or \quad \bar{\theta} \leftarrow (1-\tau)\bar{\theta} + \tau\theta \tag{3.6}$$

Algorithm 9 Homogeneous Deep Q-Network (HDQN)

```
Initialize behavior network $Q$ with random parameters $\theta$
Initialize target network $Q^*$ with parameters $\bar{\theta}$
Initialize an empty replay buffer $\mathcal{D}$ to capacity $M$

for each episode do:
    for each step episode do:
        for agent $i = 1, \dots, N$ do:
            With probability $\epsilon$, choose random action $a_t^i$
            otherwise, with probability $(1-\epsilon)$, $a_t^i = \max_{a^i} Q^*(s_t, a^i; \bar{\theta})$
        end for
        Apply actions $\{a_t^i\}$; observe reward $\{r_{t+1}^i\}$ and new state $s_{t+1}^i$
        Store transition $(s_t, a_t^i, r_{t+1}^i, s_{t+1})$ into buffer $\mathcal{D}$
        Set $y_k^i = \begin{cases} r_{k+1}^i & if\ terminal\ s_{k+1} \\ r_{k+1}^i + \gamma \max_{a^i} Q^*(s_{t+1}, a_{t+1}^i; \bar{\theta}) & if\ non\ terminal\ s_{k+1} \end{cases}$
        Compute $\mathcal{L}(\theta) = y_k^i - Q(s_t, a^i; \theta)$
        Update parameters $\theta$ by minimizing the loss $\mathcal{L}(\theta)$
    end for
    Every $C$ episodes, update target network parameters $\bar{\theta}$
    Alternatively, softly $\tau$-update target network parameters $\bar{\theta}$
end for
```

Similarly, **HA2C** implementation is shown in Algorithm 10.
The actor network is the mapping function which takes in input a state s_t, predicts the $Q(s_t, a_t)$ values for each action a and has a set of parameters ϕ; the critic network evaluates the expected goodness of a state s_t assigning a $V(s_t)$ value.
The advantage function $Adv(s_t, a_t)$ is used to quantify how much higher the expected return is when applying the specific action a_t, compared to following the actor policy π in state s_t. The advantage is positive or negative if, respectively, it reaches a higher or lower return in comparison to the expected return of the state-value. In this way the actor formulates the policy to choose the best actions, while the critic gives a feedback if the prediction is better or worse than what the critic function expects. Since it's On-Policy method, the optimization is carried on considering only the sample batches from the current policy and not leveraging batches from other policies, and it follows:

$$\phi_{k+1} \leftarrow \phi_k + \alpha Adv(s, a; \phi; \theta) \nabla_{\theta_k;\phi_t} J(\phi_k; \theta_k) \qquad \theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\theta_k) \tag{3.7}$$

Algorithm 10 Homogeneous Advantage Actor-Critic (HA2C)

```
Initialize actor network function $\pi$ with random parameters $\phi$
Initialize critic network $V$ with parameters $\theta$

for each episode do:
    for each step episode do:
        for agent $i = 1, \dots, N$ do:
            Sample action $a_t^i \sim \pi(\cdot|s_t; \phi)$
        end for
        Apply action $\{a_t^i\}$; observe reward $\{r_{t+1}^i\}$ and new state $s_{t+1}$
        if $s_{t+1}$ is terminal then:
            Advantage $Adv(s_t, a_t) \leftarrow r_{t+1} - V(s_t; \theta)$
            Critic target $y_t \leftarrow r_{t+1}$
        else:
            Advantage $Adv(s_t, a_t) \leftarrow r_{t+1} + \gamma V(s_{t+1}; \theta) - V(s_t; \theta)$
            Critic target $y_t \leftarrow r_{t+1} + \gamma V(s_{t+1}; \theta)$
        end if
        Compute actor loss $\mathcal{L}(\phi) = -Adv(s_t, a_t) \log \pi(a_t|s_t; \phi)$
        Compute critic loss $\mathcal{L}(\theta) = y_t - V(s_t; \theta)$
        Update parameters $\phi$ by minimizing the actor loss $\mathcal{L}(\phi)$
        Update parameters $\theta$ by minimizing the critic loss $\mathcal{L}(\theta)$
    end for
end for
```

3.3.3 Training and Learning

The design of policy has taken in consideration [11] to evaluate the depth and width of neural network architecture.
On the one hand, deeper networks can capture a broad range of patterns, helping to reveal complex relationships in data. They tend to build hierarchical representations of knowledge when processing abstract input features. However, they face challenges such as vanishing or exploding gradients during the optimization process due to multiple layers, and they are prone to overfitting if not properly designed, potentially memorizing the training data structure rather than learning generalizable rules.
On the other hand, wider networks can process more features simultaneously, which speeds up both training and inference times. They are typically more robust to outliers or input variations due to the large number of neurons. However, their main drawback is the "curse of dimensionality," leading to high memory and computational costs, with diminishing returns in performance as the number of neurons increases. Due to the higher computational time in training process, the architectures has followed a deeper design, limiting the number of neurons per layer to 128.
Due to the necessity of identifying spatial relationships across map environment, `Conv` layers have been used to recognize features from the input tensor $7@24 \times 24$. In order to avoid potential instability during learning or overfitting, there are some `BatchNorm` layers between convolutional blocks too. At end, a series of `Linear` layers are used to map the output into 6 $Q(s, a)$ action-values, for HDQN policy and HA2C Actor's policy, or 1 $V(s)$ state-value, for HA2C Critic's policy. An overview of the respective architectures can be seen in Figures 3.3 3.4.

The training process for HDQN and HA2C followed similar pipelines. Each transition $s_t, a_t, r_{t+1}, s_{t+1}$ is stored into the respective buffers and then used for the optimization phase. However, due to the On-Policy nature of HA2C, its buffer storage is limited to 8080 because it's the maximum number of total transitions in one game, considering 505 steps and 16 transitions for each unit in one match step.

Finally, the games have been generated using only "Self-play", meaning the agent is trained by playing against itself, widely used by [18] [35] [36]. The agent learns from its own experience by improving through interactions with itself, progressively improving over time. Alternatively, Imitation Learning could have been used but there was no guarantee that the games used were from "real experts of LuxAI S3", since no benchmark has ever been published. Moreover, it would have biased the idea of a completely agnostic agent if any information was stilled from external knowledge.

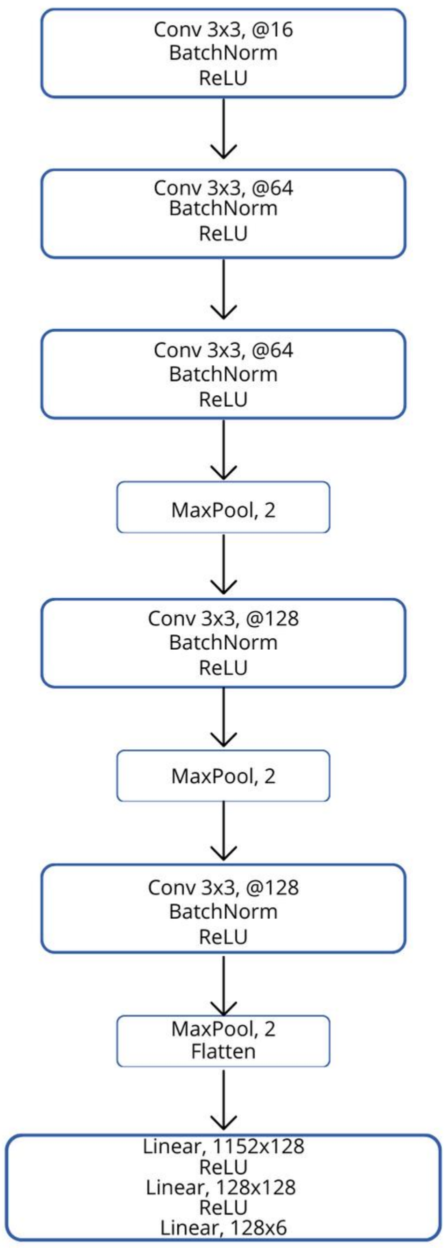


Figure 3.3: DQN architecture. Both behavior and target policy networks share the same structure.

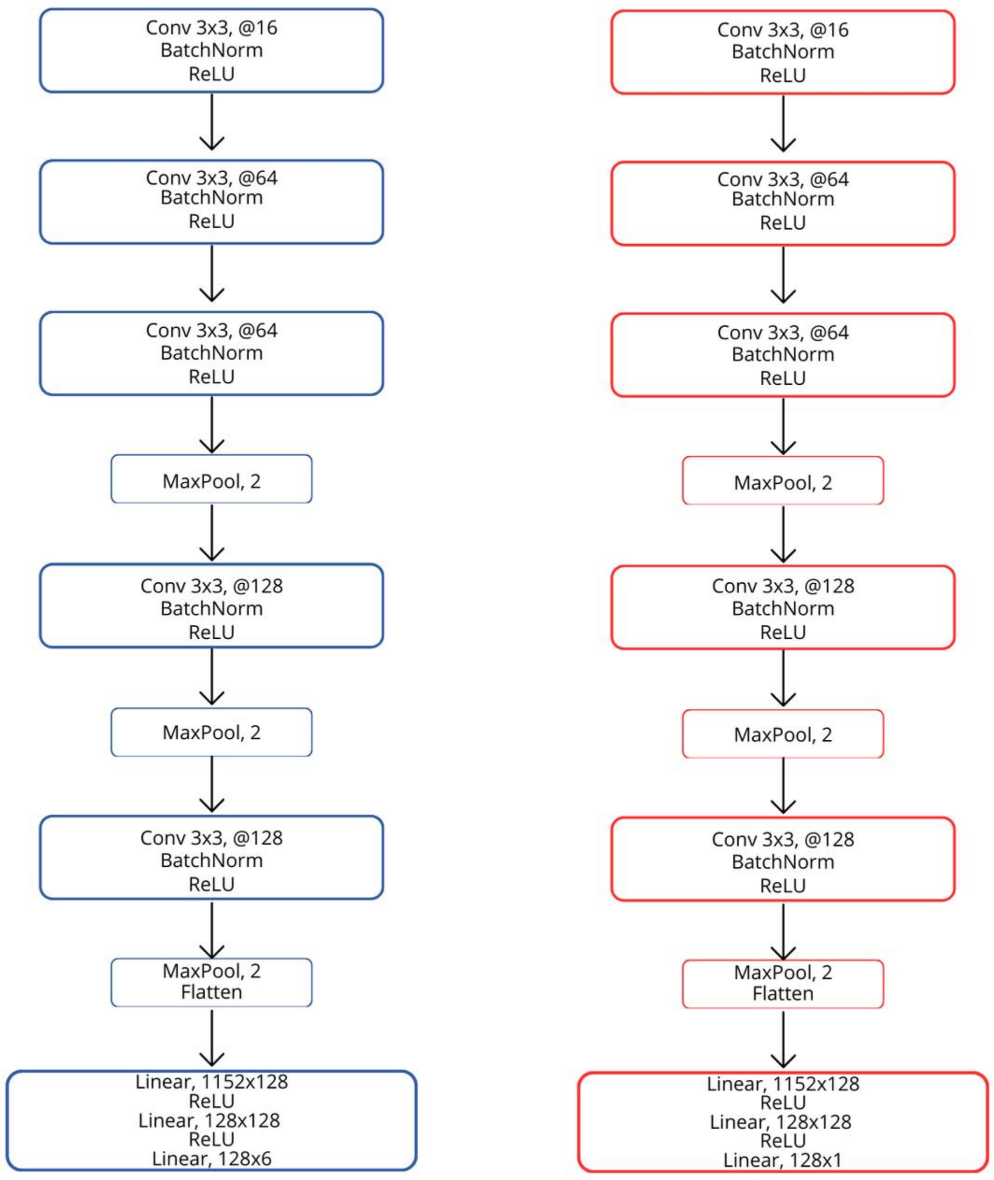

Conv 3x3, @16 BatchNorm ReLU
Conv 3x3, @64 BatchNorm ReLU
Conv 3x3, @64 BatchNorm ReLU
MaxPool, 2
Conv 3x3, @128 BatchNorm ReLU
MaxPool, 2
Conv 3x3, @128 BatchNorm ReLU
MaxPool, 2 Flatten
Linear, 1152x128 ReLU Linear, 128x128 ReLU Linear, 128x6
Conv 3x3, @16 BatchNorm ReLU
Conv 3x3, @64 BatchNorm ReLU
Conv 3x3, @64 BatchNorm ReLU
MaxPool, 2
Conv 3x3, @128 BatchNorm ReLU
MaxPool, 2
Conv 3x3, @128 BatchNorm ReLU
MaxPool, 2 Flatten
Linear, 1152x128 ReLU Linear, 128x128 ReLU Linear, 128x1

Figure 3.4: A2C architecture. Critic's policy network (left), Actor's policy network (right).

Chapter 4

Results

This chapter presents the experimental results of the two models previously described: HDQN (Homogeneous Deep Q-Network) and HA2C (Homogeneous Advantage Actor-Critic). The experiments are presented following the evolution of the reward formulation, as it represents the driving factor of a Reinforcement Learning process. The balance between optimality and computational complexity has been the main challenge, bringing many difficulties when designing the tests' workflow, since the hyper-parameter tuning wasn't straightforward for both the solutions. In particular, the chapter will give a panoramic of how the architectures have performed in terms of reward and points gained during the games, underlining which cooperative or greedy behaviors have been manifested by the agents. Finally, the competition rank is shown, as well the comparison with the top tier's agents of the competition, leaving room for reasoning on which techniques have reached better results.

4.1 Early setup

Before stepping into the explanation of the results, it's worth to specify how the tests have been carried out. Since Reinforcement Learning, and for extension MARL, doesn't have a predefined dataset as in Supervised Learning, it shall build its own one and iterate over a huge number of training steps. The number of steps required is not a formal statement that can be decided a-priori but more a "rule-of-thumb" based on the complexity of the task. However, many frameworks provide benchmarks where they show various algorithms performances, giving an idea of the cardinality of such number.
RLlib[51][52] is an open source library and one of the most authoritative frameworks in this field, which has build many scalable environment and implemented

several state-of-the-art algorithms, both single and multi-agent RL. In their GitHub repository[1], DQN and A2C algorithms have been trained with roughly $\sim 10^7$ steps in various environments.
LuxAI S3 environment is obviously brand new and no framework has never published any benchmark on it. However, considering that every game is made of 505 steps and the transition state-action-state is stored for each of the 16 units, the greatest number of transitions is $\sim 8 * 10^3$ for each player per game. Due to the high complexity and variance of the dynamic environment, the number of games played has been fixed to 6000, leading to $\sim 4.8 * 10^7$ transitions/steps.
In Table 4.2 a summary of the hyper-parameters and fixed training parameters, already discussed in the Methodology chapter. Other parameters will be clarified more specifically within experiment results' sections.

Algorithms	**HDQN**	**HA2C**
Type of learning	Off-Policy	On-Policy
# of trained networks	1	2
Buffer size	50000	8080
# of training games	6000	6000
# of validation games	10	10
Batch size	1024	1024
Loss	Huber	Huber
Optimizer	Adam	Adam
ϵ	[0.01, 1.0]	[0.01, 1.0]
ϵ **decay**	0.995	0.995
γ	0.9	0.9
Time of training	$\sim$ 11 hours	$\sim$ 20 hours

Table 4.1: Hyper-parameters set

The γ parameter represents how much the future rewards are impacting on the expected return of a given state, as in equation 2.2. Low values implies that agents only care about short coming benefits, while greater values mean a long term strategy which takes in consideration longer trajectories of actions. Generally, setting $\gamma = 0.99$ or $\gamma = 0.999$ is useful to extend the temporal horizon till the terminal state or state far enough from the present.
Despite of this, since the environment map is relatively small (24×24) and every match has only 101 steps, the weight of the last 101^{th} action on the expected return of state $s_{101} \rightarrow s_t$ will be respectively at least $w_{s_{101 \rightarrow 0} | \gamma = 0.99} = \gamma^{101} \approx 0.36$

[1]`https://github.com/ray-project/rl-experiments`

and $w_{s_{101\to0}|\gamma=0.999} = \gamma^{101} \approx 0.9$. These values are the lowest possible weights, as the horizon of 101 steps is the highest; in case the horizon is lower, for example $s_{101} \to s_{50}$, the weight will be higher $w_{s_{101\to50}} = \gamma^{51} > w_{s_{101\to0}} = \gamma^{101}$.
Since the longest path to reach a relic tile, and thus collecting points, is when the unit and relic are on the opposite corners of the map, a unit does take at most 48 actions to complete this path. Now, every other action over the 48^{th} would be overabundant to reach the target tile and the weighted reward should not impact too much on the expected return. That's why $\gamma = 0.9$ was more appropriate and $w_{s_{101\to0}|\gamma=0.9} = \gamma^{101} \approx 2 * 10^{-5}$ small enough to the neglect the effect of too long trajectories of actions.

Finally, ϵ parameter refers to the randomness of the policy's action selection. If the value is closer to 1, the action selected is very likely to be chosen randomly, while, if the value is closer to 0, the action is selected according to the current policy. The trade-off exploration-exploitation is essential to learn new dynamics, following random trajectories and then updating the policy on new discoveries. However, exploration phase shall not take the majority of training steps and, due to limited computational power and long trainings (especially for HA2C), ϵ-decay has been set to 0.995, in order to limit the exploration till $\sim 1000^{th}$ game. It's worthy to underline that the minimum value of the parameter is $\epsilon = 0.01$, meaning that in 1% of the cases there will still be a random action choice.

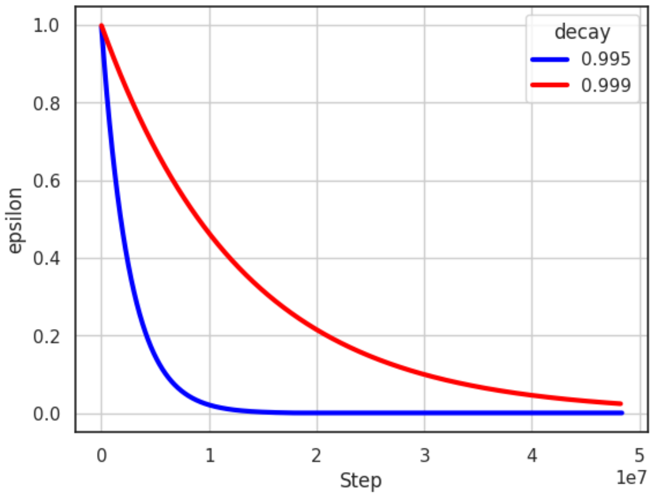


Figure 4.1: Epsilon decay influence on ϵ-greedy action selection along the training steps.

The training has been performed using a workstation with the following specifics:

Component	**Detail**
CPU	AMD RyzenTM 9 7900X
CPU's Cores	12 Core 4.5GHz
GPU	NVIDIA GeForce RTX 4080 16GB
CUDA	11.5
RAM	DDR5 2x32GB
OS	Ubuntu 22.04.5 LTS
Python	3.10
PyTorch	2.5.1
Gymnasium	1.0.0

Table 4.2: Hardware specifics.

and the GitHub repository with the code can be viewed at [2].

4.2 Global reward

The first set of experiments regards the application of a global reward, with aim of observing which behaviors are stimulated. The reward adopted is made of two components which are time-dependent, meaning that their weights are influenced by the match step.

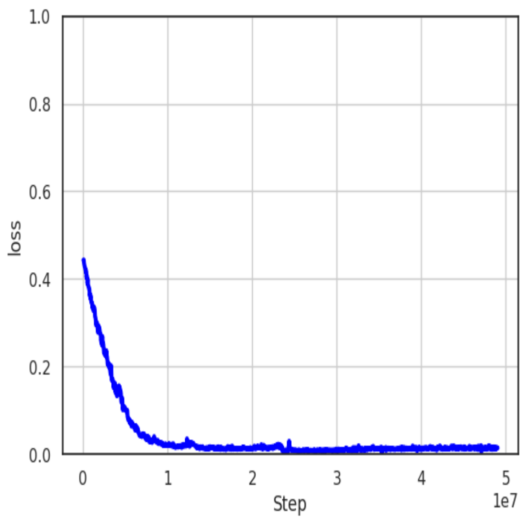
$$\mathcal{R}_{global} = (1 - W_t)E_t + W_t \frac{1}{16} \frac{\delta \Delta_P}{\delta t} \tag{4.1}$$

where W_t is the ratio $\frac{step_t}{101}$, E_t is the exploration ratio $\frac{visible\ tiles}{24^2\ tiles}$, $\frac{1}{16}\frac{\delta \Delta_P}{\delta t}$ is the difference of points between the player and opponent, only for the step t and regularized over the 16 units. In this way, $\mathcal{R}_{global}$ gives more importance to map exploration at the beginning of the match, in order to find as soon as possible the relic sites, and considering point collection more relevant in the late steps, since the victory condition depends on them.

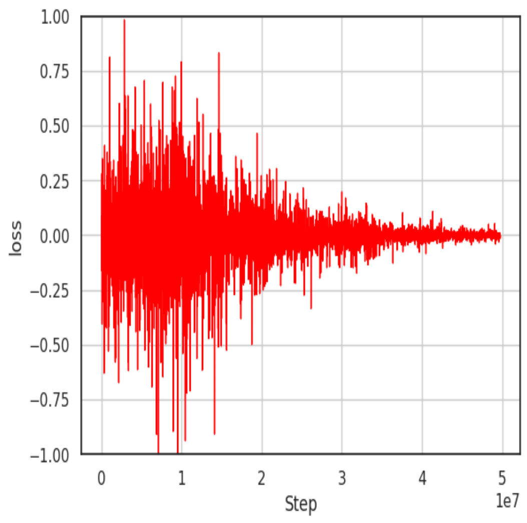
In Fig. 4.2, the losses of training phase shows that both architectures have converged. On the one side, it's interesting to notice how HDQN has a smooth learning curve, reaching the stability almost after the "exploration" phase; this happens because the behavior policy accesses batches with more variance, coming from a buffer memory which stores $5 * 10^4$ transitions, and, thus, is able to better generalize over different configurations of the environment.
On the other side, HA2C Actor's policy has access only to one environment configuration per optimization step, since the buffer memory stores transitions

[2]`https://github.com/polrizzo/marl_luxai_s3`

from one game only. Moreover, the HA2C Loss is calculated on the *Advantage Function* [5] and so the curve is continuously oscillating between positive and negative values, depending on the similarity between the Actor's prediction and the Critic's expected value.



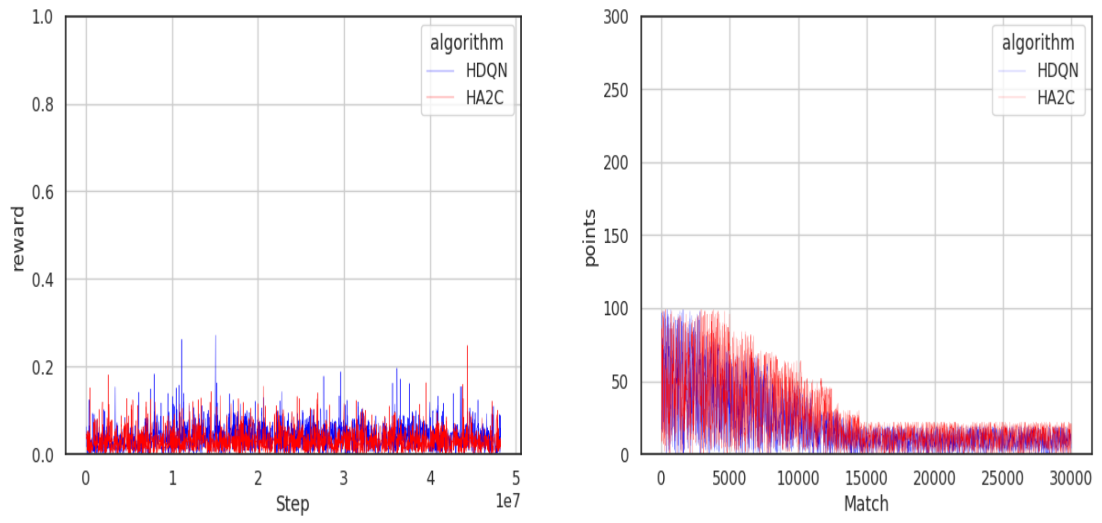
(a) HDQN's Policy Loss.



(b) HA2C Actor's Policy Loss.

Figure 4.2: Loss convergence comparison when using global reward.

However, despite the convergence of both architectures, the number of points collected has been very low since there was no direct incentive to move closer to relic sites. In fact, considering $\frac{\delta\Delta_P}{\delta t}$ is not an effective indicator to boost the gain of more points, since no specific strategy is evaluated by it, and exploration reward gives instead the major contribution, assigning positive bonus to the exploration strategy only.
In part A. of Figure 4.3, the global reward is generally low, since the contribution of exploration reward is decreased by temporal factor $(1 - W_t)E_t$ and the contribution of points difference $\frac{\delta\Delta_P}{\delta t}$ is practically irrelevant.
In part B. of Figure 4.3, the visualization of points collected during the games is shown. In particular, the x-axis refers to the matches played, hence 6000 games $\times$ 5 matches per game, resulting in a total of 30000 matches. Clearly, during the exploration phase, random actions have brought to better results, so global reward system is not able to beat an agent which plays and directs actions randomly.
Due to the poor performances of a sparse reward, hence the need of a more tailored dense reward.



(a) Global Reward trend. **(b)** Points gained at end of each match.

Figure 4.3: Rewards and Points gained when using a global reward.

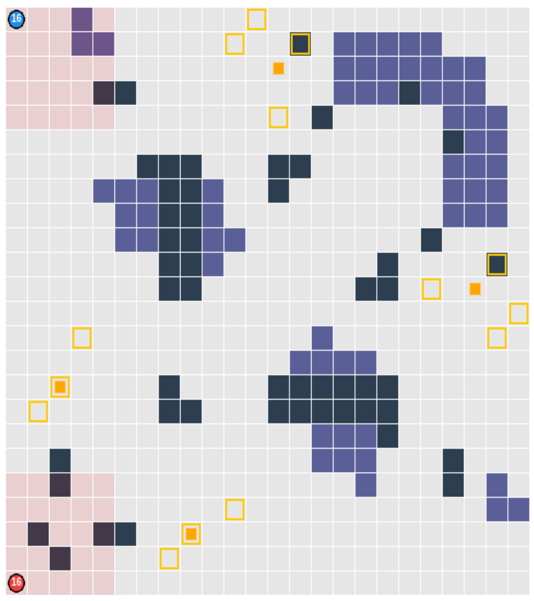
4.3 Single reward

Since the previous credit assignment doesn't stimulate a proper evaluation of the environment dynamics, it has been crafted a reward based on the action performed locally by the unit and on the unit's position in the map. In contrast to previous system which uses continuous values, the single reward outputs only scalar values, bound to the set $\{-1,\ 0, +1\}$.

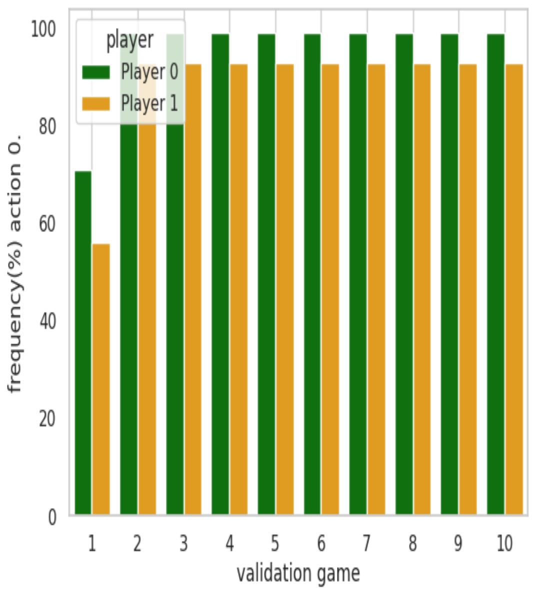
$$R_{single} = \begin{cases} -1 & if\ unit\ moves\ outside\ the\ map \\ -1 & if\ unit\ bumps\ into\ nebula\ tile \\ -1 & if\ unit\ bumps\ into\ asteroid\ tile \\ -1 & if\ sap\ action\ on\ empty\ tile \\ +1 & if\ in\ relic\ site\ range \\ +1 & if\ unit\ discovers\ new\ tiles \\ 0 & otherwise \end{cases} \tag{4.2}$$

The reward is structured as `If-Elif-Else` cycle where, starting from the first condition, the agent checks if the statements are either satisfied, i.e. outputting the value, or not, i.e. keeping on until the last condition.
Despite of this straightforward strategy, the main drawback was the unit's spawn in the corner. Since the starting position is always upper-left corner or bottom-right corner, two out four movements lead always outside the map, implying an

immediate negative reward. This has conditioned the learning process, making the unit refuse some specifics actions and propagating this vision in long-term horizon. The same has happened for the sap action: since in the exploration phase it is unlikely to find many opponent's units, the sap action always targets empty tiles, implying again an immediate negative reward.
In part A. of Figure 4.4, it can be seen how the player 0's policy only suggests to stay still in order to not receive negative rewards. Similarly, the player 1's policy suggest to either move left or staying still if the unit goes outside the map or bumps into obstacles. The frequency of action 0, i.e. staying still, has been summed up in part B, of Figure 4.4, considering as benchmark the 10 validation games. Following the experimental results, the single reward has been reshaped to a reward



(a) Snapshot of match. Both players' are stuck in the same position.



(b) Frequency% of action 0 during the 10 validation games.

Figure 4.4: Misleading behaviors due to strict single reward.

less static, combining both scaler values for local actions and continuous values according to unit's position. Applying too drastic negative rewards was clearly misleading for the agent, as well as avoiding to consider the relative position of unit with respect to relic sites. More specifically, $\mathcal{R}_{relic}$ is calculated depending on the relics discovered: if no relics is discovered so far null value is assigned, otherwise either 1 if in relic range or a continuous value calculated as the distance from the relic. The new reward becomes a combination of multiple sub-rewards:

$$\mathcal{R}_{move} = \begin{cases} -0.25 & if\ outside\ map \\ -0.25 & if\ bumps\ nebula \\ -0.25 & if\ bumps\ asteroid \\ -0.25 & if\ staying\ still \\ \ \ 0 & otherwise \end{cases} \qquad \mathcal{R}_{relic} = \begin{cases} +1 & if\ in\ relic \\ 1 - d(u, rel) & if\ outside\ relic \\ 0 & otherwise \end{cases} \tag{4.3}$$

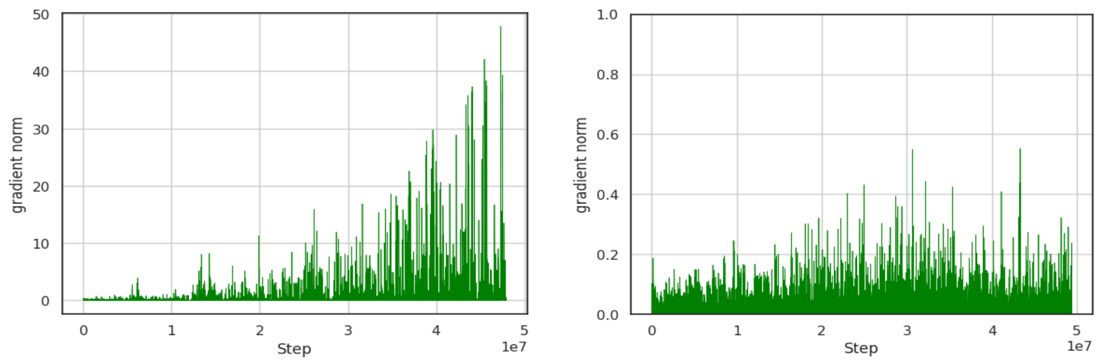
$$\mathcal{R}_{effect} = \begin{cases} +0.25 & if\ new\ tile/relic\ discovered \\ -0.25 & if\ sap\ on\ empty\ tile \\ \ \ 0 & otherwise \end{cases} \qquad \mathcal{R}_{single} = \mathcal{R}_{move} + \mathcal{R}_{relic} + \mathcal{R}_{effect} \tag{4.4}$$

Before stepping into the results, it's worth underlining that the new reward was more complex than the previous ones, bringing more instability to the learning process. In fact, when applying the gradient descent during backpropagation, the gradient norm appeared to be very high and, in order to prevent future oscillation, some optimization tricks as been adopted. The first one has been the clipping technique, which limited the gradient norm to $||grad|| \leq u_{clip}$. The second one regards HA2C Critic's policy and HDQN target policy: the updates has not been done every C steps as:

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\theta_k) \tag{4.5}$$

but rather τ-softly, being updated after every game with a continuous slow optimization:

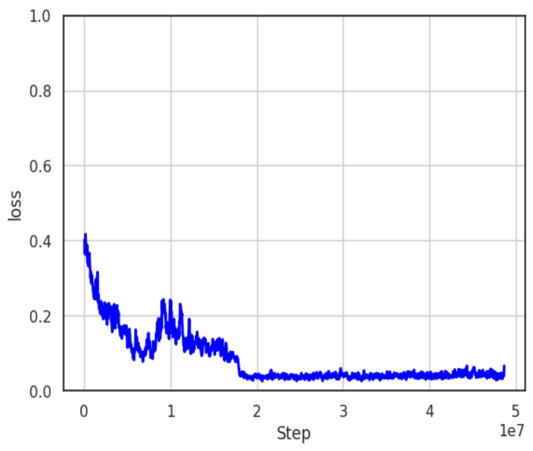
$$\theta_{k+1} \leftarrow (1 - \tau)\theta_k + \tau \nabla_{\theta_k} J(\theta_k) \tag{4.6}$$



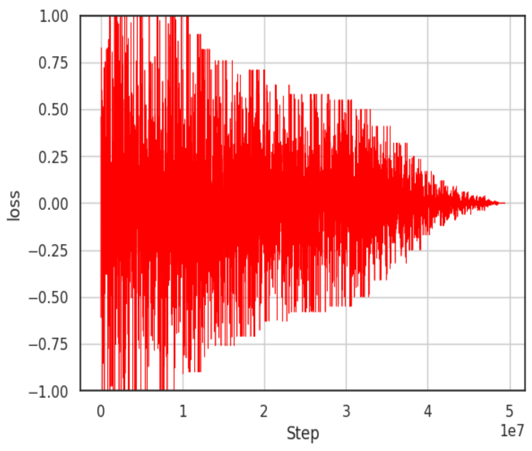
(a) Before optimization tricks. **(b)** After optimization tricks.

Figure 4.5: Gradient norm curve of Value-based Policies.

The clipping upper bound u_{clip} has been searched within the set $\{1,3,5,10\}$, while the τ hyper-parameter within $\{10^{-3}, 5 \cdot 10^{-4}, 10^{-4}, 5 \cdot 10^{-5}\}$. The best configuration has been $u_{clip} = 1$ and $\tau = 5 \cdot 10^{-4}$.

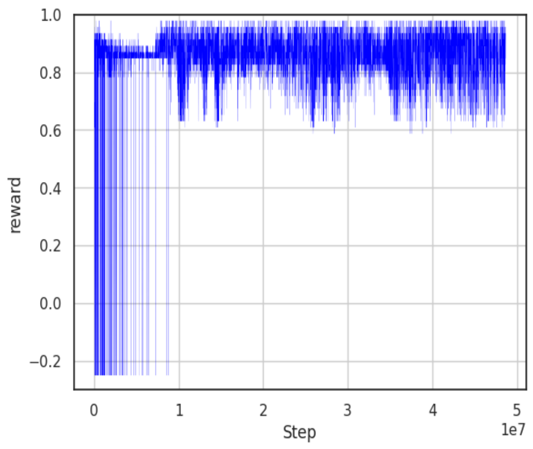


(a) HDQN's Policy Loss.

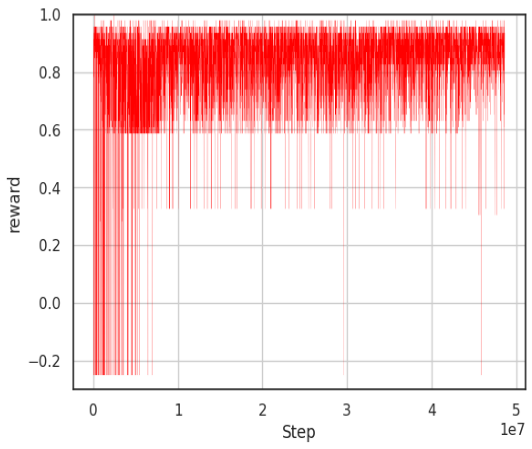


(b) HA2C Actor's Policy Loss.

Figure 4.6: Policy losses when using single reward.



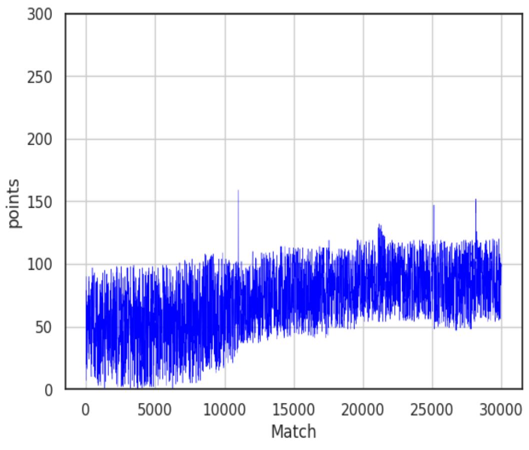
(a) HDQN Reward.



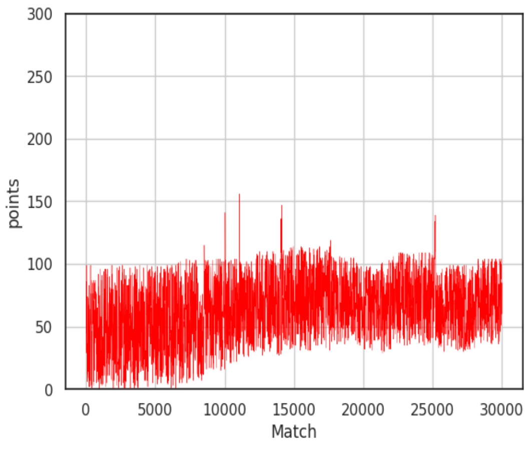
(b) HA2C Reward.

Figure 4.7: Reward when using single reward.

Both architectures have converged and scored way more points with respect to the previous configuration, increasing both the inferior bound of minimum points and the upper bound of maximum points gained.
In Figure 4.9, it can be noticed how the units of both players are all positioned



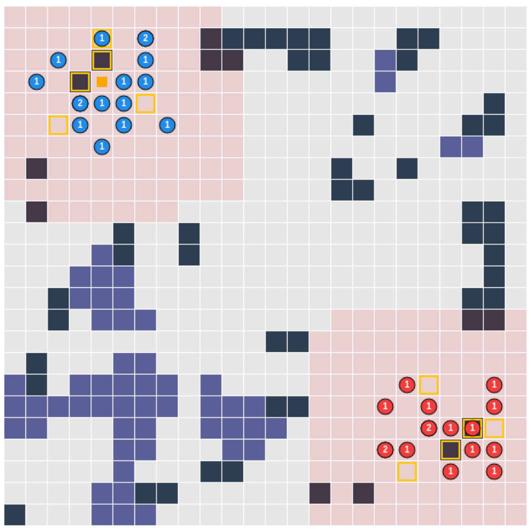
(a) HDQN Points.



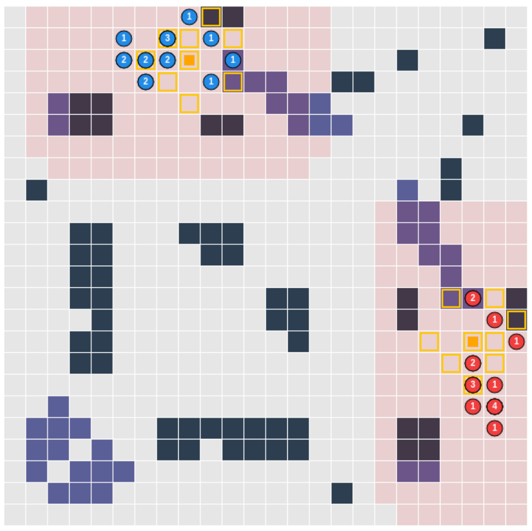
(b) HA2C Points.

Figure 4.8: Points collected when using single reward.

around the relics sites. This is a greedy behavior developed by the single reward $\mathcal{R}_{single}$, which encourages the closeness to the first relic discovered: however, in games where more than two relics sites can be present, this behavior may be limiting, as only one source of points gathering will be considered, i.e. the closest to spawn corner.



(a)



(b)

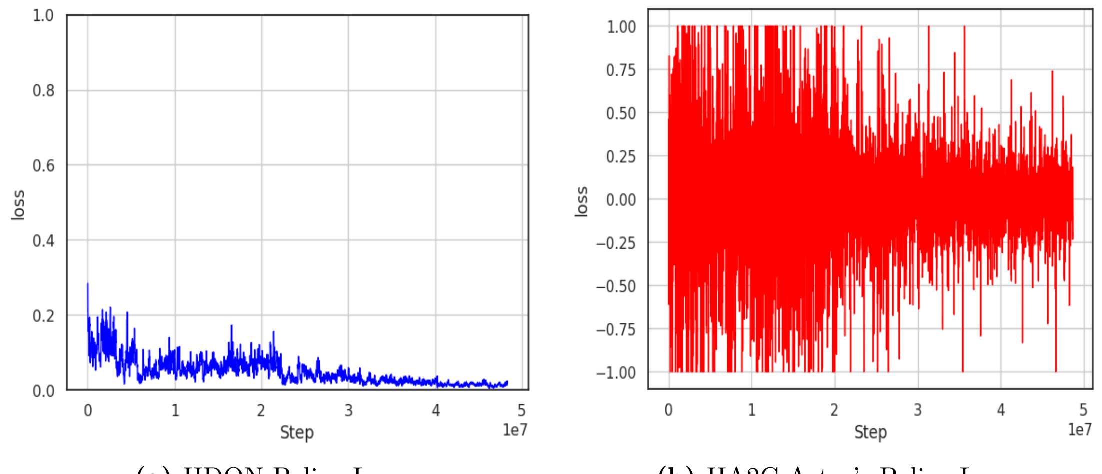
Figure 4.9: Greedy behaviors due to single reward.

4.4 Mixed reward

Despite of the higher results of single reward contribution, the units still acts very greedy. As matter of fact, once a relic site is discovered, the only purpose becomes to approach it, stopping any further exploration and any eventual discovery of new relic sites. In order to contain this selfish behavior and stimulate the exploration, the contribution of $\mathcal{R}_{global}$ has been added to $\mathcal{R}_{single}$ to verify if the units can be reactive and organize themselves in more cooperative strategies.

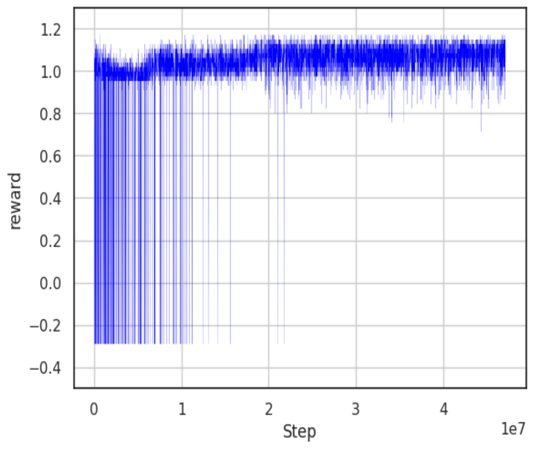
$$\mathcal{R}_{mixed} = \mathcal{R}_{single} + \mathcal{R}_{global} \tag{4.7}$$

The credit assignment is now more complete and aims to stimulate a cooperation between agents, building a hierarchy of the tasks: the most important one still remains to collect points staying closer to relics but, in second grade, a spread distribution of units on the map is evaluated more in the beginning of a match, contributing to an earlier exploration before pursuing the greedy task.
However, due to the complexity, lesser density and higher variance of the reward, HA2C has not converged as in Figure 4.10, even after $\sim$ 22 hours of training. The effects can be observed as well in Figure 4.12, as the the variance of points is way more higher than the ones achieved in Figure 4.8 and maximum number of points achieved didn't improve too.
Recalling the comparative analysis of [50], it doesn't surprise. In fact, On-Policy algorithms need extended periods to converge on highly rewarding strategies and the different trends in Figures 4.2 4.6 4.10 demonstrate how more unstable they are with respect to Off-Policy DQN when the computational time is limited.

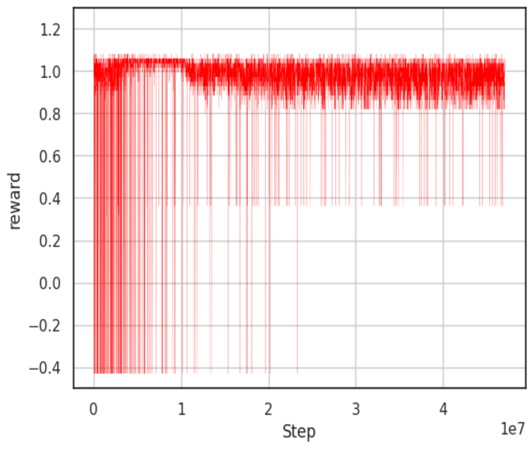


(a) HDQN Policy Loss. **(b)** HA2C Actor's Policy Loss.

Figure 4.10: Policy losses when using mixed reward.

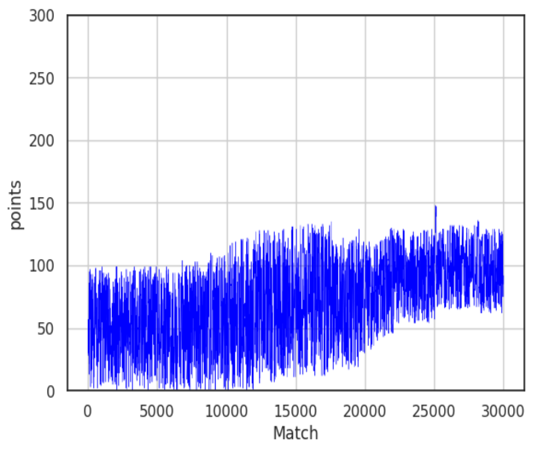


(a) HDQN Reward.

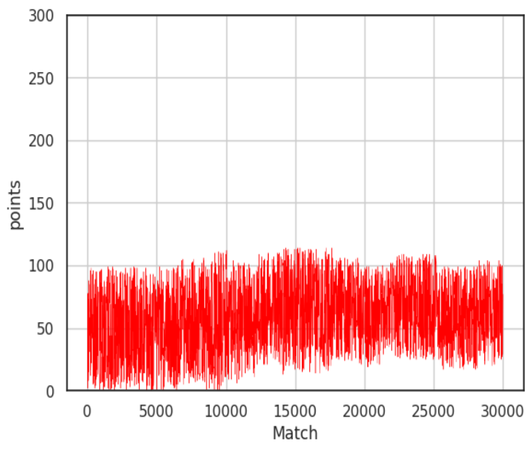


(b) HA2C Reward.

Figure 4.11: Reward when using mixed reward.



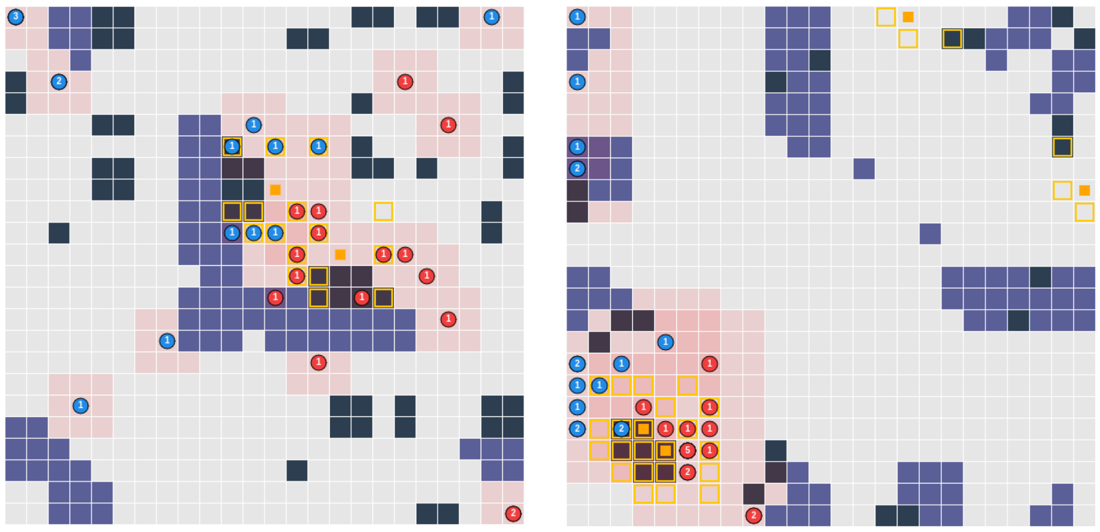
(a) HDQN Points.



(b) HA2C Points.

Figure 4.12: Reward when using mixed reward.

In contrast, HDQN has introduced a slight exploring technique while maintaining the purpose of collecting points. However, this has not been a systematic behavior, since the policy was not able to generalize enough in all environment set-ups. In part A. of Figure 4.13, some units are close to the relic site while other units try to explore the environment; in part B. of Figure 4.13, instead, the units focus only on gathering points from the first relic discovered, i.e. bottom left corner, and totally missed the other two relic sites in the opposite corner because they didn't explore enough.


(a) Greedy and exploration behaviors combined.

(b) Greedy behavior only.

Figure 4.13: Comparison of greedy and cooperative behaviors of HDQN when using mixed reward.

4.5 Final ranking

The Table 4.3 sums up the results obtained in the previous sections, considering the 10 games of validation phase. The points have been rounded to upper integer, since the score cannot be a floating point. The best results for each algorithm has been highlighted.

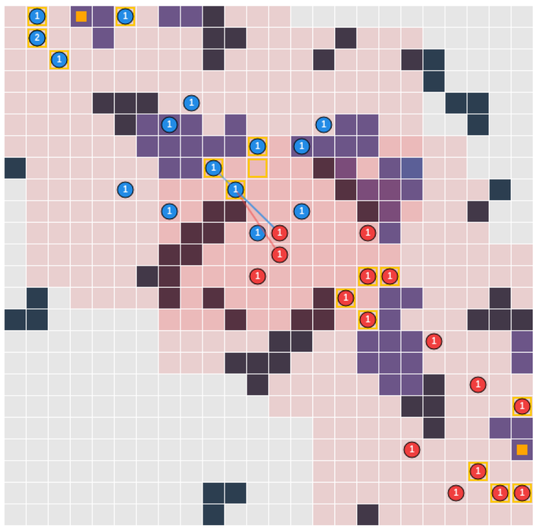
Algorithm	Reward	Avg Points	Highest ranking score
HDQN	Global	8 ± 5	*N.S.*
HA2C	Global	9 ± 5	*N.S.*
HDQN	Single	84 ± 27	$381^{th}/704$
HA2C	Single	76 ± 29	$460^{th}/704$
HDQN	Mixed	90 ± 33	$341^{th}/704$
HA2C	Mixed	66 ± 41	$544^{th}/704$

Table 4.3: Results overview.

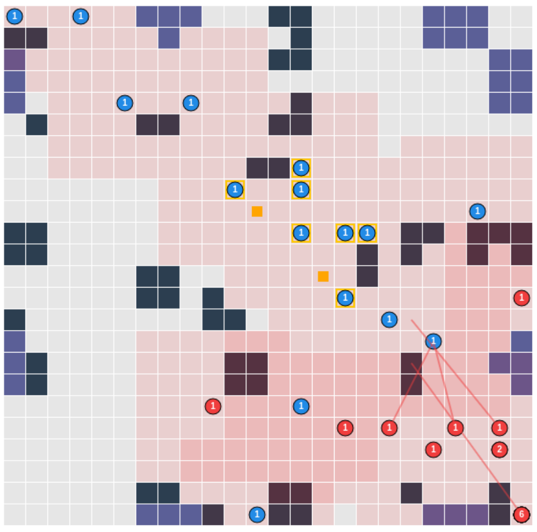
N.S. stands for "Not Submitted".

LuxAI S3 and LuxAI past seasons had never had any kind of restriction regarding the implementation of the agent. After the ending of each competition, the top 10

players are suggested to publish their solution, in order to encourage new way of developing solution for future seasons of the competition. For instance, looking at the leaderboard of LuxAI S2[3], the top 10 players used mostly a rule-based approach, using tailored crafted heuristics to optimize the agent workflow. In particular, only one player used A2C-based algorithm (4^{th}) and only one player used Imitation Learning (7^{th}).
Since the solution are not available yet for LuxAI S3, it's not possible to know if the winning solutions are rule-based or RL-based. However, looking at some snapshots of the top tier players' games in Figure 4.14, it can be noticed how the units know exactly where yellow border relic tiles are and occupy them for the majority of the time, as they have some heuristics which evaluate the map more precisely. Moreover, after the occupation of the relic tiles, the remaining units not only explore the map but also engage multiple "combats" with opponent' units through the sap action, as they are battling for the territory control. This behavior has never been seen in the thesis experiments and it would have been very difficult to implement such strategy with RL-based algorithms, as in Self-play training the contact happens rarely since the primary tasks are collection of points and exploration of territory.



(a) Game from 1^{st} player.



(b) Game from 2^{nd} player..

Figure 4.14: Games from top tier players.

[3]https://www.kaggle.com/competitions/lux-ai-season-2/leaderboard

Chapter 5

Conclusions

In this thesis, we explored the LuxAI S3 competition, organized in conjunction with NeurIPS conference, as a platform for advancing the development of Multi-Agent Reinforcement Learning (MARL) in real-world problem-solving scenarios. The competition, through its focus on multi-agent systems, provided a structured and dynamic environment where real-time decision making system and development of collaborative strategies were the main pillars. The core aim of this work was to highlight the immense potential of MARL in addressing complex competitive games, giving a solid alternative to traditional solutions based on human-crafted heuristics.

The major challenges of MARL, specifically the non-stationarity of the environment and the credit assignment problem, are the reasons for which MARL algorithms require a huge number of samples for effective training and, at the same time, don't guarantee convergence in every scenarios; hence the need of a precise formalization of the problem and tailored solution in order to foster effectively the scalability and the computational efficiency during the learning.
Through the combination of Homogeneity property and state-of-the-art algorithm DQN and A2C, the proposed architecture has been able to train an autonomous agent aware of environment evolution and agents' dynamics, developing a strategy balanced between cooperative behaviors and greedy conducts to pursue the winning conditions. Moreover, the model has reached stability even though it was trained as agnostic and with Self-Play, thus without distilling from experts' knowledge and learning only from trial-and-error simulations. Despite not achieving top tier positions of final ranking, HDQN has still reached convergence in learning phase and positioned in the first half of the ranking.

As multi-agent systems become more prevalent in various sectors, such as transportation, logistics, finance, and healthcare, the ability of agents to learn, collaborate, and make decisions autonomously will become increasingly valuable. Furthermore, the continued exploration of multi-agent systems in decentralized settings, including edge computing and distributed networks, is poised to revolutionize industries where communication and coordination are essential. Since the proposed architecture is Model-Free, it can be shifted to other domains or future LuxAI competitions, with the amendments thereto. Future works may include the combination of Homogeneity concept with other MARL algorithms, in order to create a more comprehensive and wider benchmark of its potentialities.

Bibliography

[1] A. L. Samuel. «Some Studies in Machine Learning Using the Game of Checkers». In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: `10.1147/rd.33.0210` (cit. on p. 3).

[2] D.P. Kroese, Z.I. Botev, T. Taimre, and R. Vaisman. *Data Science and Machine Learning: Mathematical and Statistical Methods.* Chapman & Hall/CRC machine learning & pattern recognition. Boca Raton: CRC Press, 2019, pp. 19–20, 67–120. ISBN: 9781138492530. URL: `https://people.smp.uq.edu.au/DirkKroese/DSML/` (cit. on p. 3).

[3] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press, 2014, pp. 33–34. ISBN: 978-1-10-705713-5 (cit. on pp. 4, 5).

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html` (cit. on pp. 6, 8, 10, 13, 23, 32).

[5] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014 (cit. on p. 8).

[6] Richard Bellman. *Dynamic Programming.* Dover Publications, 1957. ISBN: 9780486428093 (cit. on pp. 9, 11).

[7] Christopher Watkins and Peter Dayan. «Q-learning». In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: `10.1007/BF00992698` (cit. on p. 14).

[8] Yoshua Bengio. «Learning Deep Architectures for AI.» In: *Foundations and Trends in Machine Learning* 2.1 (2009), pp. 1–127. URL: `http://dblp.uni-trier.de/db/journals/ftml/ftml2.html#Bengio09` (cit. on p. 15).

[9] K. Hornik, M. Stinchcombe, and H. White. «Multilayer feedforward networks are universal approximators». In: *Neural Netw.* 2.5 (July 1989), pp. 359–366. ISSN: 0893-6080 (cit. on p. 16).

[10] F. Rosenblatt. *Principles of Neurodynamics.* Spartan Books, 1959 (cit. on p. 17).

[11] Thao Nguyen, Maithra Raghu, and Simon Kornblith. *Do Wide and Deep Networks Learn the Same Things? Uncovering How Neural Network Representations Vary with Width and Depth.* 2021. arXiv: `2010.15327 [cs.LG]`. URL: `https://arxiv.org/abs/2010.15327` (cit. on pp. 17, 51).

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. «Backpropagation Applied to Handwritten Zip Code Recognition». In: *Neural Computation* 1 (1989), pp. 541–551 (cit. on p. 18).

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition.* 2015. arXiv: `1512.03385 [cs.CV]`. URL: `https://arxiv.org/abs/1512.03385` (cit. on p. 19).

[14] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* 2017. arXiv: `1609.04747 [cs.LG]`. URL: `https://arxiv.org/abs/1609.04747` (cit. on p. 21).

[15] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980` (cit. on pp. 21, 22).

[16] Aditya Mohan, Amy Zhang, and Marius Lindauer. «Structure in Deep Reinforcement Learning: A Survey and Open Problems». In: *Journal of Artificial Intelligence Research* 79 (Apr. 2024), pp. 1167–1236. ISSN: 1076-9757. DOI: `10.1613/jair.1.15703`. URL: `http://dx.doi.org/10.1613/jair.1.15703` (cit. on p. 22).

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: `1312.5602 [cs.LG]`. URL: `https://arxiv.org/abs/1312.5602` (cit. on p. 24).

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning.* 2016. arXiv: `1602.01783 [cs.LG]`. URL: `https://arxiv.org/abs/1602.01783` (cit. on pp. 24, 51).

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms.* 2017. arXiv: `1707.06347 [cs.LG]`. URL: `https://arxiv.org/abs/1707.06347` (cit. on p. 26).

[20] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization.* 2017. arXiv: `1502.05477 [cs.LG]`. URL: `https://arxiv.org/abs/1502.05477` (cit. on p. 26).

[21] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches.* MIT Press, 2024. URL: `https://www.marl-book.com` (cit. on pp. 28, 29, 31, 33, 35).

[22] Kun Il Park. *Fundamentals of Probability and Stochastic Processes with Applications to Communications.* 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 3319680749 (cit. on p. 28).

[23] Annie Wong, Thomas Bäck, Anna V Kononova, and Aske Plaat. «Deep multiagent reinforcement learning: challenges and directions». en. In: *Artif. Intell. Rev.* (Oct. 2022) (cit. on p. 30).

[24] Yang Yu. «Towards Sample Efficient Reinforcement Learning». In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18.* International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 5739–5743. DOI: `10.24963/ijcai.2018/820`. URL: `https://doi.org/10.24963/ijcai.2018/820` (cit. on p. 31).

[25] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. *Towards Continual Reinforcement Learning: A Review and Perspectives.* 2022. arXiv: `2012.13490 [cs.LG]`. URL: `https://arxiv.org/abs/2012.13490` (cit. on p. 31).

[26] Changxi Zhu, Mehdi Dastani, and Shihan Wang. *A Survey of Multi-Agent Deep Reinforcement Learning with Communication.* 2024. arXiv: `2203.08975 [cs.MA]`. URL: `https://arxiv.org/abs/2203.08975` (cit. on p. 31).

[27] Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, Olivier Pietquin, and Laura Toni. *A Survey of Temporal Credit Assignment in Deep Reinforcement Learning.* 2024. arXiv: `2312.01072 [cs.LG]`. URL: `https://arxiv.org/abs/2312.01072` (cit. on pp. 32, 47).

[28] Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. *Learning Implicit Credit Assignment for Cooperative Multi-Agent Reinforcement Learning.* 2020. arXiv: `2007.02529 [cs.LG]`. URL: `https://arxiv.org/abs/2007.02529` (cit. on p. 32).

[29] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. *Multiagent Cooperation and Competition with Deep Reinforcement Learning.* 2015. arXiv: `1511.08779 [cs.AI]`. URL: `https://arxiv.org/abs/1511.08779` (cit. on pp. 34, 37, 48).

[30] Gregory Palmer, Rahul Savani, and Karl Tuyls. *Negative Update Intervals in Deep Multi-Agent Reinforcement Learning.* 2019. arXiv: `1809.05096 [cs.MA]`. URL: `https://arxiv.org/abs/1809.05096` (cit. on p. 34).

[31] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments.* 2020. arXiv: 1706.02275 [cs.LG]. URL: https://arxiv.org/abs/1706.02275 (cit. on pp. 35, 38, 48).

[32] Hanmo Chen, Stone Tao, Jiaxin Chen, Weihan Shen, Xihui Li, Chenghui Yu, Sikai Cheng, Xiaolong Zhu, and Xiu Li. *Emergent collective intelligence from massive-agent cooperation and competition.* 2023. arXiv: 2301.01609 [cs.AI]. URL: https://arxiv.org/abs/2301.01609 (cit. on p. 37).

[33] Roger Creus Castanyer. *Centralized control for multi-agent RL in a complex Real-Time-Strategy game.* 2023. arXiv: 2304.13004 [cs.AI]. URL: https://arxiv.org/abs/2304.13004 (cit. on p. 37).

[34] Stone Tao, Akarsh Kumar, Bovard Doerschuk-Tiberi, Isabelle Pan, Addison Howard, and Hao Su. «Lux AI Season 3: Multi-Agent Meta Learning at Scale». In: *NeurIPS 2024 Competition Track.* 2024. URL: https://openreview.net/forum?id=7t8kWYbOcj (cit. on pp. 37, 39).

[35] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* 2017. arXiv: 1712.01815 [cs.AI]. URL: https://arxiv.org/abs/1712.01815 (cit. on pp. 37, 51).

[36] Julian Schrittwieser et al. «Mastering Atari, Go, chess and shogi by planning with a learned model». In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://dx.doi.org/10.1038/s41586-020-03051-4 (cit. on pp. 37, 51).

[37] Ti-Rong Wu, Hung Guei, Pei-Chiun Peng, Po-Wei Huang, Ting Han Wei, Chung-Chin Shih, and Yun-Jui Tsai. *MiniZero: Comparative Analysis of AlphaZero and MuZero on Go, Othello, and Atari Games.* 2024. arXiv: 2310.11305 [cs.AI]. URL: https://arxiv.org/abs/2310.11305 (cit. on p. 37).

[38] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning.* 2019. arXiv: 1912.06680 [cs.LG]. URL: https://arxiv.org/abs/1912.06680 (cit. on p. 37).

[39] Michaël Mathieu et al. *AlphaStar Unplugged: Large-Scale Offline Reinforcement Learning.* 2023. arXiv: 2308.03526 [cs.LG]. URL: https://arxiv.org/abs/2308.03526 (cit. on p. 37).

[40] Christopher Amato. *An Introduction to Centralized Training for Decentralized Execution in Cooperative Multi-Agent Reinforcement Learning.* 2024. arXiv: 2409.03052 [cs.LG]. URL: https://arxiv.org/abs/2409.03052 (cit. on p. 38).

[41] Tianshu Chu, Jie Wang, Lara Codecà, and Zhaojian Li. *Multi-Agent Deep Reinforcement Learning for Large-scale Traffic Signal Control.* 2019. arXiv: `1903.04527 [cs.LG]`. URL: `https://arxiv.org/abs/1903.04527` (cit. on p. 38).

[42] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. *The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games.* 2022. arXiv: `2103.01955 [cs.LG]`. URL: `https://arxiv.org/abs/2103.01955` (cit. on p. 38).

[43] Maryam Zare, Parham M. Kebria, Abbas Khosravi, and Saeid Nahavandi. *A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges.* 2023. arXiv: `2309.02473 [cs.LG]`. URL: `https://arxiv.org/abs/2309.02473` (cit. on p. 38).

[44] Chacha Chen, Hua Wei, Nan Xu, Guanjie Zheng, Ming Yang, Yuanhao Xiong, Kai Xu, and Zhenhui Li. «Toward A Thousand Lights: Decentralized Deep Reinforcement Learning for Large-Scale Traffic Signal Control». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 3414–3421. DOI: `10.1609/aaai.v34i04.5744`. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/5744` (cit. on pp. 38, 47).

[45] Hua Wei et al. «CoLight: Learning Network-level Cooperation for Traffic Signal Control». In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management.* CIKM '19. ACM, Nov. 2019, pp. 1913–1922. DOI: `10.1145/3357384.3357902`. URL: `http://dx.doi.org/10.1145/3357384.3357902` (cit. on p. 38).

[46] Deheng Ye et al. *Towards Playing Full MOBA Games with Deep Reinforcement Learning.* 2020. arXiv: `2011.12692 [cs.AI]`. URL: `https://arxiv.org/abs/2011.12692` (cit. on p. 38).

[47] Boyin Liu, Zhiqiang Pu, Yi Pan, Jianqiang Yi, Yanyan Liang, and D. Zhang. «Lazy Agents: A New Perspective on Solving Sparse Reward Problem in Multi-agent Reinforcement Learning». In: *Proceedings of the 40th International Conference on Machine Learning.* Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, pp. 21937–21950. URL: `https://proceedings.mlr.press/v202/liu23ac.html` (cit. on pp. 38, 47).

[48] Zijian Gao, Kele Xu, Bo Ding, Huaimin Wang, Yiying Li, and Hongda Jia. *KnowSR: Knowledge Sharing among Homogeneous Agents in Multi-agent Reinforcement Learning.* 2021. arXiv: `2105.11611 [cs.AI]`. URL: `https://arxiv.org/abs/2105.11611` (cit. on p. 38).

[49] Ralf Herbrich, Tom Minka, and Thore Graepel. «TrueSkill™: A Bayesian Skill Rating System». In: *Advances in Neural Information Processing Systems*. Ed. by B. Schölkopf, J. Platt, and T. Hoffman. Vol. 19. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper_files/paper/2006/file/f44ee263952e65b3610b8ba51229d1f9-Paper.pdf (cit. on p. 40).

[50] Neil De La Fuente and Daniel A. Vidal Guerra. *A Comparative Study of Deep Reinforcement Learning Models: DQN vs PPO vs A2C*. 2024. arXiv: 2407.14151 [cs.LG]. URL: https://arxiv.org/abs/2407.14151 (cit. on pp. 48, 64).

[51] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. «RLlib: Abstractions for Distributed Reinforcement Learning». In: *International Conference on Machine Learning (ICML)*. 2018. URL: https://arxiv.org/pdf/1712.09381 (cit. on p. 54).

[52] Zhanghao Wu, Eric Liang, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. «RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem». In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2021. URL: https://proceedings.neurips.cc/paper/2021/file/2bce32ed409f5ebcee2a7b417ad9beed-Paper.pdf (cit. on p. 54).