ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

NEIGHBORING-BASED STRATEGIES FOR MULTI-AGENT REINFORCEMENT LEARNING

Elaborato in
ADVANCED SOFTWARE MODELLING AND DESIGN

Relatore
Prof. MIRKO VIROLI

Correlatore
Dott. GIANLUCA AGUZZI
Dott. DAVIDE DOMINI

Presentata da
NICOLÒ MALUCELLI

Anno Accademico 2023 – 2024

Abstract

Multi-Agent Reinforcement Learning introduces many new challenges to the single agent scenario, such as non-stationarity, scalability, partial observability, and credit assignment. While centralized training methods help address some of these problems, mitigating partial observability and non-stationarity, and facilitating credit assignment, they are affected by scalability issues as the number of agents increases.

Centralized training methods are by far the most used and studied. However, they are not always a feasible solution in real-world scenarios, especially due to the potential limitations imposed by the structure of the agent network. On the other hand, decentralized training methods received less attention, but their potential in real-case scenarios is high.

This thesis investigates different neighbor-based decentralized training strategies, proving that they can represent a valid alternative to the centralized training approach. Various distributed training methods, such as Experience Sharing, NN-Averaging, and NN-Consensus, are evaluated within a custom environment and compared against the centralized training scheme, in order to assess their efficiency and scalability.

to my family and friends

Contents

Introduction

The rapid progress in artificial intelligence highlighted the significant potential of Reinforcement Learning (RL) as a technique for empowering agents to make autonomous decisions in dynamic environments. Agents can learn optimal strategies through a trial-and-error approach by interacting with the environment and receiving feedback for their actions [28].

While single-agent reinforcement learning has been extensively studied and applied across various domains, the complexity increases when multiple agents are involved: in Multi-Agent Reinforcement Learning (MARL), agents must take into account the presence of other agents when solving a task; depending on the type of the environment, agents may collaborate or compete with the other agents, introducing new challenges not present in the single agent scenario, such as non-stationarity, scalability, partial observability, and credit assignment [3].

To address these challenges, many training schemes have been explored by researchers. The main approach consists in training agents using a centralized setup, where a central node is responsible to handle the policy of each agent. This approach allows dealing with many of the problems above cited, but it is particularly subject to scalability issues: as the number of the agents increases, the central node becomes more and more a weakness, making this approach not feasible in some large-scale, real-world scenarios.

On the other hand, distributed training schemes, where each agent independently learns its policy, offer better scalability and robustness, but are subject to other problems, such as the non-stationarity of the environment, and the lack of coordination among agents.

This thesis focuses on comparing the effectiveness of different distributed learning strategies with the goal to offer a valid solution to the centralized training approach, exploring different solutions to identify the most effective way to address the main challenges of MARL. Along with the theoretical analysis, a prototype has been developed to implement and evaluate these distributed learning strategies in practice, providing a more robust foundation for addressing the main challenges of MARL.

The thesis begins by introducing the foundational concepts of Reinforcement Learning and Multi-Agent Reinforcement Learning, providing a solid theory that allows to better understand the following more practical chapters. An overview of the main RL algorithms is provided and the challenges of MARL are described, along with the possible training schemes.

The second chapter introduces the frameworks and the technologies that have been employed in the test section. In particular it discusses Gymnasium, an open-source library for the definition of custom environment, and RLLib, a powerful reinforcement learning library.

The third chapter outlines the primary contribution of this thesis, describing the custom environment in which the experiments have been conducted, and how the different approaches have been evaluated and compared.

Chapter 4 presents the results of these experiments, offering an evaluation of the performance of the different approaches, focusing on key metrics such as learning efficiency, scalability, and communication overhead.

Chapter 1

Background

The first section of this chapter explains what reinforcement learning is and which is the idea behind two mainstream algorithms: Deep Q-learning (DQN) and Proximal Policy Optimization (PPO). The second section introduces the multi-agent case instead, along with the challenges of this formulation.

1.1 Reinforcement Learning

Reinforcement learning is a subfield of machine learning in which an agent learns by itself the correct way to act in an environment, to fulfill the given task through a trial-and-error approach [12].

In the context of reinforcement learning, the term agent refers to a generic entity that is able to interact with the surrounding environment, make decisions and perform actions in order to achieve a specific goal.

Reinforcement learning differs from classic approaches of programming agents, such as rule-based systems, since the programmer does not code the behaviour of the agent (i.e., which action to take in which situation), but rather defines a reward system. In reinforcement learning, the agent learns to perform tasks by interacting with the environment and receiving feedback in the form of rewards or penalties. This feedback helps the agent to develop a strategy that maximizes the cumulative reward over time.

A reinforcement learning problem can be described by a Markov Decision Process (MDP) [13], that is a tuple $\langle S, A, f, p \rangle$, where:

- S represents the finite set of the environment states;
- A represents the finite set of actions that the agent can perform while in any state $s \in S$;

- $f : S, A, S \to [0, 1]$ represents the transition probability function, which describes the probability of reaching a state $s_1 \in S$ by performing an action $a \in A$ while in the state $s_0 \in S$. In the scenario of a fully deterministic environment, since for each pair *state-action* there can be only an arrival state, the transition probability function can be simplified in such a way: $f : S, A \to S$;

- $p : S, A, S \to \mathbb{R}$ represents the agent's reward function, which maps each transition to a numerical value. In the same way of the transition probability function *f*, *p* can be simplified to $p : S, A \to \mathbb{R}$ in case of fully deterministic scenarios.

As shown in the image below (1.1), at a generic time instant t, the environment is in a state $s_t \in S$ and the agent performs an action $a_t \in A$, possibly causing a change in the environment state $s_{t+1} \in S$ according to the transition probability function *f*, and receiving a reward $r_{k+1} = p(s_t, a_t, s_{t+1})$.

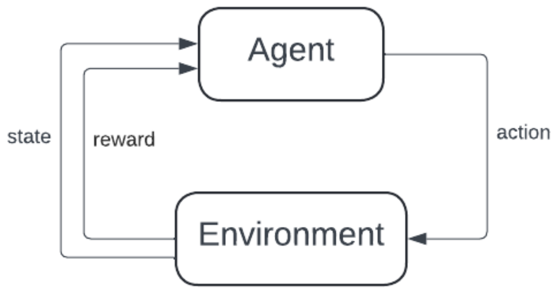


Figure 1.1: Graphical representation of the interactions between agent and environment

The goal of the agent is to maximize the cumulative discounted reward given nothing but the immediate feedback for its actions (i.e., reward and new environment state). This is a significant problem because in many real-world scenarios, since the consequences of an action may not be immediately apparent, making it difficult for an agent to learn the optimal strategy. This problem, known by the name of *delayed reward*, is a crucial challenge in the field of reinforcement learning [12].

Exploration vs exploitation In contrast to supervised learning, in reinforcement learning the agent must explore the environment by itself in order to gather more information about the surrounding environment and possibly discover new advantageous strategies. However, the exploration component must be balanced with exploitation, where the agent utilizes the knowledge it has already gained to optimize the performance based on the current understanding.

Different strategies can be adopted to balance this trade-off. One of the most effective and common approaches is the ϵ-greedy strategy where, at each step, the agent generates a random number between 0 and 1. If the number is less than ϵ, the agent picks a random action between the possible ones (exploration), otherwise the agent chooses the action having the highest expected reward (exploitation). A common strategy involves decreasing the value of ϵ while the agent learns, in order to encourage the agent to explore the environment more when the knowledge is low, and preferring exploitation as the agent gains more confidence. The effectiveness of the ϵ-greedy approach, compared to a completely greedy approach ($\epsilon = 0$) has been proved by Sutton and Barto in [28].

Another approach is the softmax approach. In this method, a probability is assigned to each action based on how good they are, prioritizing actions with a higher expected reward, while still allowing for exploration of more uncertain options [28].

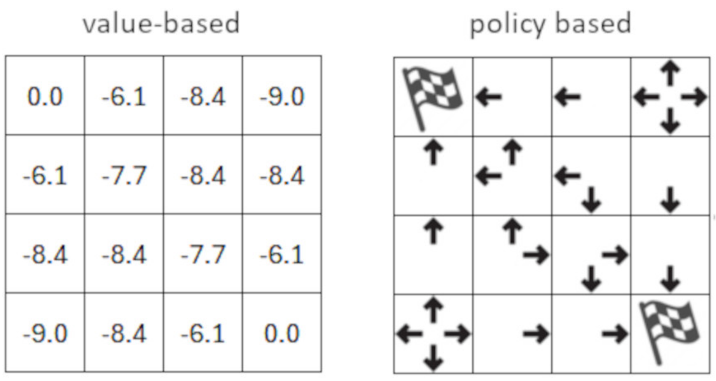
1.1.1 Overview of the main algorithms

Reinforcement learning algorithms can be classified into two main classes: value-based, and policy-based, also known as critic-based, and actor-based methods (image 1.2).

Value-based algorithms, such as Q-Learning [32], SARSA [24], and their variants, use a function to estimate how good a state, or a pair state-action, is. This function, called the value function, is then used to make decisions about which action the agent should take in a given state. Each value takes into account not only the immediate reward received for reaching that state, but also the future rewards that the agent may receive from performing actions in that state. This formulation allows to address the *delayed reward* problem previously described.

While value-based methods use a value function to estimate states' values, policy-based methods, such as REINFORCE [33], TRPO [26], and PPO [27], rely on a function, called policy, which maps each state to an action. The agent learns the optimal behaviour by directly optimizing the policy function, and adjusting the parameters in order to maximize the expected reward.

Both critic-based and actor-based methods have their strengths and weaknesses; this led to the development of a new class of algorithms, known as actor-critic methods, which aims to combine the aspects of both methods [14].



0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

Figure 1.2: Value-based vs Policy-based approaches

Q-learning Q-learning is a value-based and model-free reinforcement learning algorithm that relies its operations on the Q-table. The Q-table maps each pair state-action to a value representing the expected discounted reward obtainable by the agent performing that action while in that state.

The agent learns the optimal behaviour by iteratively updating the Q-values depending on the agent's experience using the Bellman equation:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma max_{a'}Q(s',a')) \tag{1.1}$$

where:

- α is the learning rate, a hyperparameter that determines how much the new experience will override the current value of $Q(s,a)$. Normally α is set higher at the beginning of the training process and lowered as the agent gets better.

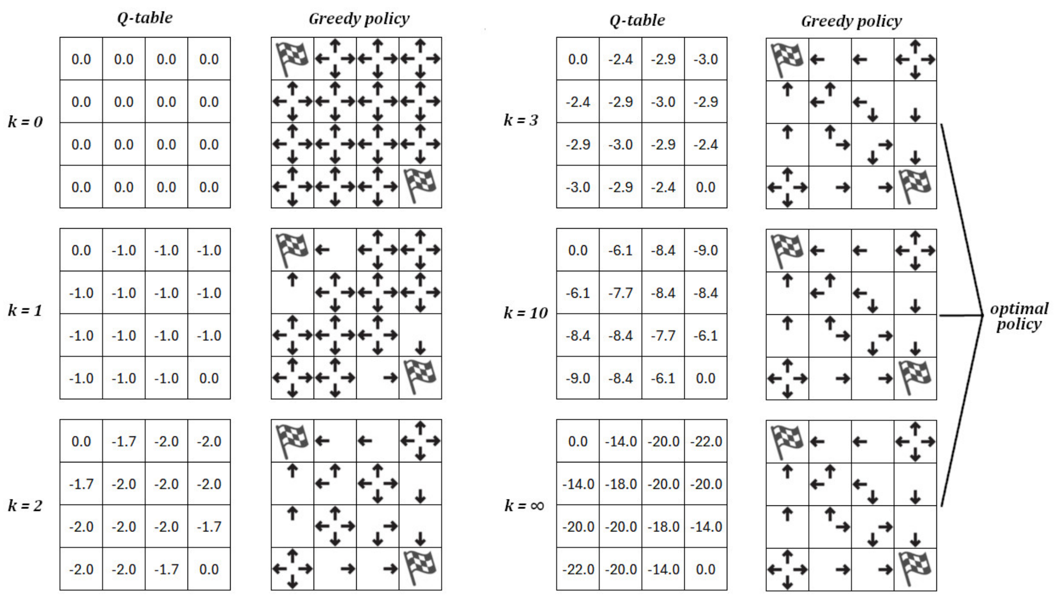
- $r = p(s,a,s')$ represents the immediate reward received by the agent as a consequence of having performed the action a while in the state s

- $s' = f(s,a)$ is the new environment's state

- $max_{a'}Q(s',a')$ is the highest reward that the agent can obtain while in the state *s'*. This is the element that allows the iterative evaluation of the Q-table, and is fundamental to predict the discounted reward in the given state.

- γ is the discount factor, a hyperparameter that determines the trade-off between immediate and future rewards. When the value of γ is 0, only the immediate reward is considered when computing the Q-table, while when γ is 1, all the rewards have the same importance, no matter how far in the future they are. Choosing one of the extremes is normally not recommended: $\gamma = 0$ does not allow the agent to develop complex strategies since only the present is considered, while with $\gamma = 1$ the problem may never converge to a solution. For this reason, γ is normally chosen between 0.90 and 0.99.

It has been proved by Watkins in [32] that this formulation always converges to an optimal solution when the number of episodes from each starting state and action is infinite. However, in order to get an optimal policy, an optimal Q-table is not necessary: as shown in the image 1.3, a sub-optimal Q-table can produce the same policy as the optimal Q-table. The difficulty, then, is to recognize when the optimal policy has been reached.



k = 0

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

k = 1

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

k = 2

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

k = 3

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

k = 10

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

k = ∞

0.0	-14.0	-20.0	-22.0
-14.0	-18.0	-20.0	-20.0
-20.0	-20.0	-18.0	-14.0
-22.0	-20.0	-14.0	0.0

Figure 1.3: Proof that a sub-optimal Q-table can produce the same policy as the optimal Q-table

The reason why this happens is related to how the policy π is computed:

$$\pi(s) = argmax_a Q(s, a)$$

Thus, two different Q-tables, Q and Q', may produce the same policy if:

$$argmax_a Q(s, a) = argmax_a Q'(s, a) \quad \forall\ s \in S$$

Q-learning: limitations and solutions Q-learning adopts a table as value function; therefore, the number of values to store directly depends on the number of states and the number of actions in each state. In many real-world problems, Q-learning is not an admissible solution, because too many states and too many actions, lead to the so-called *state space explosion*, making it impossible to memorize all the values. Even imagining having infinite space on our device, exploring all the pairs state-action would take too much time, making the learning inefficient.

Deep Q-learning (DQN) [18] has been designed to overcome this limitation, allowing to approximate the Q-table through a deep neural network. Mnih et al. [17] proved that DQN can be fed using high-dimensional input such as RGB images, outperforming previous approaches.

However, the use of a deep neural comes with some drawbacks, that if not properly addressed may render the training unstable. The first problem is the correlation between the sample of the same episode, while the second is the correlation between the value to update and the target value, since the target value directly depends on the same neural network being optimized, as described by the Bellman equation in 1.1.

To solve the first problem, researchers introduced the idea of *experience replay*: instead of directly training the neural network on the trajectory data, each experience is before stored in a randomized buffer, called *replay buffer*. This allows to remove the correlation between the samples forming a batch, increasing the performance and facilitating the convergence.

Different types of buffers can be used, but one of the most common is the *prioritized buffer*, presented by Schaul et al. in [25]. The core idea of *prioritized experience replay* is to give more importance to the samples which may lead to faster learning, instead of treating all the samples in the same way.

To overcome the second problem, the concept of *delayed update* has been introduced: during the training two distinct networks are used, the main one and an auxiliary target network. The target network is used to compute the target value and is not updated in real-time like the main network, but only at a frequency C previously defined. Therefore, the loss function becomes:

$$L_i(\theta_i) = [Q(s, a; \theta_i) - (r + \gamma max_{a'} Q(s', a'; \theta_i^-)]^2$$

Where θ_i represents the set of weights of the main network, while θ_i^- represents the weights of the target network. This strategy effectively resolves the moving target problem, allowing the main network to learn from a stable target.

Q-learning: additional enhancements Deep Q-learning performance can be further improved by adopting some additional expedients, such as Double Q-learning [9] and Dueling Q-learning [30].

- **Double Q-learning**: In the classic Q-learning, the update rule used to estimate the next state, always select the state having the highest value. This may lead to overoptimistic estimations due to the max operator's tendency to prefer higher, but potentially inaccurate, values.

  The main idea of Double Q-learning is to address this issue by adopting two value functions Q_A and Q_B, using one as target value of the other, in order to reduce the bias introduced by the max operator [9]:

  $$Q_A(s,a) = r + \gamma max_{a'} Q_B(s',a')$$
  $$Q_B(s,a) = r + \gamma max_{a'} Q_A(s',a')$$

  Since Q_A and Q_B are alternately trained on two different sets, the first one shouldn't suffer from the same positive bias as the second one and vice versa. Even though Double Q-learning has been initially designed for classic Q-learning, it can also be used in combination with Deep Q-learning [20].

- **Dueling Q-learning**: The core idea of Dueling Q-learning is to decompose the Q-value into two different components by using two different estimators: one for the state-value function and one for the advantage function, allowing to better generalize across actions.

  In particular, the proposed architecture is able to learn whether a state is good or not without having to learn the effect of each action for each state, making it especially useful in those situations in which actions do not affect the environment in a very relevant way [30].

Proximal Policy Optimization Unlike Q-learning and its variations, PPO is a policy-based method, therefore it directly optimizes the policy instead of estimating the value of state-action pairs. In particular, PPO belongs to the family of policy gradient methods along with REINFORCE [33] and Actor-Critic methods [14]. Generally, policy gradient methods use a loss function having this structure:

$$L^{PG}(\theta) = \hat{\mathbb{E}}[log \pi_\theta(a_t|s_t)\hat{A}_t]$$

Where π_θ is the stochastic policy, while $\hat{A}_t$ is the advantage function. However, this formulation can lead to very large policy updates which may destabilize the learning process.

PPO aims to guarantee the reliability typical of TRPO [26], while using a more simple and efficient optimizer. PPO utilizes a clipped objective function, which constrains the magnitude of policy updates in order to prevent drastic changes. Specifically, PPO maximizes the expected advantage while ensuring that the probability ratio between the new and the old policy stays within a given range defined by the hyperparameter ϵ:

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

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Due to their continuous nature, policy-based methods such as PPO perform much better in continuous action spaces if compared to DQN [27].

1.2 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) extends the concepts of single-agent reinforcement learning to environments in which multiple agents coexist, each one aiming to maximize its own reward.

Since more agents are involved, the base reinforcement learning formulation that relies on Markov Decision Process (MDP) is no longer suitable. The generalization of the MDPs to multi-agent scenarios is known as stochastic game [16].

In a similar way to MDP, a stochastic game consisting of n agents is defined as a tuple $\langle S, A_1, ..., A_n, f, p_1, ..., p_n\rangle$, where:

- S represents the finite set of the environment states, like in MDPs;
- A$_i$ represents the finite set of actions of the i-th agent. Different agents in the environment may have different set of actions;
- $f : S, A, S \rightarrow [0, 1]$ represents the transition probability function, which describes the probability of reaching a state $s_1 \in S$ by performing an action $a \in A$ while in the state $s_0 \in S$. Differently from the single-agent scenario, which utilizes the action performed by the only agent to compute the transition from a state s_0, in the multi-agent scenario, the result of the transition depends from the action taken from each agent. We define $A = A_1 \times ... \times A_n$ as the Cartesian product between the action set of each agent.
- $p_i : S, A, S \rightarrow \mathbb{R}$ is the reward function of the i-th agent. Similarly to what happens for the action sets, each agent may have a different reward function to shape a different behaviour.

As shown in the image below (1.4), at a generic time instant t, the environment is in a state $s_t \in S$ and each agent performs an action $a_{i,t} \in A_i$, causing a change in the environment state $s_{t+1} \in S$ depending on the transition function $f(s_t, [a_{1,t}, ..., a_{n,t}], s_{t+1})$. Each agent receives then a penalty or a reward depending on its reward function $r_{i,t+1} = p_i(s_t, [a_{1,t}, ..., a_{n,t}], s_{t+1})$.

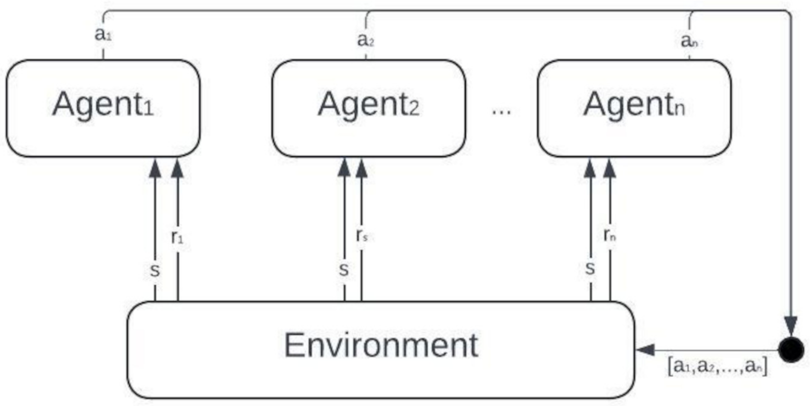


Figure 1.4: Graphical representation of the interactions between multiple agents and the environment

1.2.1 Key concepts

Types of Multi-Agent Systems Multi-agent reinforcement learning problems can be classified in three categories depending on the relationships between the different agents: cooperative, competitive and mixed [4].

- In cooperative environments all the agents share a common goal, therefore the success of an agent directly benefits also the other agents. This means that the strategies of each agent are aligned. Examples of environments of this kind include robot systems performing a collaborative task, drones swarming, a group of agents working together in a disaster recovery mission, and so on.

  A further classification consists in dividing agents into homogeneous and heterogeneous categories: homogeneous agents are those that share the same global policy and act in the same way at equal conditions, therefore they are indistinguishable and interchangeable [2]. Considering the best-case scenario of a fully observable environment, homogeneous agents are able to predict other agents action. Heterogeneous agents, on the other hand, differ in their policies, goals, or capabilities, leading to different behaviours even in similar situations.

- In competitive environments, agents have conflicting goals and the success of an agent is a defeat for the other agents. A particular case of competitive environment is zero-sum games, where the sum between the reward of each agent is zero, meaning that in an environment consisting of two agents A and B, at a generic time instant t, $r_{A,t} = -r_{B,t}$ [31]. Examples of competitive environments include strategic games such as chess and go, financial market where traders compete for profit, or adversarial scenarios like cybersecurity, where defenders and attackers have opposite goals.

  The non-stationarity is quite a big challenge in this kind of scenarios, since agents continuously adapt to their opponents' strategies, making the environment highly dynamic and unpredictable. The exploration-exploitation trade-off is also crucial, since agents need to balance the need to exploit known strategies with the need to explore new ones that might provide an edge over opponents.

- Mixed environments take elements from both cooperative and competitive settings. A typical case of mixed environment is when multiple teams compete between each others, but all the agents within a team collaborate together to defeat the other teams. This dual nature of mixed environments requires agents to develop both cooperative strategies to work effectively with their teammates and competitive strategies to beat their adversaries.

Agent Interactions In multi-agent reinforcement learning, interactions between agents play a crucial role in determining the success or the failure of a solution. Communication is a fundamental aspect of multi-agent systems, especially in scenarios in which the agents need to share information to achieve a common goal, such as the cooperative scenarios previously described.

Communication can help to partially solve the problem of *partial observability*, since agents can share with their neighbors their view of the environment, and at the same time they can use their neighbors observation to expand their knowledge about the surroundings.

Communication can also be used as a mechanism of coordination: agents of the same team can share short-term intentions and long-term strategies between each others, enabling more synchronized and efficient actions. For instance, in a multi-robot scenario, robots might share their planned paths to avoid collisions or to coordinate their movements when carrying an object together.

Furthermore, even though there are multiple agents involved in the problem, from the agent's perspective, the environment becomes stationary again when they exchange actions with one another in a fully observable environment. This is because the agent can precisely determine how it contributes to the environment update by anticipating how other agents' actions will affect the environment [5].

The challenge in implementing communication lies in ensuring that it is both effective and efficient: agents need to learn what information is valuable to share and how to interpret the messages they receive.

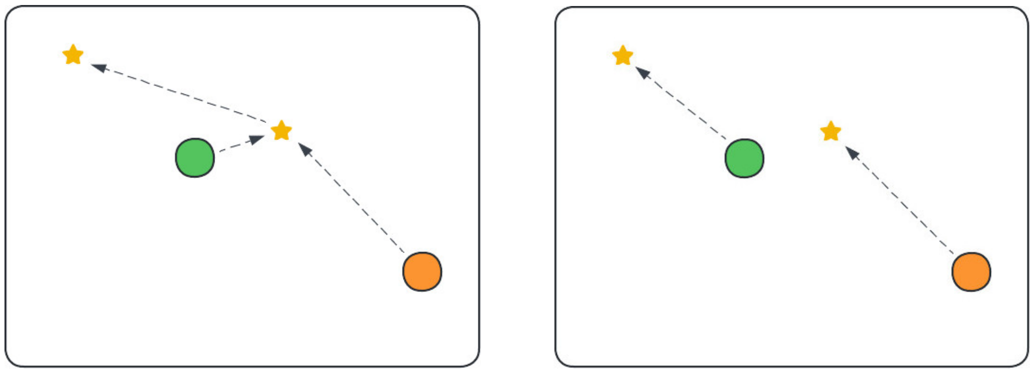
Reward structure The reward structure is the main element that determines whether the behaviour of the agent is cooperative or competitive towards the other agents within the environment (image 1.5).

When the cooperation component of the reward is greater than the component related to the contribution of the specific agent, a more cooperative behaviour emerges.

Cooperation can be obtained not only sharing positive rewards, but also by punishing all the agents for an error made by a specific agent. This can lead to scenarios in which agents help other agents to prevent situation of failure [5].

Both these strategies may lead to the problem of *credit assignment*, since part of the reward does not depend from the agent that did the action, but from the other agents.

On the opposite, if the reward obtained for collaborating with the others is low, agents might ignore it and act just for their own sake, fighting against the other agents to get the higher reward.



(a) In the competitive scenario the green agent acts greedily, trying to collect all the rewards alone

(b) In the cooperative scenario agents collaborate to collect all the rewards as fast as possible

Figure 1.5: Example of competition vs cooperation in the task *collect the items*. In the scenario on the left, only the agent who collects an item receive the reward, while in the scenario on the right both agents receive the reward

1.2.2 Main challenges

Compared to the single-agent scenario, the multi-agent case introduces different challenges. The presence of multiple learning agents creates a more complex and unpredictable environment, where each agent must not only learn the optimal strategy to reach its goal, but also adapt to the strategies of the other agents. In other words, this means that the optimal strategy of an agent depends on the strategies of the other agents, whether they are optimal or not.

This form of dependency between agents gives rise to new unique challenges such as non-stationarity, scalability issues, partial observability, and difficulties in credit assignment. These challenges must be carefully addressed to ensure successful learning and coordination among agents in multi-agent systems.

Non-stationarity The non-stationarity of the environment is one of the main challenges of multi-agent reinforcement learning.

Consider a single-agent scenario in which the environment state at a time t depends only on the previous environment state and the action performed by the agent. The agent is able to learn the optimal policy because it is able to understand the cause-effect relationship of its action on the environment.

In a multi-agent scenario this condition does not hold anymore, since the environment state transition does not depend on the action of a single agent, but on the action of many different agents which are simultaneously learning and adapting their policies. From the perspective of the single agent, the environment becomes non-stationary, therefore the Markov property no longer holds.

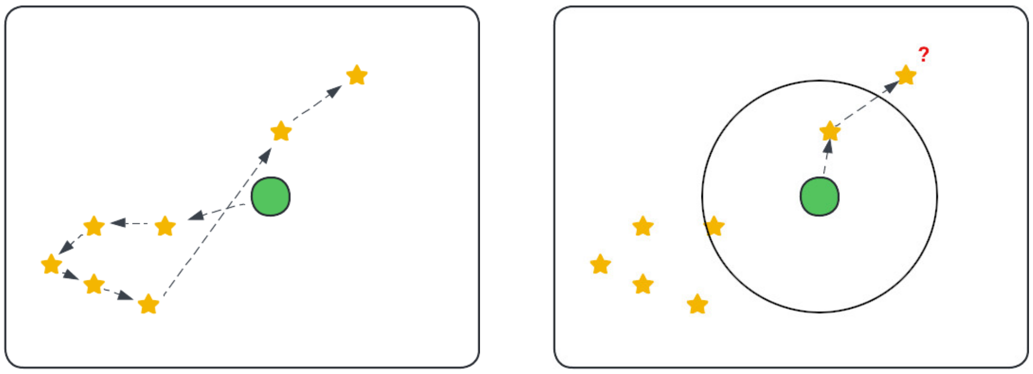
One of the most common ways to address the non-stationarity issue, involves adopting a centralized training model. In this framework, agents have a global perspective of the environment during the training phase, allowing them to better understand the dynamic of other agents. By providing additional information to the agents, the environment becomes more predictable, making it is easier for the agent to understand its dynamics.

Another approach is to transform the problem from multi-agent to single-agent, using the joint action $A = A_1 \times ... \times A_n$ as the new action set. This solution removes all the problems related to non-stationarity since the new formulation is a full-fledged single agent problem. However, this solution may not be suitable when the number of agents is high because of the action set dimensionality explosion that would make the problem too complex to be solved.

A third alternative is the so called *opponent modelling* [11] which consist in predicting other agents actions and act accordingly. By doing so the agent is able to better understand the influence of its action on the environment, reducing so the non-stationarity problem [21].

Partial observability Many multi-agent scenarios have to deal with a problem known as *partial observability*: unlike fully observable settings, where an agent has access to the entire state of the environment, in many multi-agent scenarios each agent can perceive only a subset of the overall state.

This partial view of the environment has significant implications for the Markov property, which asserts that the future state of the environment depends only on the current state and action, and not on the sequence of events that preceded it. In the presence of partial observability, the Markov property no longer holds because the observation of the agent do not capture the whole state of the environment, but only a subset; therefore the agent bases its decision-making process on incomplete and possibly misleading information. Consequently, agents may struggle to make optimal decisions and develop optimal policies (image 1.6).



(a) the agent can see all items in the environment. Initially, the agent does not choose the closest item because it recognizes that a more distant item is part of a cluster of items. By prioritizing the more distant item, the agent optimizes its long-term strategy, aiming to collect more items efficiently.

(b) the agent's view is limited to a specific radius and the agent cannot see items outside it. Consequently, the agent moves towards the nearest visible item, unaware that a better choice might be beyond its limited view, leading to a suboptimal decision.

Figure 1.6: Illustration of full observability (on the left) vs. partial observability (on the right) in a collect the items task.

To reduce the effects of partial observability, agents can adopt techniques such as recurrent neural networks to process sequences of observations, allowing them to maintain a form of memory and infer hidden aspects of the environment over time [10].

Another approach to dealing with partial observability is to adopt communication protocols between agents [35]: agents can be designed to share relevant information, mitigating the impact of partial observability and improving coordination among agents at the same time.

Scalability As said, a way to address the non-stationarity problem is to utilize a centralized learning model; however, this approach introduce a new challenge related to the limit of the central node.

For instance, in a scenario with hundreds of learning agents, the central node is responsible not only for storing the individual policies of each agent, but also for optimizing them all. This increases the computational cost significantly, as the central node must manage and refine a large number of distinct policies while considering their interactions within the environment.

A hybrid solution that addresses at the same time scalability and non-stationarity issues, is to use a decentralized setting where agents are able to communicate and exchange information with their neighbors. By using this model, each agent is responsible for storing and optimizing its own policy, but utilizes information from adjacent agents to improve the learning performance.

Credit assignment Credit assignment is a crucial challenge in Multi-Agent Reinforcement Learning, especially in fully-cooperative scenarios because of the large shared reward compared. This problem refers to the difficulty of attributing rewards to the individual actions taken by the agents.

In environments where multiple agents are involved, rewards are often the result of complex interactions between agents, making it challenging to determine which specific actions contributed to the outcome.

As described before, the reward function of an agent i is defined as $p_i : S, A, S \to \mathbb{R}$, therefore the reward (or the penalty) received by an agent does not depend solely from the action performed by that agent, but also from the action performed by all the other agents.

One of the main solution to address the credit assignment problem is the so called *reward shaping* [6]. This technique involves modifying the reward function to provide more immediate and informative feedback to agents about their actions. By designing rewards that reflect the contributions of individual actions more clearly, agents can more easily discern which behaviors are beneficial and adjust their strategies accordingly. For instance, imagine two robots which have to move an item from a starting position to a target in a 2D-environment. If we consider the reward following function:

$$r = \begin{cases} 100 & \text{if the item is at the target location} \\ 0 & \text{otherwise} \end{cases}$$

If the item reaches the target, all the robots receive a reward of 100. This setup does not provide feedback on how individual actions contributed to the outcome, making it difficult for each robot to learn which specific actions were beneficial. If instead we consider this reward function:

$$r_i = \text{progress\_towards\_target} + \text{progress\_due\_to\_robot\_i}$$

each agent receives a feedback based on how its specific action impacted the progress.

1.2.3 Training schemes

As already mentioned, scalability represents one of the main challenges in Multi-Agent Reinforcement Learning, especially during the training phase, due to the high state and action space dimensionality, which is a direct consequence of the large number of learning agents. To address this problem, two main training paradigms have been proposed, which are one the opposite of the other: the first strategy involves training the agents in a centralized manner, while the second method consists in training the agents in a distributed way.

A further classification can then be made depending on how agents evaluate their policy. Again, a centralized or a decentralized approach can be adopted.

Overall, three main settings emerge from these combinations: centralized training-centralized execution (CTCE), centralized training-decentralized execution (CTDE) and distributed training-decentralized execution (DTDE).

Centralized Training The main advantage of centralized training is that it allows agents to leverage shared information during the learning process, leading to more coordinated and effective policies.

Centralized training is particularly useful in those situations where is possible to simulate the environment reaction to agents actions. Being able to simulate the environment has two main advantages: first, it simplifies the process of collecting samples; second, it allows the sharing of an unlimited amount of information between the agents, as they are virtual entities operating on the same node.

Moreover, centralized training can be particularly beneficial in scenarios with many homogeneous agents because these agents can share the same policy, reducing the computational burden on the central node.

As anticipated, after the training is completed a further classification can be made, depending on how agents act during the execution phase (image 1.7).

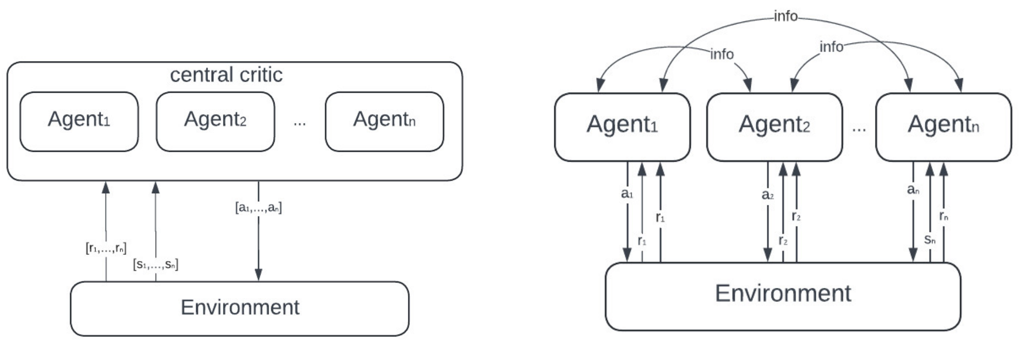
- **Centralized Execution**: in centralized execution, a single node dictates the actions of all agents based on a global perspective, allowing to reach high levels of coordination.

In this particular scenario, single-agent training methods can be adopted by using the joint observation and action spaces; however, this practise is often not feasible since the joint action and observation spaces can become excessively large, making the problem computationally intractable. In environments with many agents, what is usually preferred is to use a policy for each agent or group of agents, instead of a global policy for all the agents.

- **Decentralized Execution**: in decentralized execution, each agent maintains its own policy which maps the local observations to actions.

  CTDE approach strikes a balance between the benefits of shared information and the scalability challenges: the shared information available during the training allows for a better learning if compared to a completely decentralized setting, speeding up the learning process and reducing the effects of non-stationarity; while the decentralized execution address the scalability problem typical of this kind of systems, since each agent operates on their own without the needing of a central authority that would be a bottleneck for the network.

  For these reasons CTDE currently represents the state of the art paradigm [5] for Many-Agent Reinforcement Learning, providing a robust framework that combines the advantages of centralized training with the practical benefits of decentralized execution.



(a) Centralized execution (b) Decentralized execution

Figure 1.7: Centralized training schemes

Distributed Training In a similar way to CTDE, in distributed training each agent maintains its own policy which maps the local observation to a distributions of actions (image 1.8).

The biggest advantage of this modality is its fully decentralized nature, which enhances scalability, distributing the computational burden on each node instead of concentrating it in a single node. On the other hand the lack of coordinator node represents a big challenge because each agent is an independent learner that has only a partial vision of the surrounding environment, making the non-stationarity a very relevant issue and complicating the learning process.

To mitigate the effects of non-stationarity, one potential strategy is to enable agents to share information with their neighbors [34]. This approach can help each agent gaining a broader view about the environment and the actions of other agents, improving its ability to learn effectively. For instance, local communication protocols can be implemented, allowing agents to exchange information about their observations, actions, or rewards with nearby agents.

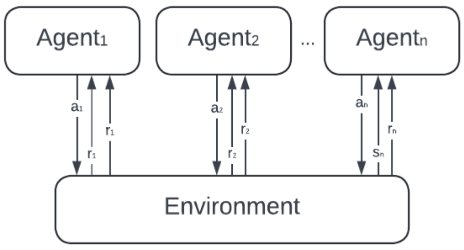


Figure 1.8: Distributed training, decentralized execution

Chapter 2

Frameworks and Technologies

This chapter shows and explains the main technologies and frameworks that have been used to run the experiments described in the following chapters.

The first part of this chapter focuses on presenting Gymnasium, an open-source library that allows to define single-agent environments, while in the second part is described in details the reinforcement learning framework RLlib.

2.1 Gymnasium

Gymnasium [29] is an open-source library that enables the definition of reinforcement learning environments. Thanks to its compatibility with a wide range of reinforcement learning frameworks, Gymnasium is widely used by researchers and developers in the field, allowing users to move from a framework to another without having to be worried about compatibility issues.

One of the reason why Gymnasium is so popular nowadays, is its extensive collection of built-in environments. These environments, that range from simple tasks in two-dimensional word to complex simulations, make Gymnasium a great resource for testing and comparing different reinforcement learning models. In addition to this, Gymnasium also provides a large suite of utilities, designed to make the researchers' work easier, by simplifying the complex processes involved in setting up, running, and evaluating experiments.

2.1.1 Environments

Defining a custom environment in Gymnasium is a straightforward operation, since it just takes the implementation of a couple of methods and the definition of a few variables, as shown in the listing 2.1.

This user-friendly approach encourages the development of personalized environments, allowing developers to customize environments to their specific needs, and facilitating experimentation and innovation in reinforcement learning research.

```python
class CustomEnvironment(gymnasium.Env):

    def __init__(self, config):
        super().__init__()
        self.action_space = ...
        self.observation_space = ...

    def reset(self):
        ...

    def step(self, action):
        ...

    def render(self):
        ...

    def close(self):
        ...
```

Listing 2.1: Defining a custom environment in Gymnasium

The method `reset` is used to set the environment to its initial state and returns the first agent observation along with an information dictionary, useful for debugging and collecting metrics.

In a typical usage scenario, the `reset` method is called at the beginning of an episode, guaranteeing that the environment is in a known state before the agent starts interacting with it. The returned observation allows the agent to take its first action based on the current state of the environment.

After the agent decides which action to take, based on the observation in its possession, the method `step` of the environment is called, passing the chosen action as parameter. Inside this method are coded how the environment reacts to the agent actions, and the reward function. For instance, in case of a two-dimensions environment in which the agent is free to move, in the `step` method is called the position update. The `step` method always returns a tuple composed by five elements: the next observation of the agent, the reward for the action, a boolean indicating if the episode is ended, a boolean indicating if the episode has been truncated and an information dictionary.

The methods `render` and `close`, while not essential for basic environment functioning, provide important additional capabilities. The `render` method is used to visually represent the environment, which can be particularly helpful for debugging, understanding the agent's behavior, or just for having a visual representation of the simulation. The `close` method is instead responsible for properly terminating any resources or processes that were opened during the use of the environment, such as graphical windows or simulation processes.

2.1.2 Action and observation spaces

Defining the action and observation spaces is a fundamental operation to enable the environment working effectively with a reinforcement learning framework. The observation space determines the shape of the learning model input, while the action space represents the set of possible actions the agent can take in response to the observations, therefore the output of the learning model.

Action and observation spaces in Gymnasium are defined using specialized classes provided by the library, such as `Box`, `Discrete`, `MultiDiscrete`, and `MultiBinary`. These classes allow the user to specify the range or the set of admissible values for each one of the two spaces (i.e., action and observation).

Defining a space involves specifying its dimensions and the type of values it can hold. Each different class serves a specific purpose:

- `Box` is used to define vectors or matrices, both continuous and discrete. It requires to specify the shape of the space and the lower and higher bounds, and it also allows to specify different bounds for each dimension.

  `Box` can be used also for modelling images, as shown in the listing 2.2.

  ```
  Box(low=0, high=255, shape=(3,512,512), dtype=np.uint8)
  ```

  Listing 2.2: Gymnasium observation space for a 512x512 RGB image

- `Discrete` is used to define a space with a finite set of values, often representing actions or states in environments with a limited number of choices.

  In the example below it is represented the action space used in a two-dimensional environment for the agent movement, where at each step the agent can move up, down, left or right.

  ```
  actions = ["UP", "RIGHT", "DOWN", "LEFT"]
  action_space = Discrete(len(actions))
  ```

  Listing 2.3: Gymnasium action space to handle a four-directions movement

- `MultiDiscrete` supports multiple discrete values with multiple axes. It is the same as `Discrete` but it works along multiple dimensions.

  In the example below, at each step the agent can move up, down, left or right in three different ways: running, walking, or crawling. The number of possible combinations is given by the Cartesian product between the different sets.

```
direction = ["UP", "RIGHT", "DOWN", "LEFT"]
movement_type = ["RUN", "WALK", "CRAWL"]
action_space = Discrete(np.array([
    len(direction),
    len(movement_type)]))
```

Listing 2.4: Example of a Gymnasium `MultiDiscrete` action space to handle a three-ways four-directions movement

- `MultiBinary` supports boolean vector and matrices and it can be used for modelling masks.

```
observation_space = MultiBinary([64, 64])
```

Listing 2.5: Example of a Gymnasium `MultiBinary` observation space

In addition to the listed fundamental spaces, Gymnasium allows the creation of more complex and flexible spaces by combining two or more of these fundamental spaces through composite spaces such as dictionaries (`Dict`) and tuples (`Tuple`). This is particularly useful when designing complex environments where the observation space may be composed by multiple elements which may have different type or constraints.

The example below (listing 2.6), shows how is possible to use `Dict` to model an action space that allows the agent to move at any direction and speed.

```
action_space = Dict({
    "direction": Box(low=-1, high=1, shape=(2,1), dtype=np.float32),
    "distance": Box(low=-np.inf, high=np.inf, shape=(1,1),
        dtype=np.float32)
})
```

Listing 2.6: Example of a Gymnasium action space obtained through `Dict`

On each one of this spaces, whether it is fundamental or obtained as combination of more simple spaces, it is possible to call the method `sample` to get a random value belonging to that space. This is particularly useful for initializing values, testing, or simulating random actions during the training process, providing a straightforward way to generate valid data points within the defined constraints of the space.

2.1.3 Simulating an episode

Once the methods described in the previous section are properly implemented, simulating the execution of an episode becomes a very straightforward task. After having initialized the environment, the method `reset` is called and the first observation is returned. This observation will serve as first input of the agent's policy function. With the environment initialized and the initial observation obtained, the next step is to enter the main loop of the simulation. Within this loop, the agent selects an action based on its policy, which could be a trained model, or even a simple heuristic; the action is then passed to the environment by calling the `step` method.

The `step` method is the core of the interaction between the agent and the environment. It processes the agent's action, updates the environment's state, and returns several useful information. After having updated the environment state, the environment is rendered using the `render` method to have a visual feedback about what the agent is doing and how its action are affecting the environment.

The loop continues, with the agent selecting actions and the environment responding, until either the `terminated` or `truncated` flag is set to `True`. At this point, the episode is considered complete. To ensure a clean exit from the simulation, the `close` method is then called in order to properly shut down possible external processes or resources.

The below listing (2.7) illustrates the process just described, using a random policy that selects actions by randomly sampling values from the agent's action space.

```python
config_params = ...
env = CustomEnv(config_params)
obs, info = env.reset(seed=3010)
terminated = False
truncated = False

while not terminated and not truncated:
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(action)
    env.render()
env.close()
```

Listing 2.7: Simulating an episode of a Gymnasium environment

2.2 RLlib

RLlib [15] is a powerful library for reinforcement learning and is part of the Ray ecosystem [19], an open-source framework designed to simplify distributed computing. RLlib provides scalable, flexible, and easy-to-use tools for developing, training, and deploying reinforcement learning models.

RLlib is designed to simplify the often intricate process of reinforcement learning, abstracting many of the complexities involved in scaling reinforcement learning algorithms across multiple nodes, allowing users to focus more on developing and fine-tuning their models.

This section contains an overview of RLlib's functionalities and capabilities. Starting from a high-level description of the available algorithms, the section continues by showing methods that can be applied to obtain better results, such as custom callbacks and curriculum learning. Lastly is presented the RLlib functionality that allows to handle multi-agent environments, also describing the main differences with the single-agent case.

2.2.1 Supported algorithms

Algorithms are the main component of RLlib: they link an environment to a policy, or a set of policies, and aim to optimize these policies based on episode trajectories.

RLlib supports a wide range of reinforcement learning algorithms, providing users with the flexibility to choose the most suitable method for their specific applications. Among the most popular algorithms supported there are Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN), both of which are widely used in the reinforcement learning community for their effectiveness and robustness. Beyond PPO and DQN, RLlib also supports many other algorithms such as Soft-Actor Critic (SAC) [7] and DreamerV3 [8].

Algorithms can be classified in three categories: offline, model-free, and model-based:

- **offline** algorithms are used for training agents using pre-collected samples, therefore they do not need any interaction with the environment;
- **model-free** algorithms, such as DQN and PPO, are among the most common and can be further classified in on-policy (policy-based) and off-policy (value-based);
- **model-based** algorithm, such as DreamerV3, involve learning a model of the environment's dynamics and using this model to plan or simulate outcomes.

The table below (2.1) lists the different RLlib algorithm and some of their characteristics. All the RLLib algorithms, except for DreamerV3, can work with both TensorFlow and PyTorch as underlying framework, and most of them are enabled to work in multi-agent environments as well, as shown in the table below (2.1).

Algorithm	**Class**	**Discrete actions**	**Contintuous actions**	**Multi-Agent**
Behavioural Cloning	Offline	✓	✓	✓
Conservative Q-Learning	Offline	✗	✓	✗
MARWIL	Offline	✓	✓	✓
APPO	Model-free On-policy	✓	✓	✓
PPO	Model-free On-policy	✓	✓	✓
IMPALA	Model-free On-policy	✓	✓	✓
Deep Q-Networks	Model-free Off-policy	✓	✗	✓
Soft Actor Critic	Model-free Off-policy	✓	✓	✓
DreamerV3	Model-based	✓	✓	✗

Table 2.1: Algorithms available in RLlib

Configuring and training an algorithm All the RLLib algorithms share a common set of configuration parameters in addition to the algorithm-specific parameters and allow the user to specify custom callbacks which are executed, for instance, after the end of an episode, or after the training is completed.

Thanks to the very simple APIs, setting up a basic training routine using RLlib is a straightforward operation, as shown in the listing below (2.8). Each algorithm has its own algorithm configuration class which follows a builder pattern to set the different algorithm parameters.

This example shows the minimal structure needed to configure and train a reinforcement learning agent using RLlib. After having configured the model, the method `build` is called and the algorithm is ready to be trained for any number of training iterations.

In this example, the algorithm is implemented using PyTorch; however, RLlib does not depend on any specific deep-learning framework and can work with both TensorFlow [1] and PyTorch [22], depending on the user's preference.

```
from ray.rllib.algorithms.ppo import PPOConfig

config = (
    PPOConfig()
    .training(
        gamma=0.95,
        lr=0.001,
        train_batch_size=128)
    .environment("CartPole-v1")
    .framework("torch")
)

algo = config.build()

for _ in range(10):
    print(algo.train())

algo.evaluate()
```

Listing 2.8: Simple reinforcement learning in RLlib using PPO

Parameters tuning The method shown in the listing 2.8 is just one of the three ways that can be used to configure and train an algorithm.

A second method consists of using the Ray tuning tool, which is part of the Tune component of Ray. This second solution allows to define a search space and perform a grid search in order to find the best hyperparameter configuration.

The example below (listing 2.9) shows how is possible to do so.

```
from ray import train, tune

config = (
    PPOConfig()
    .environment("CartPole-v1")
    .training(
        gamma=tune.grid_search([0.90, 0.95, 0.975]),
        lr=tune.grid_search([0.01, 0.001, 0.0001]),
        train_batch_size=tune.grid_search(128,512))
    .framework("torch")
)
```

```
tuner = tune.Tuner(
    "PPO",
    param_space=config,
    run_config=train.RunConfig(
        stop={"env_runners/episode_return_mean": 150.0}
    ),
)

tuner.fit()
```

Listing 2.9: Hyperparameters tuning using Ray Tune functionalities

the method `fit` returns a `ResultGrid` that can be used to better analyze the training result and also retrieve checkpoints.

Custom callbacks RLlib offers a powerful way to extend and refine the training process of reinforcement learning models, based on custom callbacks. Callbacks provide a mechanism to execute custom code at specific points during training, such as before or after each environment step, episode, or training training, allowing users to monitor and log various aspects of the training process.

One common way to use custom callbacks is for monitoring and logging the training process, but callbacks can also be used to share information between different policies. By implementing a custom callback, users can track metrics and statistics that are not included in RLlib's default output; for instance, users might use callbacks to log additional performance metrics, such as agent behavior patterns, or environment interactions. This enhanced monitoring capability can be crucial for diagnosing issues, analyzing learning trends, and gaining deeper insights into the training dynamics.

Custom callbacks can also be employed to implement dynamic modifications to the training process. For example, you can use callbacks to adjust hyperparameters on-the-fly based on performance metrics or specific conditions during training. This flexibility enables adaptive learning strategies where parameters like learning rates or exploration strategies can be adjusted in response to observed performance, potentially leading to more efficient training and better overall results.

Adding a custom callback requires to extend the `DefaultCallbacks` class and add the new defined callback to the algorithm configuration through the `callback` method.

In the listing below (2.10) a dummy callback is defined and added to the algorithm during its configuration.

```python
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.algorithms.callbacks import DefaultCallbacks

class DummyCustomCallback(DefaultCallbacks):
    def __init__(self):
        super().__init__()

    def on_episode_start(self, *, worker, base_env, policies,
        episode, env_index):
        ...

    def on_episode_step(self, *, worker, base_env, policies,
        episode, env_index):
        ...

    def on_episode_end(self, *, worker, base_env, policies, episode,
        env_index):
        ...

    def on_sample_end(self, *, worker, samples):
        ...

config = (
    PPOConfig()
    .environment("CartPole-v1")
    .framework("torch")
    .callbacks(DummyCustomCallback)
)

algo = config.build()
```

Listing 2.10: Example of custom callback in RLlib

Curriculum learning Curriculum learning is an advanced reinforcement learning training strategy that consists in organizing the learning process in a way that gradually introduces increasingly complex tasks.

The core idea behind curriculum learning is to start with simpler and easier problems and progressively move towards more complex and difficult ones, allowing the agent to learn by step. This approach aim to emulate the human learning process, where basic tasks are mastered before tackling more complex ones.

In the context of reinforcement learning, curriculum learning involves designing a sequence of tasks or environments that gradually increase in difficulty. For example, in a robotic control task, an agent might first be trained in a simplified environment with fewer obstacles and then gradually be exposed to more challenging scenarios with additional obstacles or more complex dynamics. By starting with simpler tasks, the agent can build a solid understanding of basic skills and strategies, which can then be transferred to more complex scenarios. This staged approach helps the agent avoid the difficulties of starting with a very challenging task, which might otherwise making learning harder due to the high complexity.

Curriculum learning also help reducing the risk of an agent getting stuck in local optima, because the way tasks are structured encourages the agent to constantly explore new strategies.

RLlib allows to apply curriculum learning to a task, as shown in the example below (listing 2.11): after the average episode return is higher than a certain value, the task of each environment present in the algorithm is updated to a more difficult one.

```python
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.algorithms.callbacks import DefaultCallbacks

tasks = [...,...,...]

class CurriculumLearning(DefaultCallbacks):
    def on_train_result(self, algorithm, result, **kwargs):
        task_idx = 0
        if result["env_runners"]["episode_return_mean"] > 200:
            task_idx = 2
        elif result["env_runners"]["episode_return_mean"] > 100:
            task_idx = 1
        algorithm.env_runner_group.foreach_worker(
            lambda ev: ev.foreach_env(
                lambda env: env.set_task(tasks[task_idx])))

algo = (PPOConfig()
    .environment("CartPole-v1")
    .framework("torch")
    .callbacks(CurriculumLearning)).build()
```

Listing 2.11: Curriculum learning in RLlib

2.2.2 Extension to multi-agent scenarios

RLlib provides support to the previously explained Gymnasium environments; however, it also supports environments that might require more advanced setups, such as multi-agent environments.

To create a multi-agent environment in RLlib, the environment must extend the `MultiAgentEnv` class provided by the RLlib framework. This class serves as the foundation for multi-agent environments, offering a structured way to manage the interactions of multiple agents within a single environment. Implementing this class requires defining the same core methods and the same variables that are essential for any Gymnasium environment: `reset()`, `step()`, `render()`, `close()`, `actions_space`, and `observation_space`.

When creating a multi-agent environment in RLlib by extending the `MultiAgentEnv` class, the methods `reset` and `step` do not return simple objects like in single-agent environments; instead, they return dictionaries where the keys are the agent ids. In particular, in the observation dictionary only the observations for the agents that are expected to take an action in the next step should be included. In the same way, when updating the environment through the `step` method, a dictionary containing the different agents' actions is expected.

This structure allows each agent to have its own individual observation, action, reward, and termination status. For instance, after executing the `step` method, the environment will return a dictionary that maps each agent's id to its respective observation, reward, and done signal. This design ensures that the environment can handle the simultaneous and independent actions of multiple agents, facilitating complex interactions and making it suitable for a wide range of multi-agent reinforcement learning tasks.

Multi-agent interaction dynamics The RLlib API offers significant flexibility in designing multi-agent environments, making it possible to model various interaction dynamics among agents. Whether the scenario involves turn-based games, environments where all agents act simultaneously, or a hybrid structure, RLlib's architecture supports it. This flexibility is essential for accurately simulating complex real-world scenarios or intricate games where agent interactions vary significantly.

For example, in turn-based games like chess and Go, agents take turns making decisions. In such cases, during the first turn, only the first agent takes an action, while in the next turn, only the second agent does.

On the other hand, in environments where all agents act simultaneously, such as in many real-time strategy games or multi-agent simulations, every agent selects and executes an action at each environment step.

Additionally, RLlib supports hybrid solutions where agents may or may not perform an action at any given moment based on the specific situation. Agents expected to take an action at a time *t* are those whose observation time *t-1* was returned.

Handling multiple policies While in single-agent reinforcement learning a single policy is used, in multi-agent reinforcement learning different agents may use different policies to select their actions, whether their goal is the same or not, as depicted in the image below (2.1).

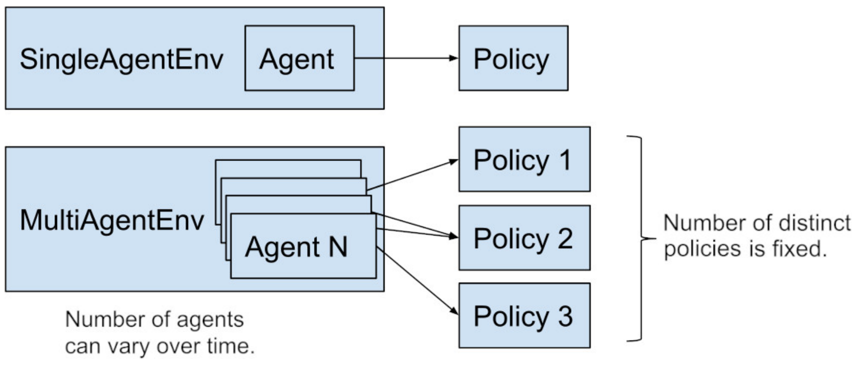

SingleAgentEnv
Agent
Policy
MultiAgentEnv
Agent N
Policy 1
Policy 2
Policy 3
Number of distinct policies is fixed.
Number of agents can vary over time.

Figure 2.1: Difference in the number of policies between single-agent and multi-agent scenarios. Source: `https://docs.ray.io/en/latest/rllib/rllib-env.html`

If not differently specified, all the agents within the environment utilize the same default policy. Nevertheless it is possible to modify the number of policy to allow the agents act in different ways.

In the example below (listing 2.12), an algorithm for a custom environment containing an undefined but fixed number of agents is configured. At each agent is assigned its own policy, which is determined using the policy mapping function.

```python
from ray.rllib.algorithms.ppo import PPOConfig

policies = {f"agent-{i}": (None, None, None, {}) for i in
    range(n_agents)}

config = (
    PPOConfig()
    .environment("custom_environment")
    .framework("torch")
    .multi_agent(
        policies=policies,
        policy_mapping_fn=(lambda agentId, *args, **kwargs: agentId))
)

algo = config.build()
```

Listing 2.12: Learning multiple policies in RLLib

It is also possible to specify which policies should be updated and which ones instead are static by using the configuration parameter `policies_to_train`.

When multiple policies are present, computing an action requires to specify which one of the policies to use. In the example below (listing 2.13), for each agent in the previous step observation dictionary is computed an action using the policy specified by the given mapping function. Once all the actions are computed, they are passed to the environment through the `step` and the new observations are returned along with rewards and termination flags.

```
actions = {}
for agent in obs.keys():
    actions[agent] = algo.compute_single_action(
        obs[agent],
        policy_id=policy_mapping_function(agent)
    )

obs, reward, terminated, _, infos = env.step(actions)
```

Listing 2.13: Computing actions using multiple policies

Advanced functionalities In addition to what already said, RLlib offers some advanced functionalities that allow the users to highly customize their environments and training processes:

- **experience sharing**: experience sharing in RLlib refers to the ability of different policies or agents to share and learn from each other's experiences. This functionality can be particularly advantageous in scenarios with limited interaction time or when agents face similar challenges, as it helps in creating a more generalized and effective learning process across multiple agents.

- **grouping agents**: grouping agents in RLlib involves organizing agents into distinct groups based on their goal or characteristics, in order to manage their interactions and learning processes more efficiently. Agents group can be defined through the `with_agent_groups` function of the `MultiAgentEnv` class.

- **hierarchical environments**: hierarchical environments in RLlib allow the creation of complex, multi-level learning structures where agents operate at different levels of abstraction. This feature supports the development of hierarchical reinforcement learning [23] strategies, where high-level policies can coordinate the actions of lower-level policies, enabling the decomposition of complex tasks into more manageable sub-tasks and facilitating the learning process.

Chapter 3

Contribution

3.1 Problem definition

In the last years, training schemes involving centralized training have been extensively explored by the researchers, while distributed training has received much less attention. Nevertheless, as anticipated in the previous chapters, centralized training is not always a valid solution due to different problems:

- **scalability issues**: a high number of learning agents makes difficult to train in an effective way due to the limited resources of the central node.
- **communication overhead**: centralized training requires frequent communication between the agents and the central node, which can become a bottleneck, especially in scenarios where the number of agents is high and the observation space is big.
- **limited flexibility**: the presence of a central node during the training limits the flexibility of the network. Moreover, if the central node fails, the whole learning process does.
- **necessity of an accurate simulator**: centralized training typically relies on simulators capable of accurately handle the dynamic of the environment.

The goal of this work is to compare the effectiveness of different distributed training strategies that can be applied in those situations in which centralized training is for some of these reasons not feasible.

To achieve this objective, a custom multi-agent environment was developed and some experiments were conducted on it, in order to compare the training performance of different distributed training schemes with the centralized training scenario one.

3.2 Inspected distributed learning strategies

This section outlines the distributed learning strategies that were tested in this work: independent learners, NN-averaging, NN-consensus and experience sharing.

3.2.1 Independent learners

Among the tested techniques, independent learners is the most straightforward approach. In this approach, each agent learns independently and does not share any information with the other agents, except the information necessary to compose the observation.

The main advantage of this strategy is the high scalability; in fact, since agents are completely independent, increasing the number of agents does not affect in any way the computational burden of a specific node.

Since this strategy does not require to share experience or part of the network between the agents, it allows for heterogeneous agents: different agents could adopt different networks or different input shapes without affecting the correct operating of the collective.

However, this approach comes with significant challenges, especially in scenarios in which the interaction and the coordination between the agent is important such as the one taken in exam. Since each agent is completely independent, this approach is highly affected from the non-stationarity issue.

3.2.2 NN-averaging

Unlike independent learners, where each agent learns its policy in isolation, NN-averaging introduces information sharing between agents.

Each agent maintains its own neural network, but periodically averages the weights with those of the other agents, typically its neighbors. This process allows the agents to influence each others learning, allowing the development of more coordinate behaviours. By sharing network weights, agents exchange knowledge about the environment, helping to mitigate the non-stationarity problem. The equation below (3.1) describes the process of updating the weights, where w_t^i represents the weights of the agents i at the time t, and *nbrs(i)* represents the set of neighbors of the agent i.

$$w_{t+1}^i \leftarrow \frac{(w_t^i + \sum_{j \in nbrs(i)} w_t^j)}{|nbrs(i)| + 1} \tag{3.1}$$

Even though this approach introduces information sharing, it is still very scalable since the information sharing is normally limited between a small amount of neighbors; moreover, the amount of shared information can be easily controlled by choosing how often to share the network.

If the weight sharing is very frequent, agents tend to converge toward the same policy, potentially reducing the overall performance. On the other hand, not enough frequent averaging might not provide enough coordination, leaving the agents too independent. The frequency of the information sharing becomes therefore an additional hyperparameter of the model that the user has to tune.

NN-averaging can be modified to give more influence to certain agents based on their performance, for instance the reward they obtained in the last episodes. By doing the weighted average, agents that perform poorly can benefit more and learn faster, while agents that perform better are not slowed from the agents that perform worse, accelerating the learning process and improving the performance of the entire system.

In the equation below (3.2), the reward of each agent at the time t is used as the factor for averaging when computing the weights of the step *t+1*.

$$w_{t+1}^i \leftarrow \frac{r_t^i w_t^i + \sum_{j \in nbrs(i)} r_t^j w_t^j}{r_t^i + \sum_{j \in nbrs(i)} r_t^j} \tag{3.2}$$

Weighted averaging can also be used to limit the amount of changes in the agent's neural network, helping to develop different policies instead of converging to the same one. For instance, it is possible to assign higher weight to the agent's own network (e.g., 0.75) distributing the remaining weight among their neighbors. By doing so, agents still share information, but at the same time their network is not completely changed at each iteration.

Both in the weighted-averaging case and in the basic one, the averaging is done at the end of each training iteration. Each agent share its network with a fixed set of neighbors that is selected before starting the learning. It is worth mentioning that if we think the agents network as a graph, where each agent represents a node, and each connection represents an edge, the resulting graph is connected; in other words, there are no disconnected subgraphs or isolated agents.

3.2.3 NN-consensus

NN-consensus is another learning approach based on sharing network's weights between neighbors. Differently from NN-averaging, where the resulting network is the result of a combination of multiple networks, in NN-consensus the network is chosen as the best performing network between the agent and its neighbors. In NN-consensus, after each training iteration, each agent evaluates the performance of its own network, shares the result with their neighbors and then selects as new network the best performing one (equation 3.3).

$$w^i_{t+1} \leftarrow w^k_t \quad \text{, where} \quad k = argmax_{j \in nbrs(i) \cup i}(r^j_t) \tag{3.3}$$

Like in NN-averaging, the frequency of the updates is a key factor in the learning process. For instance, if the neighborhood with which the agent shares its performance consists of the entire network of agents, then at each synchronization step, all agents will end up adopting the same policy. This can lead to rapid convergence but it might also reduce the diversity of strategies within the agent population, potentially making the system less robust to changes or variations in the environment.

Like NN-averaging, NN-consensus has been implemented through a callback that is called after each training iteration. As performance metric, the mean episode reward of the policy has been chosen, while the agent network structure is the same as the previous method one.

3.2.4 Experience sharing

Unlike NN-averaging and NN-consensus, in which agents share neural network weights, in experience sharing agents share their experience as a list of tuples of type $\langle \text{initial state}, \text{action}, \text{final state}, \text{reward} \rangle$.

$$\text{replay\_buffer}^i_{t+1} \leftarrow \text{replay\_buffer}^i_{t+1} \cup \text{trajectory}^j_t \quad \forall \quad j \in nbrs(i) \tag{3.4}$$

Experience sharing can enhance the learning efficiency especially in environments where it may be difficult to get useful information due to sparse interaction. Through experience sharing, agents can use a broader range of data, accelerating the learning process. For instance, an agent that has not encountered a particular scenario yet, can still learn from the experiences of another agent that instead has.

Another advantage of experience sharing is that by learning from the experiences of others, an agent can reduce the need for extensive exploration, as it can leverage the knowledge gained by others.

Experience sharing can be done either asynchronously, where agents periodically exchange experiences at fixed intervals, or synchronously, where agents share experiences in real-time as soon as they occur.

In the adopted strategy, each agent builds its trajectory, and once the episode is over, it shares the collected information with their neighbors.

Chapter 4

Evaluation

4.1 Introduction to the test environment

The scenario taken into account consists in a “collect the items” task: in a two-dimensional unbounded world, are located a certain amount of agents (*n_agents*) and items (*n_items*), as shown in the image below (4.1).

Each agent shares the same goal, that is to collect all the items present in the environment in the fewest number of steps. Achieving this goal requires the agents to collaborate effectively, coordinating their movements to optimize the items collection. This makes the task a pure coordination problem, where success depends on the agents’ ability to work together.

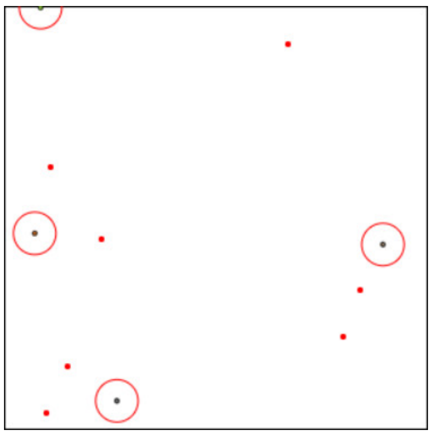


Figure 4.1: Visual representation of the environment at a generic time instant t. The red dots identify the items, while the colored encircled dots represents the agents.

One of the main challenges is to ensure that the agents do not overlap their intent excessively, for example by aiming to collect the same item as another agent. Instead, the agents should cover the space in an efficient way, avoiding to leave the items uncollected for too long.

In the following pages are described in detail the characteristics of the environment, such as action and observation spaces and reward structure.

4.1.1 Configuring the environment

The environment was implemented by extending the `MultiAgentEnv` class of the RLlib framework. When creating a new environment, an `EnvironmentConfiguration` object is required. This object contains all the configurable parameters of the `CollectTheItems` environment and is used to avoid passing plenty of parameters directly to the environment instance.

Here are listed some of the parameters that is possible to configure: *n_agents*, *n_items*, *spawn_area* and *agent_range*.

The first two parameters, as previously explained, specify respectively the number of agents and the number of items. Even though agents are able to freely move within the environment, each agent and item spawns within an area of size $spawn\_area \times spawn\_area$, which is the same area that is visible when rendering the environment.

The *agent_range* parameter represents instead the action range of the agent. When the distance between an item and the agent is less equal than *agent_range*, the item is considered collected.

Additional parameters related to more specific aspects of this section, such as action and observation spaces, will be described later.

4.1.2 Observation space

This task considers only homogeneous agents, therefore all the agents have the same observation space. The observation space currently in use is the result of multiple tests, aimed to find the combination of data that allows for the best performance.

The shape of the observation space depends on three parameters specified in the `EnvironmentConfiguration` object: *visible_nbrs*, *visible_items* and *memory_size*.

The parameters *visible_nbrs* and *visible_items* are used to specify how many agents and how many items the agent is able to see. By adopting this strategy, the observation size is the same regardless the number of actors or items present at a generic time t in the environment, therefore is possible to use the same trained model for different environment configurations.

In particular, each agent knows the relative position of the *visible_nbrs*-closest agents and of the *visible_items*-closest items. Each one of these relative positions is defined as a dictionary containing direction and distance from the object. The direction is specified by a normalized vector, while the distance is calculated as the norm of the distance vector normalized using this formula:

$$obs\_dst\_from\_nbr1 = np.log(1 + ||distance\_vector(agent, nbr_1)||)$$

By normalizing the distance in this way, the trained model is able to work with environments having different *spawn_area* compared to the one the model has been trained for.

Since the number of available items decreases over time, sooner or later the number of items left will be lower than *visible_items*. Since the observation space must have a fixed size, the remaining items' observations are filled with empty values (direction=[0,0], distance=-1).

What described until now contains only spacial information. Adding temporal information can help the agent to better understand how its actions affect the environment. Temporal information can be added by setting the parameter *memory_size* higher than one. For instance, if *memory_size* is three, at each observation the agent perceive the environment status at the time *t*, *t-1* and *t-2*. This is realized by wrapping what described before in a dictionary that has as many keys as *memory_size*.

The resulting observation space comprehensive of *memory_size* becomes the following:

```
direction = Box(low=-1, high=1, shape=(2,1), dtype=np.float32)
distance = Box(low=-np.inf, high=np.inf, shape=(1,1),
    dtype=np.float32)

nbrs = Dict({f"nbr-{i}": Dict({"direction": direction, "distance":
    distance}) for i in range(self.visible_nbrs)})
items = Dict({f"item-{i}": Dict({"direction": direction, "distance":
    distance}) for i in range(self.visible_items)})

time_t_obs = Dict({"nbrs": nbrs, "items": targets})

obs_space = Dict({f"t[-{t}]": time_t_obs for t in range(0,
    self.memory_size)})
```

Listing 4.1: Observation space of a generic agent

Since the observation contains nested dictionaries, which are not directly supported from RLlib algorithms, before returning the observation, the method `flatten_space` is called to transform the observation in a list.

4.1.3 Action space

Experiments were conducted using both PPO and DQN. Since DQN does not support a continuous action space, two different formulations were implemented: the first based on continuous actions and the second on discrete actions.

Continuous action space The continuous action space is defined in a similar way as the observation space explained before: the first component represents the direction of the movement, while the second one the speed.

In the setup below (listing 4.2), the direction component is a vector normalized within the range [-1.0, 1.0], which determines the direction in which the agent will move (the first component represents the movement along the x axis, while the second on the y). The speed component, ranges from 0.0 to 1.0 and controls how fast the agent moves in the specified direction.

```
def action_space(self, agent):
    direction = Box(low=-1.0, high=1.0, shape=(2,1),
        dtype=np.float32)
    speed = Box(0.0, 1.0, dtype=np.float32)
    return flatten_space(Tuple([direction, speed]))
```

Listing 4.2: Continuous action space definition

Discrete action space The discrete action space depends on two parameters of the `EnvironmentConfiguration` object: *movement_sensitivity*, and *speed_sensitivity*. In particular, the space is defined as follow:

```
def action_space(self, agent):
    return Discrete(self.movement_granularity *
        self.movement_granularity * self.speed_granularity)
```

Listing 4.3: Discrete action space definition

The parameter *movement_sensitivity* determines the number of possible movement directions available to the agent. For instance, if *movement_sensitivity=3*, the agent can move in one of eight directions: up, down, left, right, up-left, down-left, up-right, down-right, or choose to not move at all. The same concepts applies to *speed_sensitivity* but mapping the values in the range [0,1]. Increasing one of those values, allows the agent to move more smoothly.

Since the environment was originally designed to handle continuous actions, a function to convert discrete actions in continuous was implemented, as shown in the listing below (4.4).

```
def __continuous_action(self, discrete_action):
    component_1 = (discrete_action // (self.movement_sensitivity *
        self.speed_sensitivity)
    component_2 = (discrete_action % (self.movement_sensitivity *
        self.speed_sensitivity)) // (self.speed_sensitivity)
    component_3 = discrete_action % self.speed_sensitivity

    return [(2*(component_1 / (self.movement_sensitivity-1))-1),
            (2*(component_2 / (self.movement_sensitivity-1))-1),
            (component_3) / float(self.speed_sensitivity-1)]
```

Listing 4.4: Converting discrete actions in continuous actions

4.1.4 Interacting with the environment

Initializing the environment When calling the method `reset`, the environment is set to its initial state: at each agent is assigned a random position within the spawn area and the same is done for the items. Even though the spawn position is always random, it is possible to pass a seed to the reset method. This allows to compare different setups in a fair way and replicate tests using the same initial state.

Updating the environment When the method `step` is called, the following procedure is executed:

1. the position of each agent present in the actions' dictionary is updated according to the action. Since the environment does not have boundaries, the agents can freely move as far as they want.

2. any item located within the action range of any agent is marked as collected.

3. the new observations are produced.

4. the agents' rewards are computed.

5. if all the items have been collected, `terminated['__all__']` is set to true, to notify who called the `step` method that the episode is over. This is the only possible way for an episode to end successfully; until this does not happen, all the agents keep performing actions.

However, an episode can also end when the number of steps reach `max_steps`. This parameter is passed during the environment creation through the `EnvironmentConfiguration` object.

Rendering the environment The base class `CollectTheItems` does not implement the method `render`. In order to be able to visualize the environment, the environment must be a `RenderableCollectTheItems` instance.

This class extends the base class by adding the possibility to visualize the current environment status through a canvas. At the object creation, at each agent is assigned a color and when calling the render method, both agents and uncollected items are shown.

Since the environment is unbounded, only a portion of it can be shown to the user. In particular, the visible region goes from the point (0,0) to the point (`spawn_area`,`spawn_area`). If an agent moves outside this area it can not be seen anymore until it does not move back to the visible area.

The image below (4.2) shows the evolution of the environment during the episode. The small red dots represent the items, while the other dots represent the agents. The big circle around each agent represents the agent's action range.

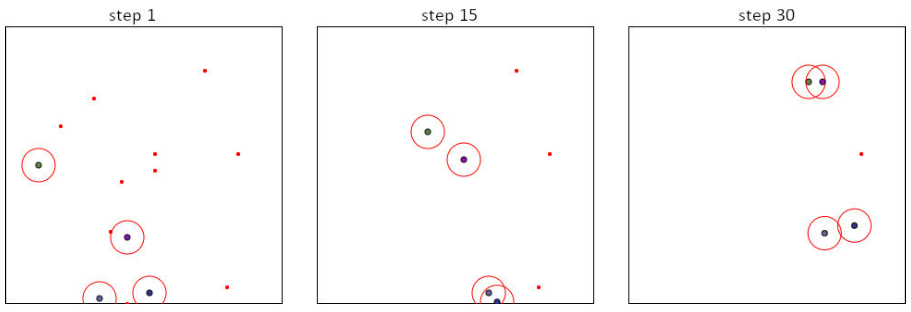


Figure 4.2: Evolution of the environment in 1, 15 and 30 steps

4.1.5 Reward structure

The reward of each agent is calculated as the sum of five components:

- Agents are penalized when their action areas intersect. If two agents' action areas intersect, both agent are penalized by a factor of -2, therefore this reward ranges from zero (no intersections) to -2*(`n_agents`-1) in case of multiple intersections.

  The idea of this component is to maximize the covered area avoiding multiple agents covering the same part of the environment.

- Agents are penalized at each step by a factor `step_penalty` defined in the environment configuration.

  The value of this penalization is therefore constant, but it can be updated using the `increase_step_penalty` method if adopting a curriculum learning approach.

- Agents are rewarded if they move toward one of the visible targets. In particular, is calculated the difference between the old and the new distance of the visible targets and the best value is taken and multiplied by a factor of three. If all the values are negative, this component is zero. Since each agent can move at most one unit at the time, the max value of this component is three.

- Agents are reward if they increase the distance from their neighbors. In particular, it is computed the difference between the new and the old distance between an agent and their `visible_nbrs`-closest agents. After that, is computed the average and the the result is divided by two. If the result is negative, the component is set to zero, therefore this reward can range between zero and one.

  This component encourages agents to move toward different target compared to their neighbors, allowing for better coordination strategies.

- Agents are rewarded for any visible item that has been collected whether they personally collected it or another agent did. The reward for collecting an item is 100, therefore this reward can range from zero to 100*`visible_items`.

  This strategy encourages the agent to develop a cooperative behaviour. On the opposite, if only the agent who physically collects the item was rewarded, competitive behaviors would emerge.

4.2 Experimental setup

The experiments were conducted using the version 2.22.0 of ray and the version 0.29.1 of gymnasium, while as underlying machine learning framework it was used torch (version 2.3). The source code can be found in the repository below[1].

To ensure a fair and accurate comparison of the different algorithms, each algorithm was initialized with the same seed, using the debug configuration parameter offered by ray. Four model instances were trained for each one of the

[1]`https://github.com/NicoloMalucelli/neighbor-based_MARL`

algorithms, using the seeds 2908, 3010, 911, and 2312. The final comparison between the algorithms was done using the average performance across the different runs.

In order to understand how this distributed strategies perform compared to centralized training models, a fully centralized approach was employed as a baseline. In this centralized setup, all agents were trained on a shared policy, allowing them to learn in a coordinated manner as if they were a single entity.

The environment configuration used to conduct the tests is the one shown below:

```
env_config = EnvironmentConfiguration(
    n_agents = 4,
    n_items = 10,
    spawn_area = 200,
    max_steps=300,
    agent_range = 3,
    visible_nbrs = 3,
    visible_items = 3,
    memory_size=3,
    movement_sensitivity=5,
    speed_sensitivity=5)
```

Listing 4.5: Environment configuration used in the experiments

Evaluation metrics To evaluate the effectiveness of the different learning strategies, two main metrics have been used: the average episode length and the average reward obtained by the agents. The average episode length highlights how quick the agents are to achieve their goal, while the average reward offers a measure of how well the agents are performing within the environment.

Each learning strategy was evaluated under three different aspects: performance on the experimental scenario, scalability and communication overhead.

To evaluate scalability, the number of agents and the size of the spawn area have been progressively increased and the performance metrics above described were evaluated again considering the new setup.

It is worth mentioning that during the scalability tests, the models were not trained again, nor fine-tuned. Each one of the realized tests is based on the models trained with the initial setup described in the listing above (4.5).

Algorithm configuration Each of the approaches discussed in this work was trained using a DQN configured with the same set of hyperparameters. The discount factor was set to 0.95, which strikes a balance between prioritizing immediate rewards and considering long-term gains. The learning rate, instead was set at 0.001, ensuring a gradual adjustment to the agent's policy.

The training batch size was set to 32, while the target network update frequency was set at 500 steps, and both Double Q-Learning and a dueling network architecture were enabled.

This combination of hyperparameters is the result of a large tuning and proved to be the best one for this specific task.

4.3 Results

4.3.1 Performance on the experimental scenario

The first test involved comparing the fully centralized method, where all the agents learn from the same policy, with the fully decentralized approach, where each agent maintains its own policy and does not share it with the other agents. This first comparison has been used as a baseline to compare the more sophisticated distributed methods.

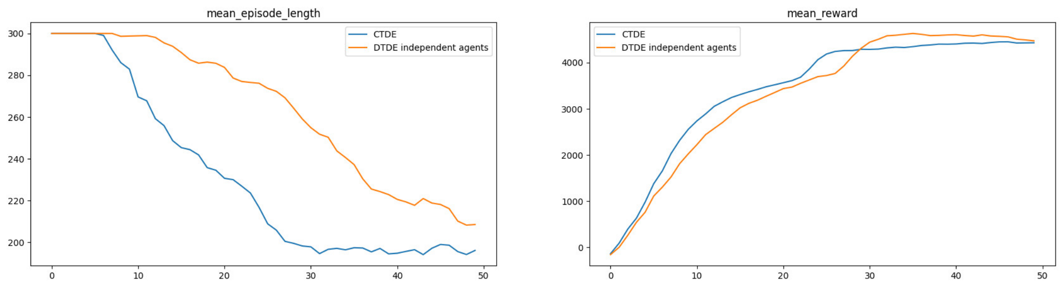


Figure 4.3: Performance comparison: independent learners vs centralized approach

The left graph (4.3) shows the average number of steps required to complete the task across different training iterations. The blue line represents the centralized training approach, while the orange line represents the fully decentralized approach. Initially, agents of both models do not manage to complete the task within the maximum number of steps specified in the environment configuration, therefore both graphs plateau at 300 steps.

As training progresses, the centralized approach shows a significant reduction in episode length, stabilizing at around 200 steps after only 30 training iterations. In contrast, the decentralized learning proceeds more slowly, and the average number of steps after the fiftieth training iteration is still above 210 steps.

NN-averaging The graphs below (4.4) show the performance of the NN-averaging learning strategy. As noticeable, the performance increases when considering an higher number of neighbors. In any case, each approach is better than the independent learners approach, proving the effectiveness of this strategy.

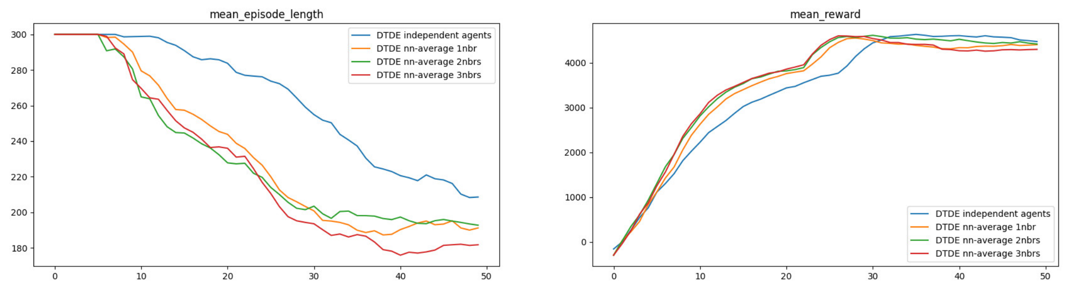


Figure 4.4: Performance comparison: NN-averaging vs independent learners

The effectiveness of the weighted average method has also been tested, using the average agent reward as factor for the mean. Nevertheless, this strategy did not yield any significant improvement over the base NN-averaging method and the performance remained essentially the same when compared with the standard NN-averaging approach for the same number of neighbors (image 4.5).

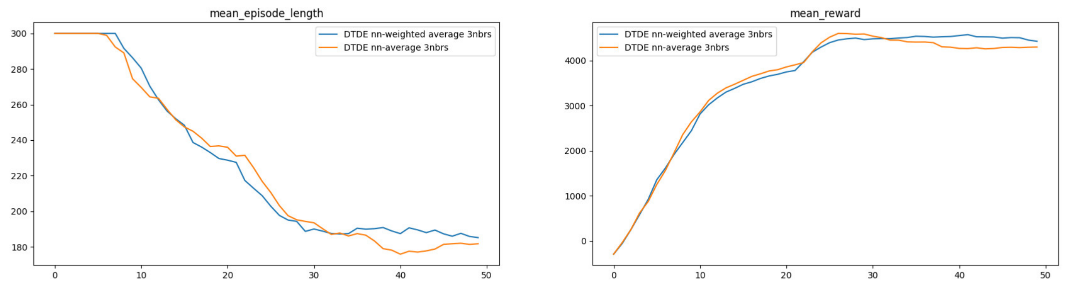


Figure 4.5: Performance comparison: NN-averaging vs NN-weighted-averaging

NN-consensus Differently from NN-averaging, NN-consensus seems to perform slightly better when the number of neighbors considered for the weight sharing is lower (image 4.6). This is probably due to the fact that when the number of neighbors is high, all the agents tend to adopt the same policy after a training iteration, reducing the exploration of different strategy. Nevertheless, at the end of the last training, all the configurations yield almost the same result.

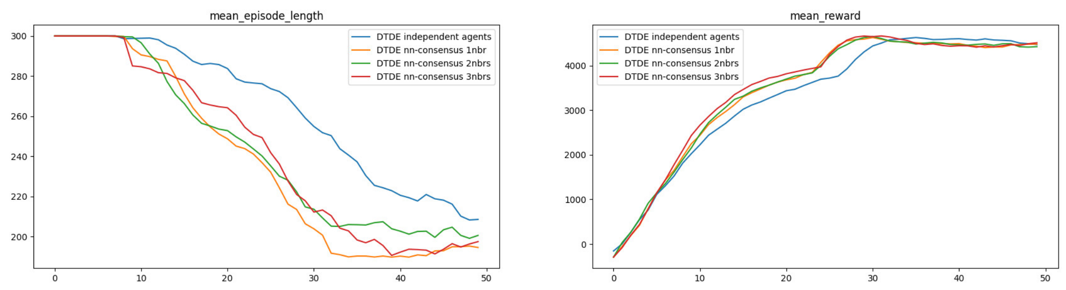


Figure 4.6: Performance comparison: NN-consensus vs independent learners

Experience sharing Like the previous strategies, experience sharing yields better results if compared to the independent learners approach. We can notice from the graph below (image 4.7) how the higher number of considered neighbors determines a faster learning in the initial phase of the training, while after a certain point the three settings tend to converge at the same average episode length. The reason why this happens is related to the size of the replay buffer used to store the experience. For instance, when the replay buffer has limited capacity, adding an extra neighbor, such as considering five neighbors instead of four, would not change the performance that much, because a small buffer would fills up quickly regardless of the number of neighbors, reducing the advantage of additional shared experiences over time. Using a prioritize replay buffer could help in this situation, allowing low value experiences to be discarded as the buffer gets full, and replaced by more valuable information shared by the neighbors.

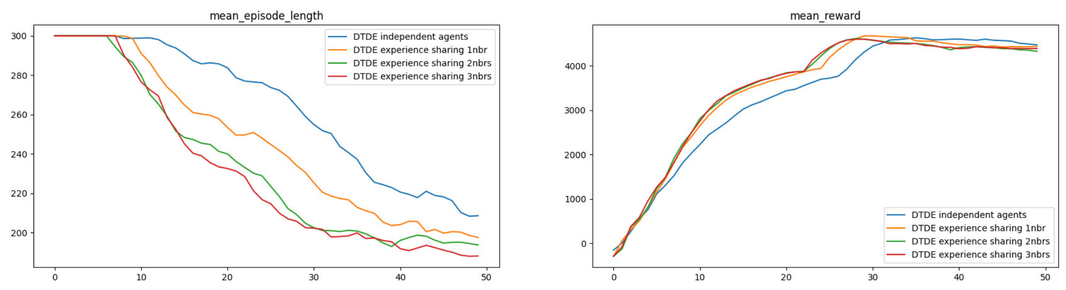


Figure 4.7: Performance comparison: experience sharing vs independent learners

Overall comparison The figure 4.8 compares the various strategies by the "episode length" metric, taking in account the average between the four training seeds, as anticipated.

The graph shows that three strategies perform almost the same: centralized training, NN-averaging with 3 neighbors, and NN-consensus with 3 neighbors, proving the effectiveness of the distributed training approaches. At the end of the fiftieth training iteration, the experience sharing method also converges to the same result of the best strategies even though the learning is slower especially in the middle phase.

The fully independent agents strategy, serving as a baseline for comparison, shows how much worse a distributed approach performs if no further strategies are adopted.

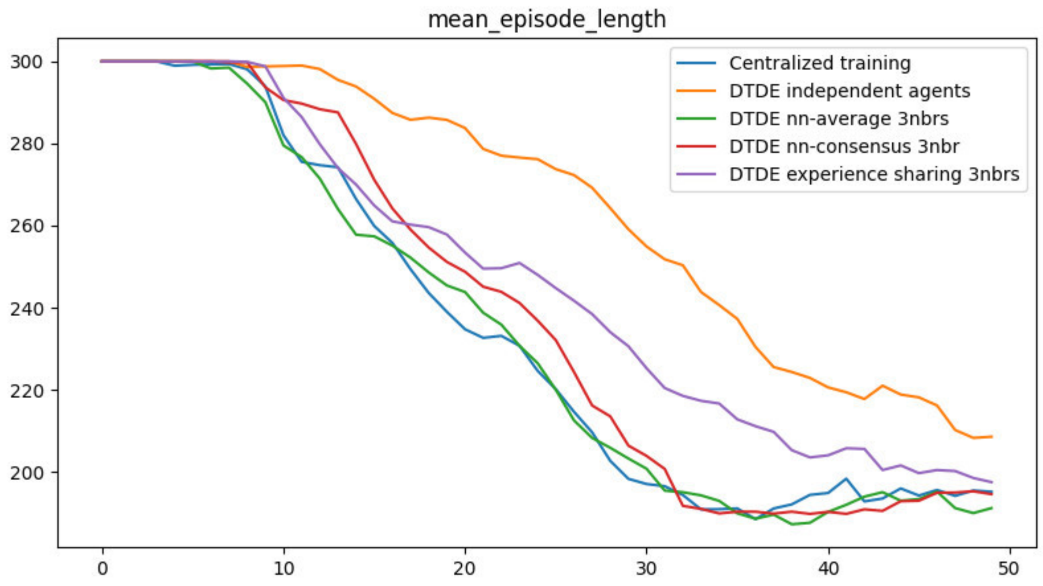


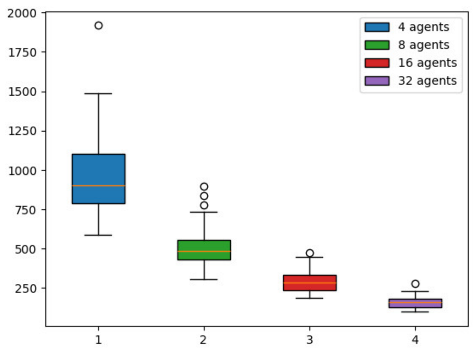
Figure 4.8: Performance comparison of the different learning strategies on the training case scenario

4.3.2 Scalability

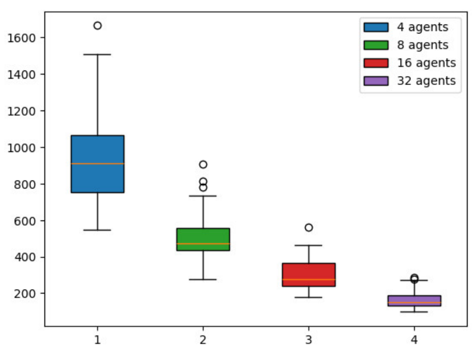
In order to evaluate the scalability of these approaches, different simulations were run using the previously trained models, but adopting different environment configuration. In particular, the spawn area was set to 500 and the number of target to 30. The the performance of the different policies was evaluated using 4, 8, 16, and 32 agents.

For each one of these simulation settings, 100 episodes were run, and the average and the standard deviation of the previously described metrics were computed. The results of these experiments are shown in the image below (4.9).

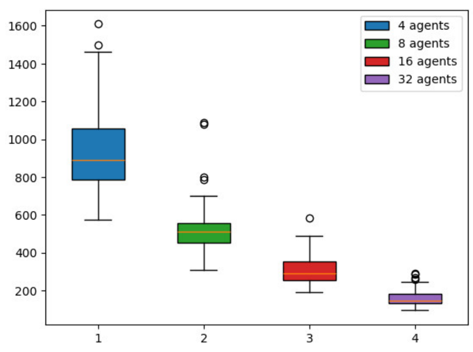
As it can be noticed from the graph, the average number of steps necessary to accomplish the goal decreases with the number of agents, following the same curve for each one of the different strategies.



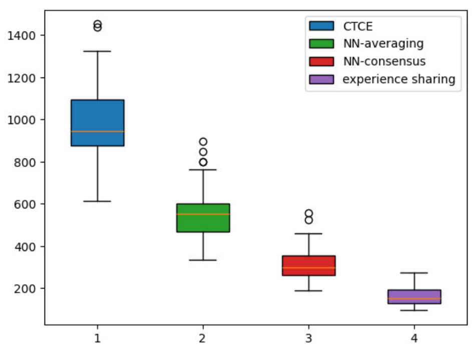
(a) centralized training



(b) NN-averaging



(c) NN-consensus



(d) experience sharing

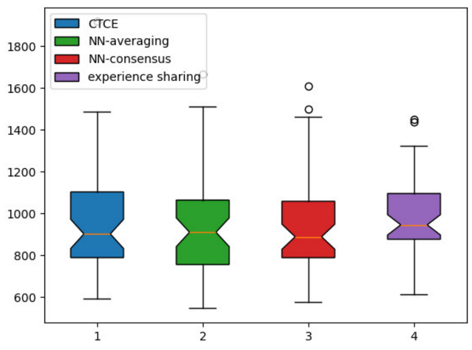
Figure 4.9: number of episodes to complete the task using the different strategies and a varying number of agents: 4, 8, 16, and 32.

It is worth to mention that since the distributed training of the model was done with four learning agents, the number of produced policies is four. Nevertheless, the number of agents in some of these experiments is higher than four, meaning that multiple agents adopt the same policy. This creates some sort of unintentional coordination between the agents having the same policy, in a similar way to the fully centralized approach.

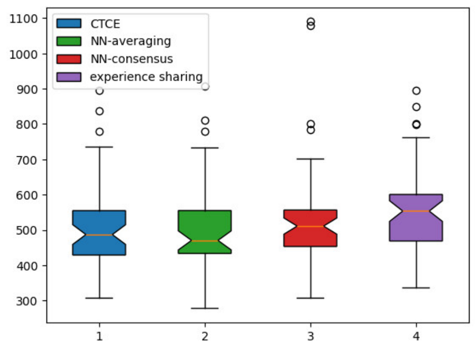
In particular, each agent i utilizes the policy $i\%4$, therefore in the 32 agents scenario, four agents are using the policy A, four the policy B, and so on.

This mechanism can actually be useful in scenarios in which new agents join the environment later. Instead of starting the learning from scratch, the new agent uses the policy of another agent as starting point for its training, avoiding that part of the training in which the agent performs bad.

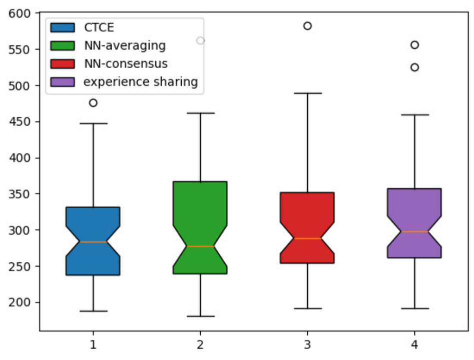
These box-plots (image 4.10) help visualizing the difference in performance between the different approaches when the number of agent increases. As it can be noticed, the performance are quite similar, confirming the results obtained before (graph 4.8). Nevertheless, some of these approaches seems to give slightly more confident results if compared to the others.



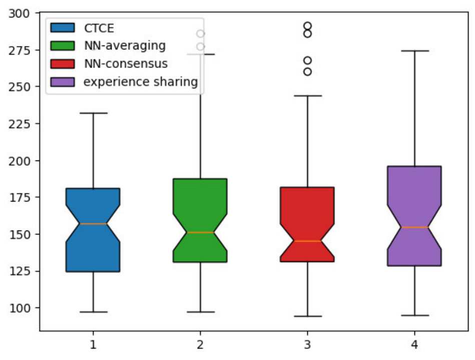
(a) 4 agents



(b) 8 agents



(c) 16 agents



(d) 32 agents

Figure 4.10: Comparing the scalability of the different learning strategies as the number of agents increases

4.3.3 Communication overhead

The communication overhead depends on a vast range of parameters, such as the frequency of the communication, and the number of considered neighbors; however, some general consideration can still be done based on the considered test case.

For the same communication setup (i.e., number of neighbors and frequency), this implementation of NN-consensus always requires less information sharing than NN-averaging. In NN-averaging, in fact, each agent has to receive the weights of everyone of its neighbors in order to be able to compute the average network. In NN-consensus, instead, the communication is divided in two phases: in the firth stage, each agent sends to their neighbors its average performance; then the agent selects among its neighbors the one having the highest performance and asks for its weights. By doing so, at each step, each agent receives just one set of network's weights, instead of one for each neighbors as it happens in NN-averaging.

Comparing experience sharing is more difficult since the information shared are not of the same kind as NN-averaging and NN-consensus; however, this strategy can reduce the communication load since the amount of information shared is normally smaller than the full model parameters. The communication overhead can still be significant if the agents have to frequently exchange large volumes of experience data. Some additional techniques can be used in order to reduce the amount of shared information; for instance, an agent may send to its neighbors only the experiences that are actually relevant, instead of the whole episode.

When evaluating experience sharing, an additional consideration must be taken in account: the security of the information. While in NN-averaging and NN-consensus, the information are embedded in the network weights, and therefore not interpretable, in experience sharing the agent directly shares tuples of the type $\langle state, action, state, reward \rangle$. This can be problematic in scenarios in which the information are valuable and must be kept secret.

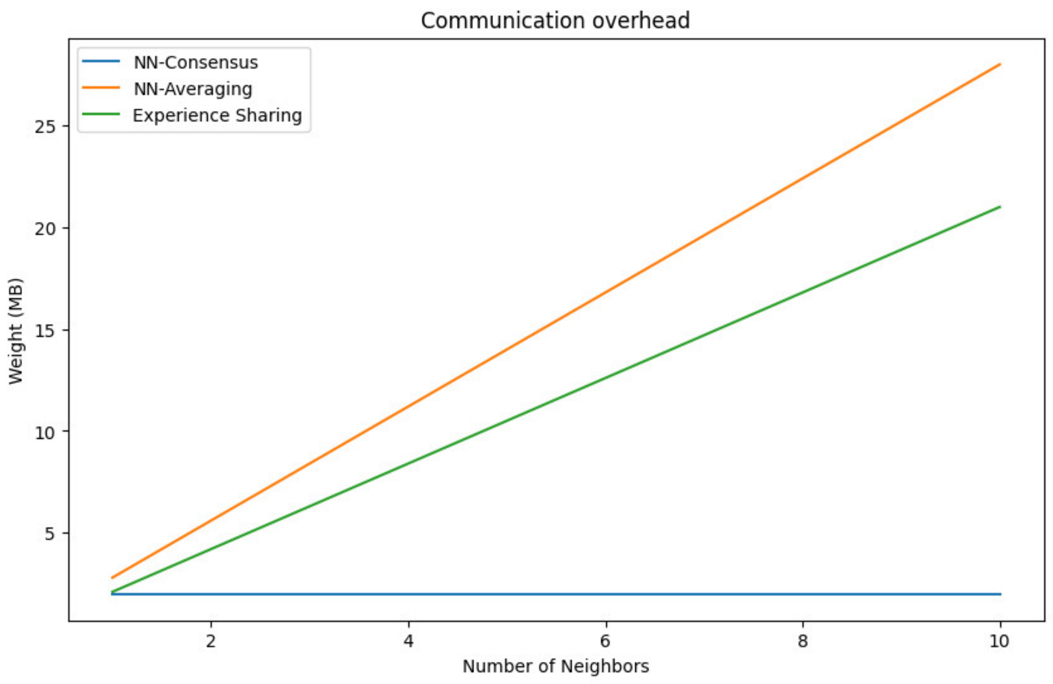


Figure 4.11: Amount of shared information across different training strategies and an increasing number of neighbors

The graph 4.11 compares the amount of information that each agent receives at each sharing step in relation to the number of considered neighbors. As shown, NN-consensus results in a flat graph, since the amount of information remains constant regardless of neighborhood size. In contrast, the communication overhead for NN-averaging and experience sharing increases with the number of neighbors, making these algorithms less efficient as the neighborhood size grows.

Conclusions

This thesis explored the application of Multi-Agent Reinforcement Learning in distributed environment with the main goal of finding a better and more scalable alternative to the centralized training approach, which suffers from scalability issues as the number of agents in the environment increases.

The work began by examining the theoretical foundation of reinforcement learning, starting with considering simple scenarios involving of a single agent, and then moving toward multi-agent scenarios.

To test the effectiveness of distributed learning approaches compared to the fully centralized scenario, a custom test environment was defined, and agents were trained using different distributed strategies: independent learners, NN-averaging, NN-consensus and experience sharing. Each one of these strategies was then compared with the fully centralized training, in which each agent shares the same common policy.

The evaluation showed how some of these strategies were able to reach the same performance of the centralized training approach in the considered test case scenario, outclassing the simplest form of distributed training: the independent learners approach. In particular, NN-averaging and NN-consensus are the ones that performed better, achieving very good results after only 35 training iterations. Experience sharing required more training iterations to achieve the same results as these methods, but on the other hand, it required lower communication between the agents. This shows that performance is strictly related to the amount of shared information: the more information is shared, the faster the learning becomes.

In conclusion, this thesis contributed to understanding how different MARL strategies can be effectively applied in distributed environments. While centralized approaches provide advantages in terms of coordination and efficiency, their practical limitations make distributed strategies more appealing for large-scale, real-world applications.

Future works could focus on evaluating the tested approaches across multiple environments, providing a more comprehensive understanding of how well the different MARL strategies generalize to a wider variety of tasks. It would also be valuable to explore the adaptation of policies in an online setting, such as through transfer learning, studying how well policies trained in one environment can be transferred and fine-tuned in a new unseen environment. Additionally, comparing other state-of-the-art algorithms, such as Proximal Policy Optimization (PPO), could provide better insights into the effectiveness of neighboring-based distributed learning methods.

Acknowledgements

This thesis represents for me the end of my academic journey, so I wish to thank all the people that supported me during this important stage of my life.

First and foremost, I would like to thank my family, because without you, none of this would have been possible. Not only you always let me free choice about what do to of my life, but you also did everything you could to help me achieve it, and I am really grateful for that.

I also want to thank all of my friends for having make my study path less difficult and more enjoyable. You have always been there when I needed you, and I will never be able to thank you enough.

Lastly, I would like to give a special thanks to all the people I met in Oulu. Even though we did not share that much time together, every moment was quality time: I lived in Oulu the best days of my life, and the credit is also yours, so thank you. I will never forget you.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Gianluca Aguzzi, Mirko Viroli, and Lukas Esterle. Field-informed reinforcement learning of collective tasks with graph neural networks. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 37–46. IEEE, 2023.

[3] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[4] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences*, 11(11):4948, 2021.

[5] Sven Gronauer and Klaus Diepold. Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review*, 55(2):895–943, 2022.

[6] Chaoyi Gu, Varuna De Silva, Corentin Artaud, and Rafael Pina. Embedding contextual information through reward shaping in multi-agent learning: A case study from google football. In *2023 IEEE 13th International Conference on Pattern Recognition Systems (ICPRS)*, volume 57, page 1–8. IEEE, July 2023.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[8] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[9] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[10] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[11] He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *International conference on machine learning*, pages 1804–1813. PMLR, 2016.

[12] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[13] Alan F. Karr. Chapter 2 markov processes. In *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, pages 95–123. Elsevier, 1990.

[14] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[15] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray rllib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 85:245, 2017.

[16] J F Mertens and Abraham Neyman. Stochastic games. *International Journal of Game Theory*, 10:53–66, 1981.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[19] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI}

applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[20] Brian Ning, Franco Ho Ting Lin, and Sebastian Jaimungal. Double deep q-learning for optimal execution. *Applied Mathematical Finance*, 28(4):361–380, 2021.

[21] G Papoudakis, F Christianos, A Rahman, and SV Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning. arxiv 2019. *arXiv preprint arXiv:1906.04737*, 2019.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[23] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

[24] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[25] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[26] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] Richard S Sutton. Reinforcement learning: an introduction. *A Bradford Book*, 2018.

[29] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[30] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[31] Alan R Washburn et al. Two-person zero-sum games. *International Series in Operations Research & Management Science*, 2014.

[32] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[33] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[34] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *International conference on machine learning*, pages 5872–5881. PMLR, 2018.

[35] Changxi Zhu, Mehdi Dastani, and Shihan Wang. A survey of multi-agent reinforcement learning with communication. *arXiv preprint arXiv:2203.08975*, 2022.