Reinforcement Learning: Training and Evaluating of Agents in a Graphical Environment

Vasiliki Vintzilaiou

Department of Information Systems

University of Piraeus

Submitted in partial satisfaction of the requirements for the
Degree of Master of Information Systems and Services
in Computer Science

Supervisor : Mr. Michael Filippakis

October, 2025

Statement of Originality

The work presented in this thesis/dissertation is entirely from the studies of the individual student, except where otherwise stated. Where derivations are presented and the origin of the work is either wholly or in part from other sources, then full reference is given to the original author. This work has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Statement of Availability

I hereby acknowledge the availability of any part of this thesis/dissertation for viewing, photocopying or incorporation into future studies, providing that full reference is given to the origins of any information contained herein. I further give permission for a copy of this work to be deposited to any repository authorised for use by University of Piraeus. I acknowledge that University of Piraeus may make the title and a summary of this thesis/dissertation freely available.

Signed: Vasiliki Vintzilaiou

Acknowledgements

I would like to express my sincere gratitude to my supervisor for the guidance, feedback, and support provided throughout the development of this thesis. Their advice and encouragement were invaluable during both the research and writing process.

I would also like to thank the faculty members and colleagues who contributed through discussions, suggestions, and technical support during the course of this work.

Finally, I am deeply grateful to my family, my partner Stelios, my close friends, and my cat Sporitos for their patience, understanding, and continuous support throughout my studies.

Abstract

This thesis investigates cooperative behavior in multi-agent reinforcement learning through a shared-policy formulation based on Proximal Policy Optimization (PPO). The study uses the Pistonball-v6 environment as a case study, where multiple agents operate under partial observability and decentralized execution while being controlled by a single shared neural policy. The objective is to examine how coordination emerges without learned communication channels, centralized control, or agent-specific role specialization.

The implementation is based on a convolutional actor–critic architecture trained with PPO using parameter sharing across all agents. The experimental analysis focuses on the effects of hyperparameter selection, rollout length, entropy regularization, fine-tuning strategies, and evaluation methodology. Multiple training configurations and random seeds were evaluated in order to examine the stability and reproducibility of cooperative behavior.

The results demonstrate that non-trivial coordination can emerge despite strong observational and architectural constraints. Several configurations were able to produce sustained cooperative behavior and maintain collective ball movement for extended periods of time. At the same time, the experiments revealed substantial sensitivity to initialization, stochasticity, and training dynamics, with policy performance varying considerably across seeds and evaluation settings. Additionally, in our experiments, stochastic evaluation consistently produced stronger and more representative behavior than deterministic execution, suggesting that stochasticity may contribute to behavioral adaptability in partially observable cooperative environments.

Overall, the findings indicate that shared-policy PPO constitutes a viable framework for studying coordination in cooperative multi-agent reinforcement learning.

Although the approach exhibits important limitations in terms of stability and reproducibility, the results demonstrate that meaningful cooperative behavior can emerge from relatively simple shared-policy architectures without explicit communication mechanisms.

List of Tables

List of Figures

Table of Contents

Chapter 1

Introduction

Reinforcement Learning (RL) has become a central research area within modern artificial intelligence, providing a framework in which agents learn to make decisions through trial-and-error interaction with their environment, Richard S. Sutton and Barto, 2018. Its formulation as a sequential decision-making problem has enabled substantial progress in domains such as robotics, autonomous systems, and simulation-based control, where actions must be optimized over time under uncertainty Bertsekas, 1987.

The integration of deep learning with reinforcement learning has further extended these methods to high-dimensional state spaces and complex control problems, allowing agents to operate directly on rich sensory inputs such as images Schulman, Wolski et al., 2017. This combination has significantly broadened the applicability of reinforcement learning beyond small, structured domains.

As reinforcement learning methods have matured, increasing attention has shifted toward settings involving multiple interacting decision-makers, where learning dynamics are influenced not only by the environment but also by the evolving behavior of other agents Hernandez-Leal et al., 2017. Such multi-agent systems introduce additional challenges, including non-stationarity, partial observability, and credit assignment, which complicate both training stability and performance evaluation Albrecht and Ramamoorthy, 2012.

Within this broader context, this thesis examines cooperative learning in physics-based environments through a shared-policy learning formulation, exploring how

coordinated behavior can arise when multiple interacting components are optimized toward a shared objective. By focusing on the dynamics of cooperation and stability under uncertainty, the work aims to provide empirical insight into the relationship between learning structure, coordination, and robustness in complex reinforcement learning settings.

Chapter 2

Theoretical Background

2.1 Fundamentals of Reinforcement Learning

2.1.1 Introduction of Reinforcement Learning

To begin this chapter, we will attempt to describe and explain reinforcement learning and its main features. To do so, let us first briefly place it within the broader field of machine learning.

Machine learning is a general class of solution methods designed to solve specific types of problem by learning from data. Within this field, we traditionally distinguish between two main types of learning: supervised and unsupervised learning. However, there is also a third category—reinforcement learning, our category—which follows a very different logic and is used to solve problems that the other two cannot easily address.

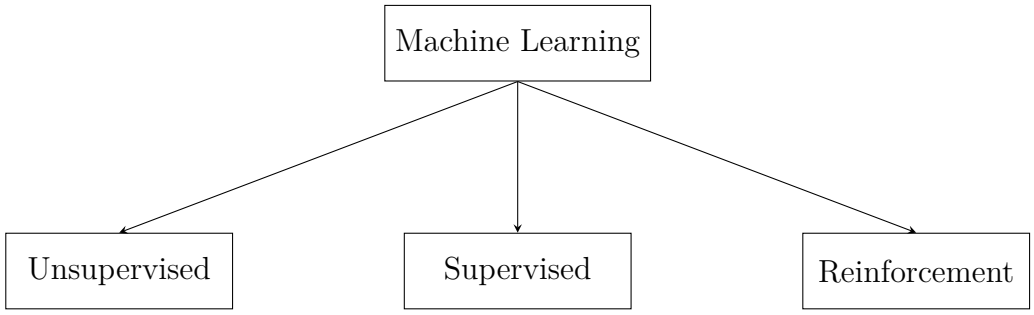

Figure 2.1: Basic ML categories.

Reinforcement learning (RL) is a method in which we have an agent that interacts with an environment and gradually learns which actions to take in order to achieve

a goal. The goal is predefined by us, and the agent must figure out how to reach it based on experience. Even from this basic description, we can already sense how different RL is from both supervised and unsupervised learning.

But there are also specific characteristics that clearly set RL apart as a learning approach:

1. Closed-Loop Problems

   The types of problem RL deals with are closed-loop problems. That means that the agent does not just act once and stop—it acts, observes the result, and acts again, in an ongoing loop. The goal is to achieve the best possible outcome by taking actions that lead to the highest total reward. This reward is not only about the next step—it also includes the long-term consequences of each action. To be able to calculate this future impact, the problem must have a structure that allows feedback and continuous interaction.

2. No Explicit Instructions

   RL learns from interaction without being told what to do in each situation. Unlike supervised learning, we do not give examples of the "right" action. In many environments, it would be impossible to define in advance how the agent should act in every possible scenario, especially when the future steps also depend on the current one. RL solves this by letting the agent learn from experience.

3. Uncertainty in Rewards

   A key challenge in RL is that we do not always know the reward of an action right away. Sometimes an action may seem bad in the short term but lead to a good outcome later on—or the opposite. This uncertainty about long-term results is another reason why RL problems are fundamentally different.

Now, let us compare this with the other two main types of learning to understand the distinction more clearly.

In supervised learning, the model is trained using a dataset of labeled examples, where each situation is matched with the correct answer or category. This kind of

learning dominates much of the current research in machine learning. The system is told: “In this situation, do this.” It does not interact with the environment—it only learns from the provided examples.

Although this method is very useful, it is static. Learning comes from fixed data and known outcomes. The model does not improve through experience or learn how to act in new, unseen situations without being told what to do. In contrast, in RL, the agent learns from its own actions and their consequences.

In unsupervised learning, there are no labels. The goal is to discover hidden patterns or groupings in the data. Because it lacks predefined answers, some might assume it is closer to RL. But the key difference is that unsupervised learning does not involve a goal. There is no feedback in the form of rewards, and the system does not act—it simply analyzes.

This absence of both interaction and goal-directed behavior is why unsupervised learning also does not cover the types of problem that RL addresses. In fact, it is this gap—the need to learn through experience, without being told what to do, while pursuing a goal—that created the need for this third approach.

Now that we have outlined this difference and introduced the logic of reinforcement learning, we can move toward a more formal definition:

Reinforcement Learning (RL) is a computational framework in which an agent interacts with an environment by taking actions in order to maximize a notion of cumulative reward over time. The process is typically formalized through a sequence of observations, actions, and rewards. Richard S. Sutton and Barto (2018)

2.1.2 Elements of Reinforcement Learning

From here, we will continue by analyzing the internal structure of reinforcement learning (RL) and its core components in more detail. To do so, we will use examples similar to those found in Sutton and Barto’s "Reinforcement Learning: An Introduction" , and then comment on the structure of the main elements: Agent, Environment, Reward, State, and Action.

Example 1 — Student Solving a Puzzle

A student working on a jigsaw puzzle places a piece in the wrong spot. This not only fails to complete that section, but also changes what pieces seem to "fit" later, possibly leading to more mistakes. Correct placements, on the other hand, open up new edges and reduce possibilities for the next move.

Example 2 — Playing Chess

A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counter replies—and by immediate, intuitive judgments of the desirability of particular positions and moves.

Example 3 — Mobile Robot Cleaning

A mobile robot decides whether to enter a new room to search for more trash or return to its battery recharging station. It makes this decision based on its current battery level and how quickly and easily it has been able to locate the charger in the past. (Sutton and Barto)

In all these examples, we have an agent , a learner, or a decision maker who aims to achieve a goal (completing the puzzle, winning the chess game, cleaning rooms) while interacting with something called the environment . The student does not yet know how to solve the puzzle, the chess player does not know what move the opponent will make, and the mobile robot must periodically check the rooms for trash while also monitoring its battery level.

It is important to note that the agent's actions change the state of the environment, which in turn affects the options available in future steps. Every snapshot of the environment perceived by the agent is called a state. For example, when a chess game begins, the initial arrangement of pieces defines the starting state. After the first move, the board changes, resulting in a new state, and the agent must choose its next action based on this updated information.

Each action taken by the agent moves it closer to or further from its goal. If it moves closer, we say that the action produced a positive reward (e.g., capturing an opponent's piece in chess, correctly placing a puzzle piece, cleaning a room). The

agent will reach its goal only if it consistently selects actions—based on current state—that produce higher rewards and lead to the goal.

Finally, we must emphasize that the reward from an action is not valued solely for its immediate effect, but also for its influence on future states. For example, in chess, a move might sacrifice a piece now but set up a winning strategy later; in the puzzle, placing a tricky corner piece early may make future placements much easier; for the robot, returning to recharge at the right time ensures it can continue cleaning in the future. This involves both planning and forecasting.

How we define the problem plays a crucial role, as we must identify the agent and decide what will be modeled as the environment in order to achieve the goal. In the chess game and puzzle, this distinction is straightforward, but in the cleaner robot example the boundaries are more complex — the environment is "not necessarily only what is outside of a robot or organism. The battery of the example robot is part of the environment of its controlling agent, it is a feature of the environment of its internal decision-making agent" (Sutton and Barto, 2018). In many cases, the state of the environment of an agent might also include aspects of the machine or organism itself, such as its internal status, stored memories, or even projected goals.

We can now clearly see that what reinforcement learning methods aim to do is find the best way to forecast and plan so that the agent chooses actions which, over time, will achieve the goal as quickly and as consistently as possible.

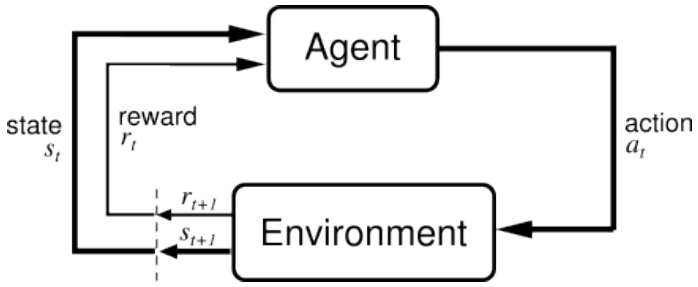

Figure 2.2: Illustration of a Reinforcement Learning Process.

There are some more specific subelements that define—and are, at the same time—the heart of how agents work. These subelements are the policy, the reward signal, the value function, and optionally a model of the environment.

A policy defines the way the learning agent behaves at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states (Sutton and Barto). We can think of it as the agent's response to a stimulus (with the state being the stimulus). A policy can be a simple function or involve extensive computation, depending on the set of actions available to the agent. The policy is at the core of reinforcement learning in the sense that it determines the behavior of the agent. In general, policies may be stochastic. However, the behavior of the agent can change over time. What can change the behavior of the agent?

We talked about rewards earlier in this section, but now we will give a more detailed definition. A reward signal defines the goal of a reinforcement learning problem. In each time step, the environment sends to the reinforcement learning agent a single number—a reward. The agent's sole objective is to maximize the total reward it receives over time. The reward signal thus defines what the good and bad events are for the agent (Sutton and Barto). We can think of a positive result as pleasure and a negative result as pain. The important point is that the only way for an agent to obtain a reward, as described above, is through its actions. When the agent acts, two things happen: the state changes (the environment changes), and the environment returns a reward signal (pleasure or pain). If this action produces a low reward, the policy may change so that some other action is chosen in the future for a similar or identical state.

This brings us to our third definition: the value function. In general, a value function also specifies what is good or bad, but in the long run. If an agent chooses an action, the reward acts as an immediate signal indicating whether it is good or bad, but the value function also estimates the total amount of reward the agent can expect from that action in the future as well. The reward is immediate, but the value function must forecast the states likely to follow and the rewards available in those states. Without rewards, there would be no value function, and the purpose of the value function is to achieve more rewards. To sum up: actions (and therefore policies) are based on estimated values, which are themselves based on rewards. As you can

imagine, the difficult part of this problem is predicting the value of an action in an unknown environment.

The fourth and final element—though not always necessary in reinforcement learning—is the model. A model mimics the behavior of the environment after selecting an action in a specific state, allowing it to predict the following state and reward. Models are used for planning and to help select actions that will bring higher rewards in the future. Methods that use models are called model-based and are generally more complex than model-free methods, which are usually based on trial-and-error learning.

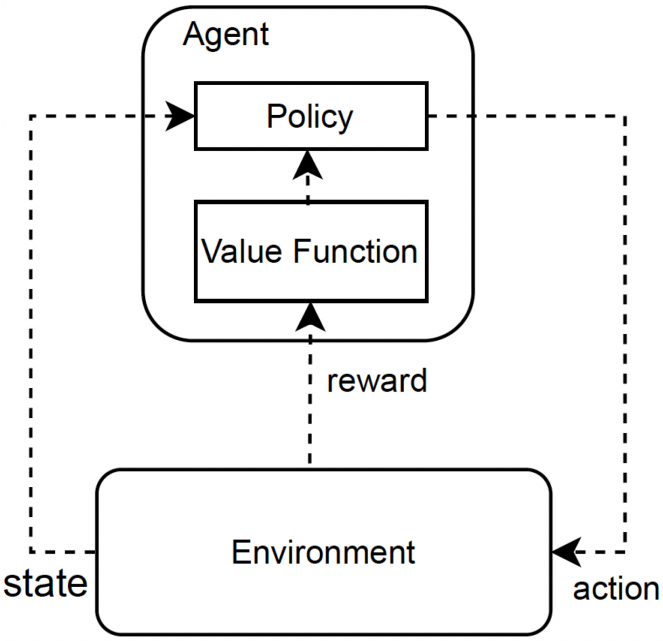

Figure 2.3: Illustration of a Reinforcement Learning Process.

2.1.3 Mathematical Framing of RL

So far, we have described reinforcement learning with words and examples. To move toward a more formal description, let us introduce a simple mathematical notation.

We imagine the interaction between the agent and the environment as occuring in a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step:

- The agent observes the current **state** $S_t \in \mathcal{S}$.

- Based on this state, it chooses an **action** $A_t \in \mathcal{A}(S_t)$, following its policy π.
- The environment responds by giving a **reward** $R_{t+1} \in \mathbb{R}$ and by moving to a new **state** S_{t+1}.

This produces a trajectory of interaction:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

The agent's objective is to choose actions in a way that maximizes the total reward it can expect over time. We call this total the **return**, defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $0 \leq \gamma \leq 1$ is the **discount factor**, which makes future rewards count a little less than immediate ones.

Policy. The agent's behavior is defined by a **policy** π, which is a mapping of states to probabilities of actions:

$$\pi(a|s) = \Pr(A_t = a \mid S_t = s).$$

Value Function. A central concept in RL is the idea of predicting future reward. The **state-value function** $v_\pi(s)$ under policy π is the expected return starting from state s and then following π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s].$$

Action-Value Function (Q-function). Similarly, the **action-value function** $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a].$$

These value functions provide a way to compare different actions or states in terms of their long-term desirability. They are the mathematical version of the "value" idea we introduced earlier.

Model (optional). Finally, some methods also use a **model** of the environment, which specifies the transition probabilities and rewards:

$$P(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a),$$

$$R(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a].$$

Models are useful for planning because they let the agent simulate possible futures without directly experiencing them.

This mathematical framework allows us to describe RL problems more precisely. However, to make them solvable, we need one more important assumption: that the current state contains all the information needed to predict what will happen next. This assumption is called the **Markov property** and leads directly to the idea of a **Markov Decision Process (MDP)**.

2.1.4 Markov Decision Process (MDP)

In this subchapter we are going to describe the concept of Markov Desicion Proccess and why is it so crucial for the reinforcement learning scientific field.

Based on the definitions we gave previously, agent decides in part on experience—on the past states, and in part on the actions it has taken. In tasks with many steps, or in continuous problems, this would mean relying on an enormous amount of historical data, which would be impossible to handle explicitly. The real question then is: *which details of a state should we keep*? The answer is given by the Markov property: a state is Markov if it retains all the information necessary to predict the future.

For example, think of navigating a maze. The agent's current position is a Markov state because it summarizes everything important about the path that led there. The details of the path are lost, but all that matters for future decisions is preserved in the current position. This is sometimes described as an *"independence of path"* property (Richard S. Sutton and Barto, 2018), meaning that the significance of the current state does not depend on the exact history of events that preceded it.

Formally, Richard S. Sutton and Barto (2018) define the dynamics of environment as follows. In the most general case, the next state and the reward at time $t+1$ could depend on the entire history:

$$\Pr\{R_{t+1} = r,\ S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}.$$

However, if the state has the Markov property, then the response of environment at $t+1$ depends only on the state and action at time t:

$$p(s', r \mid s, a) = \Pr\{R_{t+1} = r,\ S_{t+1} = s' \mid S_t = s, A_t = a\}.$$

In simple words, with the Markov property we can decide the next action using only the current state since it already captures all relevant history we need. Thus, the best policy defined over Markov states is just as good as one defined over complete histories. Even when tasks are not strictly Markov, reinforcement learning often treats them as approximate MDPs.

Definition. A reinforcement learning task that satisfies the Markov property is called a *Markov Decision Process (MDP)*. If the state and action spaces are finite, it is a *finite MDP*, which is central to reinforcement learning theory and underlies most modern applications (Richard S. Sutton and Barto, 2018).

The transition dynamics of an MDP are given by:

$$p(s', r \mid s, a) = \Pr\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s, A_t = a\}.$$

From these dynamics, we can derive useful quantities:

- **Expected rewards for state–action pairs:**

$$r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a),$$

- **State-transition probabilities:**

$$p(s' \mid s, a) = \Pr\{S_{t+1} = s' \mid S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a),$$

- **Expected rewards for state–action–next-state triples:**

$$r(s, a, s') = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{r \in \mathcal{R}} r\, p(s', r \mid s, a)}{p(s' \mid s, a)}.$$

We can now understand how crucial it is for a problem to satisfy the Markov property. This assumption greatly simplifies the computation of value functions and policies. When it does not hold, the problem becomes partially observable and significantly more complex; however, such cases are beyond the scope of this work and will not be analyzed further.

2.2 Key Reinforcement Learning Algorithms

So, based on the introduction to reinforcement learning and the description of its elements, we are now ready to describe how the agent actually learns to reach the goal defined in each reinforcement learning problem. The answer lies in algorithms. In a simplified way, algorithms either estimate the value function to induce a better policy (value-based methods) or optimize the policy directly (policy-based methods). The two perspectives are fundamentally connected through the Bellman equations.

To introduce these ideas, we will first assume that **the transition probabilities between states are known** ,that is because we have a complete model of the environment. Under this assumption, reinforcement learning can be framed within the dynamic programming paradigm, where the Bellman (1956) equations provide the formal link between value functions and policies. If these equations could be solved exactly, the learning problem would be solved in principle. In reality, the model is rarely available, and even if it were, solving the Bellman equations exactly is computationally infeasible for large or continuous state spaces.

The **Bellman expectation equation** for a policy π is:

$$v_\pi(s) = \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\Big],$$

The value of a state s is the expected immediate reward plus the discounted value of the next state, if the agent follows policy π.

Expanding the expectation, we have:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big].$$

Here: $\pi(a|s)$ is the probability of taking action a in state s, $p(s', r|s, a)$ is the probability of moving to s' and receiving reward r, and γ is the discount factor.

For state–action pairs, the **action-value function** is:

$$q_\pi(s, a) = \mathbb{E}_\pi\Big[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\Big].$$

This expresses the value of taking action a in state s, and then following policy π afterwards.

Replacing the expectation with a maximum leads to the **optimality equations**. For states:

$$v_*(s) = \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \gamma v_*(s')\Big],$$

The optimal value of state s is the best possible expected return over all actions a.

And for state–action pairs:

$$q_*(s, a) = \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big].$$

The optimal action-value $q_*(s, a)$ equals the expected reward plus the discounted value of the best action at the next state.

These optimality equations define the basis for dynamic programming methods such as value iteration and policy iteration.Presenting the value iteration algorithm in pseudocode helps to translate these equations into a concrete, step-by-step procedure. This not only illustrates how the update is applied in practice but also clarifies the link between theory and implementation.

Value Iteration

The *value iteration* algorithm is based directly on the Bellman optimality equations. At each iteration, it updates the value of every state by applying the Bellman optimality operator:

Algorithm: Value Iteration

1. Initialize $v_0(s)$ arbitrarily for all $s \in \mathcal{S}$.
2. Repeat until convergence:
$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big], \quad \forall s \in \mathcal{S}.$$
3. Derive the optimal policy:
$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_*(s')\Big].$$

Value iteration is a simple iterative algorithm that has been shown to converge to the optimal value function v^* and the corresponding optimal policy π^* Bellman, 1957; Bertsekas, 1987.

Policy Iteration

The *policy iteration* algorithm manipulates the policy directly, rather than finding it indirectly via the value function. It alternates between policy evaluation and policy improvement:

Algorithm: Policy Iteration

1. Choose an arbitrary initial policy π_0.

2. Repeat until convergence:

   (a) **Policy Evaluation:** Compute the value function of π_k by solving the Bellman expectation equations:

$$v_{\pi_k}(s) = \sum_{a \in \mathcal{A}} \pi_k(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_{\pi_k}(s')\Big].$$

   (b) **Policy Improvement:** Update the policy by acting greedily with respect to v_{π_k}:

$$\pi_{k+1}(s) = \arg\max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_{\pi_k}(s')\Big].$$

The policy iteration alternates between evaluating the current policy and improving it. This iterative process is guaranteed to converge to the optimal policy π^* after a finite number of iterations in the case of finite MDPs.

In summary, dynamic programming methods such as value iteration and policy iteration demonstrate how the Bellman equations can be used to compute optimal policies when the environment model is fully known. However, in most real-world reinforcement learning problems, the transition dynamics and reward function are not available. This motivates the development of algorithms such as temporal-difference learning, Monte Carlo methods, Q-learning, and SARSA which approximate these equations directly from experience. Next, we are going to present two important algorithms for the evolution of value based approach: Q-learning and SARSA.

2.2.1 Value-based methods: Q-learning and SARSA

Value-based reinforcement learning methods aim to estimate value functions and then derive policies from them. Instead of directly parameterizing and optimizing a policy, these methods focus on learning the expected return of states or state-action pairs, and use these estimates to guide decision-making. Both Q-learning and SARSA belong to the family of temporal-difference (TD) methods (R. S. Sutton, 1988). Temporal-Difference (TD) learning combines the strengths of Monte Carlo methods—learning directly from experience without requiring a model—with

the bootstrapping approach of dynamic programming, where current estimates are updated based on other learned estimates. In practice, this means that we update our estimates after every transition using the information from the *next step* — the immediate reward plus the discounted estimate of the next state — to improve the current estimate. This approach allows the agent to learn directly from the raw experience without waiting until the end of an episode.

Q-learning

Q-learning learns an action–value function $Q(s, a)$, which estimates the expected discounted return of taking action a in state s and then following the optimal policy. The update rule is as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big).$$

Watkins, 1989; Watkins and Dayan, 1992.

The term

$$r + \gamma \max_{a'} Q(s', a')$$

is termed *target*, as it represents an improved estimate of the true value of $Q(s, a)$ by combining the immediate reward with the discounted value of the best possible action in the next state. The difference between the target and the current estimate $Q(s, a)$ is the temporal-difference error that drives the update. The key idea is that information from the next step is used to improve the current estimate of the present step, a process known as *bootstrapping*.

The policy is derived indirectly by acting greedily with respect to the Q-values:

$$\pi(s) = \arg\max_{a} Q(s, a).$$

Q-learning is an off-policy algorithm: it updates as if the agent always selected the greedy action, regardless of the action actually taken. In other words, the *behavior policy* (the one used to explore the environment, e.g. ϵ-greedy) can be different from

the *target policy* (the optimal greedy policy that Q-learning is learning about). This separation means that Q-learning can still converge to the optimal action values even if the agent explores in a non-optimal way during training.

SARSA

SARSA (State–Action–Reward–State–Action) is also a temporal-difference algorithm that, like Q-learning, learns an action–value function $Q(s, a)$. This function estimates the expected discounted return of taking action a in state s and then continuing with the policy currently being followed. The update rule is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma Q(s', a') - Q(s, a)\Big).$$

The target here is $r + \gamma Q(s', a')$, which combines the immediate reward with the discounted value of the *actual next action* a' chosen by the policy in state s'. The difference between this target and the current estimate $Q(s, a)$ is again the temporal-difference error, and as in Q-learning, the key idea is to use information from the next step to improve the current estimate (bootstrapping).

The policy is derived in the same way as in Q-learning, by acting greedily with respect to the learned Q-values:

$$\pi(s) = \arg\max_a Q(s, a).$$

The essential difference is that SARSA is an *on-policy* algorithm: it updates the Q-values using the action that the agent actually took, according to its current behavior policy (often ϵ-greedy). This means that SARSA learns the value of the policy that is really being followed, including both its greedy choices and its exploratory moves.

In contrast, Q-learning is off-policy: it always updates as if the greedy action had been chosen, even if the agent in practice selected a different action during exploration. Therefore, while both methods use the same principle of temporal-difference

learning and bootstrapping, SARSA tracks the performance of the *current exploratory policy*, whereas Q-learning aims directly at the *optimal greedy policy*.

2.2.2 Policy-based methods

REINFORCE

As an introductory example of policy-based methods, we consider the REINFORCE algorithm **1987**; Williams, 1992, one of the earliest and most influential policy gradient methods. Formally, policy π_θ is a function mapping states to a probability distribution over actions, parameterized by weights θ. What differentiates REINFORCE from the previously discussed value-based algorithms (such as Q-learning and SARSA) is that it is based on the Monte Carlo principle rather than on temporal-difference (TD) learning. Most importantly, as we have emphasized before for policy-based algorithms, REINFORCE directly updates the policy function and does not rely on first finding the optimal value function and then deriving a policy from it.

In TD methods, updates are performed by *bootstrapping*, i.e., by using the immediate reward and the estimate of the next state value to update the current value function. By contrast, Monte Carlo methods wait until the end of the episode and use the complete return as feedback. The return from a given timestep t is defined as

$$G_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k},$$

that is, the discounted sum of rewards from timestep t until the end of the episode. It is important to note that G_t is a sample return from a particular trajectory: it is fully determined by the rewards observed in that episode and does not involve the parameters θ directly. However, the distribution of G_t depends on the policy π_θ, and thus ultimately on θ. Our objective is to maximize the *expected* return under the policy π_θ parameterized by weights θ:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t].$$

The central idea of REINFORCE is to adjust the policy parameters θ in the direction of the gradient of $J(\theta)$, which produces the update rule

$$\Delta\theta = \alpha(G_t - b)\,\nabla_\theta \log \pi_\theta(a_t|s_t),$$

where α is the learning rate. The update consists of two distinct factors. First, $(G_t - b)$ acts as the learning signal: G_t is the observed return from timestep t, while b is a baseline used to reduce variance without introducing bias. This term measures whether the chosen action performed better or worse than expected. Second, the gradient $\nabla_\theta \log \pi_\theta(a_t|s_t)$ specifies how the parameters should change to increase the probability of the action actually taken in state s_t.

Combining these terms, if an action a_t taken in state s_t led to a higher return than expected, the gradient update will increase its probability under the policy. In contrast, if the return was lower, its probability will be decreased. This way, the policy gradually shifts toward favoring actions that bring higher rewards.

Unlike TD methods, REINFORCE does not use bootstrapping and is therefore unbiased, but its updates tend to have high variance. Despite this limitation, REINFORCE represents the foundation of policy gradient methods and illustrates the essential idea of directly optimizing the policy. It can be seen as the starting point for a broader generalization of reinforcement learning, where parameterized policies are updated directly, paving the way for more advanced algorithms such as actor–critic methods.

2.2.3 Actor-Critic methods: foundations and intuition

In the previous chapters, we explained two of the three main groups of RL algorithms. For better understanding, we are going to refer to *value-based algorithms* as *critic-only* and *policy-based algorithms* as *actor-only*, and we are now going to introduce the third group, the *actor–critic algorithms*. The first two groups have both advantages and disadvantages, which we will mention briefly to explain why actor–critic algorithms are considered the most reliable ones for real-life applications at this

point. This also works as a good introduction to the next chapter, where we will present PPO which is also a strong actor–critic algorithm.

We call value-based functions *critic-only* because, at their core, they do not use a policy function directly. They instead try to reach the highest possible reward through the critic, by calculating the reward for each action and then choosing the best value to apply to the policy. We described this in the previous chapters using Bellman equations, with Q-learning and SARSA as examples. The problem with these solutions, according to Grondman et al. in *"A Survey of Actor–Critic Reinforcement Learning: Standard and Natural Policy Gradients"*, is that when we act based only on the value function, there is no guaranty for the near-optimality of the resulting policy when learning online. They also mention that Q-learning and SARSA, with specific function approximators, have been shown not to converge even for simple MDPs. This instability is largely attributed to the use of bootstrapped updates, where current estimates are updated based on other learned estimates ("estimation upon estimation"), which can lead to the propagation and amplification of approximation errors. In later research "Grondman et al., 2012", it is shown that with proper approximation functions, such as those that are linear in parameters, convergence can be ensured. Nevertheless, for most choices of basis functions, a value function learned using temporal difference methods will still be biased.

On the other hand, we have policy-based algorithms, which we describe as *actor-only*, because these algorithms update only the policy function and do not use any kind of stored value function. As we explained with Bellman equations, most actor-only algorithms work with a parameterized family of policies and try to optimize the cost directly over the policy space. As mentioned earlier, policy-based methods work much better than critic-only ones in problems with continuous action spaces. Their main advantage is their strong convergence property, which comes from gradient descent methods. However, their main disadvantage is that the estimated gradient can have a high variance (Grondman et al.).

Now, actor–critic methods — which are the focus of this chapter — as their name suggests, are a combination of actor-only and critic-only approaches. They keep the

strengths of both by learning a policy (the actor) and a value function (the critic) at the same time. In this new dynamic setup, we have two components that work together to solve the problems mentioned before and to combine the advantages of both sides. As Grondman et al. describe, "The actor is only responsible for generating a control input u, given the current state x. The critic is responsible for processing the rewards it receives, i.e. evaluating the quality of the current policy by adapting the value function estimate."

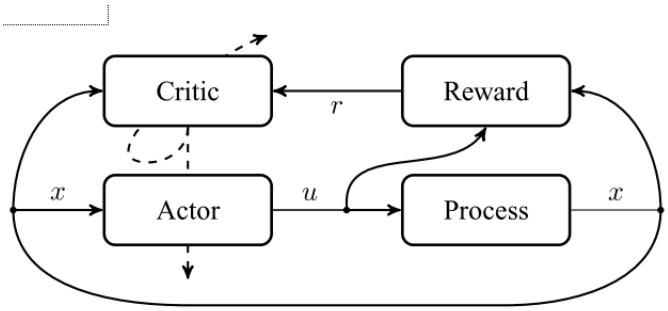

Figure 2.4: Schematic overview of an actor–critic algorithm. The dashed line indicates that the critic is responsible for updating the actor and itself (Grondman et al.).

Basically, the algorithm process goes like this: the critic evaluates the policy followed by the actor. Just like in critic-only methods, this can be done with the same algorithms we mentioned earlier, such as TD(λ). The critic approximates and updates the value function using samples. Then, this value function is used by the actor to update its policy parameters in the direction of the policy gradient, using only a small step size. This means that even if the value function changes significantly, the policy will only be slightly affected, ensuring a stable, non-oscillatory behavior. So, the choice of algorithms is still very important to achieve the expected outcome, and gradient policy methods play a key role in this whole learning structure. A critical aspect of actor–critic training is the relationship between the learning rates of the two components. The critic must learn faster than the actor in order to provide an accurate value estimate before the actor updates its policy. If the actor updates too quickly, it may rely on inaccurate feedback and cause divergence. Balancing these learning rates is essential for the overall stability and convergence of the algorithm.

The critic typically learns by minimizing the temporal-difference (TD) error:

$$\delta_t = r_t + \gamma V_\theta(s_{t+1}) - V_\theta(s_t),$$

which measures how much the predicted value differs from the observed return.

The actor updates its policy parameters using the policy gradient, guided by the temporal-difference error:

$$\theta \leftarrow \theta + \alpha\, \delta_t\, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \tag{2.1}$$

Here, δ_t is the TD error provided by the critic, which serves as an estimate of the advantage. The term $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ represents the direction in which the policy parameters should be adjusted. This update increases the probability of actions that yield higher-than-expected returns and decreases the probability of less favorable actions.

Characteristic	**Value-based**	**Policy-based**	**Actor–Critic**
Learns policy directly?	No (indirectly via values)	Yes	Yes
Learns value function?	Yes	No	Yes
Continuous actions	Difficult	Yes	Yes
Training variance	Low	High	Medium
Examples	Q-learning, SARSA	REINFORCE	PPO, A2C

Table 2.1: Comparison of Value-based, Policy-based, and Actor–Critic methods.

The effectiveness of actor–critic methods largely depends on the quality of the policy updates. In practice, the actor is typically updated using the standard policy gradient, where the parameters follow the direction of steepest ascent in the parameter space. Although simple and widely applicable, this approach ignores the structure of the policy distribution, which may result in inefficient or unstable updates, especially when combined with function approximation and noisy value estimates.

To mitigate these issues, natural gradient methods incorporate information about the geometry of the policy space, leading to more stable and well-scaled updates. Nevertheless, standard policy gradients remain prevalent due to their simplicity and computational efficiency. Overall, actor–critic algorithms form the foundation of

many modern reinforcement learning methods, combining value estimation with direct policy optimization. Despite their strengths, they remain sensitive to update instability and hyperparameter choices, motivating the development of more robust approaches such as *Proximal Policy Optimization (PPO)*, which will be examined in the following chapter.

2.3 Proximal Policy Optimization (PPO)

In the previous chapters we described the core logic and milestones in the evolution of reinforcement learning algorithms, reaching as far as policy gradient methods and the introduction of actor–critic architectures. The purpose of this historical trajectory was to reach a position from which we can introduce the main algorithmic focus of this thesis: the Proximal Policy Optimization (PPO) family of algorithms. PPO was introduced in 2017 in the paper *Proximal Policy Optimization Algorithms* by Schulman et al. In essence, PPO can be viewed as a simpler and more practical variant of TRPO, which retains the main idea of constraining policy updates to trusted regions of the parameter space, but avoids the computationally expensive constrained optimization of TRPO.

2.3.1 Motivation and Advantages

PPO belongs to the class of policy gradient and actor–critic methods. As discussed previously, such methods consist of two main components: the **actor**, which determines the actions taken by the agent (in our case, a stochastic policy $\pi_\theta(a_t \mid s_t)$ parameterized by θ), and the **critic**, which estimates the value of states or state–action pairs (using a value function $V_\phi(s_t)$ parameterized by ϕ). The goal of training is to progressively update the parameters θ and ϕ through gradient-based optimization so that the agent achieves a higher cumulative reward, which is indicative of successful learning.

In classical policy gradient methods, the gradient estimator takes the form:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \hat{A}_t \right],$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t .Schulman, Wolski et al., 2017. The estimator $\hat{g}$ can be obtained by differentiating the surrogate objective

$$L_{\mathrm{PG}}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t \mid s_t)\, \hat{A}_t \right].$$

Schulman, Wolski et al., 2017

Although this approach successfully addressed limitations of previous methods, it presents new issues: performing multiple optimization steps on L_{PG} using data from the same trajectory is not theoretically well-justified and empirically often leads to destructively large policy updates. PPO introduces two modifications that significantly improve stability and sample efficiency: (1) minibatch stochastic optimization over collected trajectories, allowing data to be reused more efficiently, and (2) a clipped surrogate objective together with Generalized Advantage Estimation (GAE), which provides a more robust advantage estimator. This prevents the policy from drifting too far from the behavior policy that generated the data, reducing variance, and avoiding training collapse.

These design choices make PPO a stable and easy-to-tune algorithm that typically requires minimal hyperparameter engineering while maintaining a favorable balance between exploration and exploitation. In the next subsection, we describe these components in more detail and conclude with a schematic overview of the full PPO algorithm.

Generalized Advantage Estimation (GAE)

So PPO algorithms use GAE (Generalized Advantage Estimation) to provide a more robust advantage estimator. GAE is an estimator that measures "how good" an action was for a given state and time compared to the expected value of that state.

It achieves this by combining the critic's value estimates with a multi-step method to compute the advantage without explicitly estimating Q. Instead of predicting Q directly, PPO computes A_t using the actual rewards from the environment together with the state values $V(s_t)$ that the Critic estimates with its value network. In

this way, the Critic provides the expected return baseline $V(s_t)$, while the rewards along the trajectory provide the realized return. The difference between these two quantities tells us whether the action performed better or worse than expected, consistent with the original definition $A = Q - V$.

In traditional actor–critic methods this difference can be computed in various ways (e.g., one-step TD, n-step returns, or Monte Carlo returns), each with different bias and variance trade-offs. GAE in PPO does not commit to a single choice. Instead, it forms a weighted combination of multiple k-step return estimators using a decay parameter λ. This results in the following multi-step advantage estimator:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal-difference residual, which measures how much better or worse the outcome at time t was compared to what the Critic had predicted.Schulman, Moritz et al., 2016. When the action gives a better value than expected, $A_t > 0$, and when it does not, $A_t < 0$.

This gives a better balance between bias and variance and makes the training more stable. Once the advantage $\hat{A}_t$ is computed, PPO uses it to decide in which direction and by how much to adjust the policy parameters in the next update step. More specifically, the policy gradient term

$$\hat{A}_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

increases the probability of taking action a_t when $A_t > 0$, and decreases it when $A_t < 0$. The remaining part of the algorithm, the clipped surrogate objective, is responsible for controlling the magnitude of these updates during the optimization process.

2.3.2 Clipped Objective

The clipped objective is a key feature of PPO. The basic idea behind it is related to the TRPO algorithm. TRPO identified the problem of excessively large updates to the stochastic policy, which could be catastrophic for the learning process. In order

to control this behavior, TRPO introduced mechanisms that limited how far the new policy could move away from the old one during an update. It did this by using the probability ratio $r_t(\theta)$ in combination with the advantage A_t inside the surrogate objective, and by adding a constraint on the KL divergence between the two policies, effectively creating a "trust region" within which updates were considered acceptable.

While theoretically appealing, this approach requires solving a constrained optimization problem with second-order methods, which makes TRPO computationally expensive and difficult to tune in practice. PPO replaces this mechanism with a first-order clipped surrogate objective that achieves a similar trust region behavior without the need for explicit constraints or penalty tuning. In the following section we describe how this clipped objective is constructed inside PPO.

Borrowing the trust region idea from TRPO, PPO defines a new gradient objective that needs to be maximized. This objective is the clipped surrogate loss:

$$L_t(\theta) = \min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t\right)$$

Schulman, Wolski et al., 2017 where $r_t(\theta)$ is the probability ratio and $\hat{A}_t$ is the advantage computed using GAE as described earlier. The ratio is defined as:

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

Schulman, Wolski et al., 2017 and it measures how much more (or less) likely the new policy π_θ is to take the same action a_t in the same state s_t compared to the old policy $\pi_{\theta_{\text{old}}}$ that generated the data. If $r_t > 1$ the new policy increases the probability of that action, and if $r_t < 1$ it decreases it. Connecting this ratio with the advantage means that if the action was better than expected ($\hat{A}_t > 0$) the algorithm pushes the policy to increase its probability, and if it was worse ($\hat{A}_t < 0$) it pushes it to decrease it.

The clipped part of the objective says that if the update pushes r_t too far from 1, then it gets cut (clipped). In this way the new policy always remains close to the old one. The old policy is the one at the beginning of the update phase, and the

new policy is the one with the proposed parameter changes. The hyperparameter ϵ is typically set to 0.2 and controls how much deviation we allow, depending on how exploratory we want learning to be.

Controlling how far the new policy can move from the old one gives stability and also allows PPO to run several minibatch updates over the same sample data. This was not possible in pure policy gradient methods, where the policy would quickly become inconsistent with the data that generated the trajectories. Therefore, data efficiency is also a major benefit of the clipped objective.

2.3.3 Schematic Steps of the Algorithm

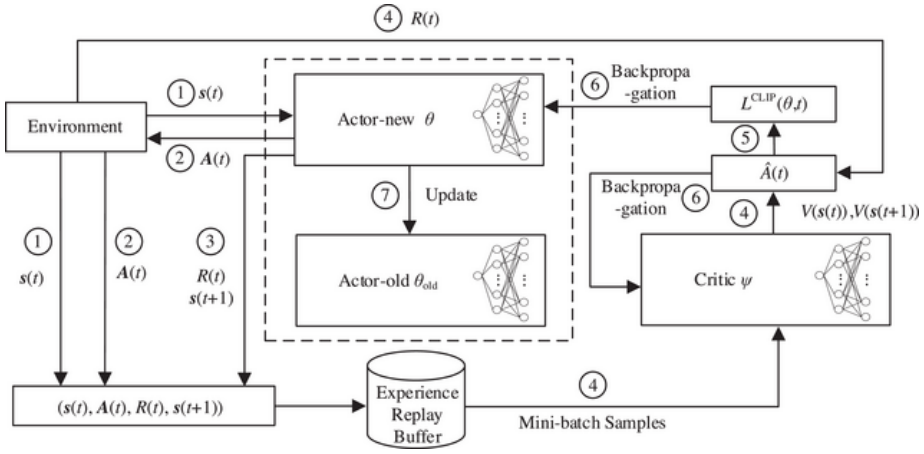

Figure 2.5: Schematic of proximal policy optimization PPO.Tan et al., 2023

Schematic Steps of the PPO Algorithm

The schematic representation of PPO in Figure 2.4 shows the full interaction loop between the environment, the Actor, and the Critic. At the beginning of each iteration, the agent collects trajectories by interacting with the environment using the current policy $\pi_{\theta_{\text{old}}}$, producing sequences of states, actions, rewards and next states. These samples are stored in a buffer and are later reused for optimization.

Using the stored trajectories, the Critic evaluates the value function $V_\phi(s)$ for each visited state. Based on these values and the observed rewards, the generalized advantage estimates $\hat{A}_t$ are computed. Once the advantage information is available, the Actor is updated by maximizing the clipped surrogate objective, while the Critic

is updated by regressing its value estimates toward the empirical returns. Both updates are performed using minibatch stochastic gradient optimization over the same collected data, typically for several epochs. After the updates are completed, the new policy parameters θ replace the old ones, and a new iteration of interaction with the environment begins.

Having presented the theoretical foundations of PPO, we now move to a more complex setting in reinforcement learning. In many real-world problems, multiple agents interact within the same environment, introducing additional challenges such as coordination, non-stationarity, and partial observability. The following section provides a brief introduction to Multi-Agent Reinforcement Learning.

2.4 Multi-Agent Reinforcement Learning

Deep learning has significantly advanced reinforcement learning, enabling the training of agents in large-scale continuous and discrete environments. It has also made it possible to move beyond single-agent settings and study scenarios where multiple agents interact. Many real-world problems naturally involve multiple decision-making entities, making multi-agent reinforcement learning (MARL) a more realistic modeling framework. Multi-agent reinforcement learning can be viewed as a natural extension of the standard Markov Decision Process (MDP) framework to settings with multiple interacting agents. However, unlike single-agent environments, MARL introduces additional structural complexities that depend on several key factors.

First, environments may be fully or partially observable, depending on whether agents have access to the complete state or only local observations. Second, agent interactions can be cooperative, competitive, or mixed, depending on whether agents share the same objective or have conflicting goals. Finally, agents may act either simultaneously or sequentially, which affects how decisions and outcomes are modeled.

Based on these characteristics, different frameworks have been proposed to formalize multi-agent problems. The simplest extension is the Markov Game (or stochastic game), which generalizes the MDP to multiple agents under full observability, where

all agents select actions simultaneously and the outcome depends on their joint actions.

When observability is limited, the problem is often modeled as a Partially Observable Markov Game (POMG), where agents receive incomplete information about the environment. A further extension is the Decentralized Partially Observable Markov Decision Process (Dec-POMDP), which captures fully cooperative settings under partial observability, where agents must coordinate based only on local observations and a shared reward signal.

In addition, sequential interaction settings can be modeled using extensive-form games, where agents take actions in turns rather than simultaneously. These different frameworks provide ways to model increasingly complex and realistic multi-agent environments, depending on the assumptions about observability, interaction, and decision timing.

In general, four key challenges are commonly identified in multi-agent reinforcement learning: computational complexity, non-stationarity, partial observability, and credit assignment. Computational complexity and non-stationarity arise in most multi-agent settings, while partial observability and credit assignment are particularly prominent in decentralized and cooperative environments.Wong et al., 2022.

Overall, these factors significantly increase the difficulty of learning in multi-agent systems. As a result, a variety of training paradigms and algorithmic approaches have been proposed to address these challenges.

In the following subsections, we examine these concepts in more detail.

2.4.1 New challenges in Multi-Agent Reinforcement Learning

Non-stationarity

In multi-agent environments, all agents interact with the environment simultaneously. As a result, the dynamics of the environment are not determined by the

actions of a single agent, but by the joint actions of all agents. This fundamentally differentiates multi-agent settings from the single-agent case.

From the perspective of an individual agent, the environment becomes non-stationary, since the policies of other agents are continuously changing during training. This violates the Markov assumption typically adopted in single-agent reinforcement learning, where the environment is assumed to be stationary and fully characterized by the current state Van Otterlo and Wiering, 2012.

As a consequence, the transition dynamics and reward distributions are no longer stable over time, making it more difficult for agents to learn accurate value functions and optimal policies. This issue is one of the central challenges in multi-agent reinforcement learning, and various training paradigms have been proposed to mitigate its effects.

A simple example of non-stationarity is illustrated in the Rock–Paper–Scissors game Papoudakis et al., 2019. Consider two agents selecting actions $a_i \in \{a_{paper}, a_{rock}, a_{scissors}\}$. Each agent aims to maximize its reward by adapting to the opponent's behavior. For instance, if one agent develops a preference for playing paper, the other agent will adapt by favoring scissors. In response, the first agent may change its policy again. This continuous adaptation leads to an evolving environment from each agent's perspective, which is the essence of non-stationarity.

Credit assignment

The credit assignment problem arises in multi-agent environments where multiple agents act simultaneously. The main challenge is that an agent cannot easily determine how much its own actions contributed to the joint reward signal. Since all agents act at the same time, it becomes difficult to identify whether a positive outcome is the result of a specific agent's action or the combined effect of multiple agents.

One possible approach is to assign local rewards to each agent. However, this can negatively affect cooperation, as agents may prioritize individual performance over the collective objective.

A closely related challenge is the construction of reward functions that encourage cooperative behavior. This is particularly challenging in environments with mixed incentives, such as social dilemmas. In these settings, agents may develop undesirable strategies. For instance, the lazy agent problem may arise, where one agent learns an effective policy while another agent reduces its effort, relying on the performance of the first agent instead Sunehag et al., 2018.

Partial observability in MARL

Partial observability arises from the fact that agents do not have access to the global state and must make decisions based only on their local observations. As a result, each agent has incomplete information about the environment, making the learning process more difficult.

While partial observability also exists in single-agent environments, it becomes significantly more challenging in the multi-agent setting. This is because the behavior of other agents is part of the environment dynamics. Consequently, an agent cannot easily determine whether a change in its observations is the result of its own actions or the actions of other agents Wong et al., 2022.

Such settings are often modeled as Decentralized Partially Observable Markov Decision Processes (Dec-POMDPs), where agents operate under partial observability and aim to maximize a shared team reward through decentralized decision-making.

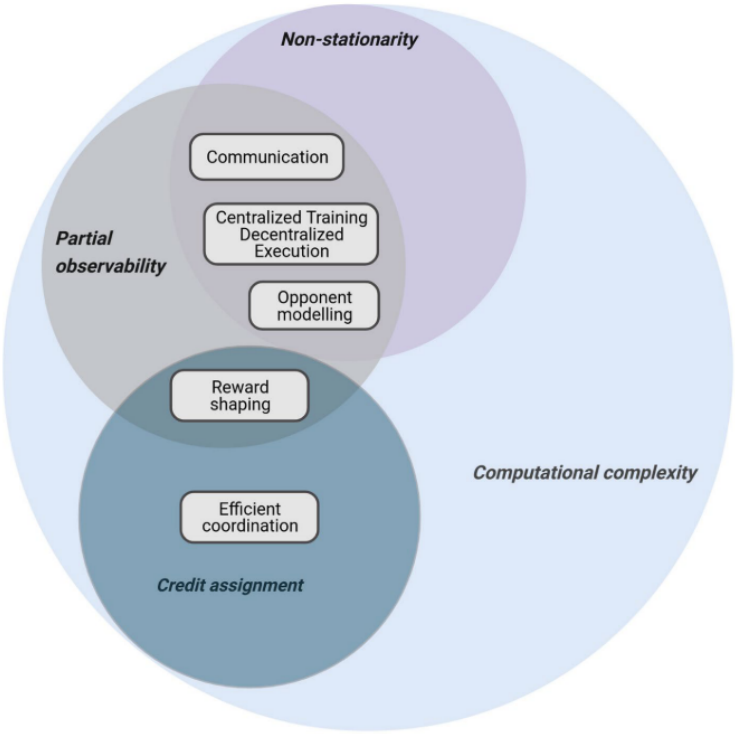

Figure 2.6: Venn diagram of challenges and solutions. The taxonomy of DMARL algorithms comprises five groups: centralised training and decentralised execution, opponent modelling, communication, efficient coordination and reward shaping. Approaches may tackle one or more challenges: nonstationarity, partial observability, credit assignment and computational complexity. Computational complexity is a universal challenge for all approaches. This Venn diagram shows the relations between the surveyed groups of studies and the addressed challenges.Wong et al., 2022

2.4.2 Training Paradigms in MARL

Fully Decentralized Learning

A fundamental approach in multi-agent reinforcement learning is fully decentralized learning, often referred to as independent learning. In this setting, each agent learns its own policy using only its local observations, actions, and rewards, without sharing information or parameters with other agents.

While this formulation is simple and does not require access to global state information, it introduces significant challenges in cooperative environments. In particular, from the perspective of each agent, the environment becomes non-stationary, as the policies of other agents are continuously changing during training. This violates the

standard reinforcement learning assumption of a stationary environment and makes convergence more difficult Papoudakis et al., 2019.

Moreover, independent learning is typically less sample-efficient, since experience is not shared across agents, and coordination between agents must emerge implicitly. As a result, fully decentralized approaches are often outperformed by methods that incorporate some form of information sharing or coordination Wong et al., 2022.

Fully Centralized Training

In centralized training, all agents share a joint model over their actions and observations. A centralized policy maps the joint observation of all agents to a joint action and is equivalent to an MPOMDP (Multiagent Partially Observable Markov Decision Process) policy Gupta, Egorov and Kochenderfer, 2017.

Although this approach addresses non-stationarity by transforming the multi-agent problem into a single-agent one with a joint observation space, it introduces a new challenge: computational cost Wong et al., 2022. In fully centralized approaches, both training and execution rely on a central controller. Due to the presence of this central controller, to which all agents send their observations and which determines the actions for each agent, the size of the joint observation and action spaces grows exponentially as the number of agents increases.

Some approaches attempt to mitigate this issue by factorizing the action space; however, the problem fundamentally remains. Additionally, when each agent treats the others as part of the environment, there is limited capacity for learning coordinated behaviour.

Centralized Training with Decentralized Execution (CTDE)

A more intermediate approach separates the components of the learning architecture into training and execution. It is still considered a form of centralized learning, as the critic component is centralized during training and has access to the observations and actions of all agents, while the actor component remains decentralized. Each agent follows its own policy for action selection based on local observations, rather than

the joint view of the "know-it-all" critic, while the critic leverages joint information Papoudakis et al., 2019. Thus, each agent acts locally, but it has been trained by a critic that had access to the full picture.

As noted by Papoudakis et al., once the critic has fulfilled its role during training, it can be removed, resulting in fully decentralized execution. This ensures consistency between training and execution and helps mitigate the non-stationarity problem, without incurring the high computational cost of fully centralized approaches.

This framework integrates naturally with policy gradient methods, where each agent's policy is conditioned during training on joint information. In this context, the counterfactual advantage can be interpreted as the value of a selected action compared to alternative actions Papoudakis et al., 2019, which also helps address the credit assignment problem.

Parameter Sharing

Another intermediate architecture is parameter sharing. It works better with homogeneous agents. In this approach, we allow agents to share the parameters of a single policy Gupta, Egorov and Kochenderfer, 2017. Thus, the policy is trained using the experience of every agent participating in the learning process. As Gupta et al. point out, this does not mean that each agent selects the same action, since each one receives a local observation and selects an action based on it. This makes the architecture highly scalable.

Both CTDE and parameter sharing employ decentralized execution; however, their training differs. CTDE relies on a centralized critic that utilizes joint observations and actions during training Papoudakis et al., 2019, whereas parameter sharing uses a shared policy trained on the collective experience of all agents Gupta, Egorov and Kochenderfer, 2017. This shared learning process helps mitigate non-stationarity, as all agents update a common policy, reducing the instability caused by independently changing behaviours. However, since no joint information is explicitly modeled, coordination must emerge implicitly through shared experience rather than being directly enforced during training.

The above training paradigms highlight different ways of addressing the challenges of multi-agent reinforcement learning, particularly non-stationarity and partial observability. Fully centralized approaches offer strong coordination but suffer from scalability limitations, while CTDE introduces a balance between centralized learning and decentralized execution. Parameter sharing further improves scalability by leveraging shared experience among agents, while maintaining decentralized control.

Based on these considerations, the choice of learning paradigm depends on the trade-off between coordination, scalability, and practical applicability in complex environments.

2.5 Environments and Benchmark Games

Overview of PettingZoo Environments

PettingZoo is a Python library created to standardize environments for multi-agent reinforcement learning (MARL). It was first released in 2020 by Terry et al. and later adopted under the Farama Foundation, which also maintains related projects such as Gymnasium and SuperSuit. PettingZoo was designed to play the same role for MARL that OpenAI Gym played for single-agent reinforcement learning: it provides a common, unified interface for environments, making it easier to compare algorithms across a wide range of tasks Terry et al., 2020.

A key feature of PettingZoo is that it supports two main interaction styles. The first is the *Agent-Environment Cycle (AEC)* API, where agents take turns one after the other. The second is the *Parallel API*, where all agents can act simultaneously. This dual API design gives researchers flexibility, allowing them to test algorithms that are either strictly turn-based or that assume synchronous parallel actions.

The library is organized into several families of environments, each designed to capture different types of multi-agent interactions. Examples included in the table above:

Table 2.2: Main environment families in PettingZoo Terry et al., 2020,Farama Foundation, 2023.

Family	**Description and Examples**
Butterfly	Cooperative graphical games that require teamwork. Examples include *Pistonball* (agents coordinate pistons to push a ball) and *Cooperative Pong*.
MPE (Multi-Agent Particle Environments)	Simple continuous environments with agents represented as particles. Used to study cooperation, communication, and adversarial dynamics.
Classic	Multi-agent board and card games such as Tic-Tac-Toe, Chess, or Texas Hold'em. Useful for testing algorithms on structured, strategic decision-making.
Atari	Multi-agent versions of Atari games (e.g., Pong, Space Invaders), extending the ALE framework to multi-agent settings.
SISL (Sequential Social Dilemmas)	Environments designed to study cooperation vs. defection, inspired by classical game theory. Examples include *Prisoner's Dilemma*-like setups with sequential interactions.

Since its release, PettingZoo has a wide coverage of cooperative, competitive, and mixed-sum games that allows algorithms to be evaluated under diverse conditions. Combined with SuperSuit for environment preprocessing, PettingZoo has helped establish reproducible standards for multi-agent reinforcement learning.

2.5.1 Detailed Description of Pistonball

Pistonball is one of the cooperative environments in the Butterfly family of PettingZoo. It is a two-dimensional physics-based game where up to twenty piston agents must work together to push a large ball toward the left side of the screen. Each piston can only move up or down, and success depends on many pistons coordinating their movements to apply force on the ball at the right time. The physics of

the game are simulated using the Chipmunk2D engine, which ensures that the ball's movement follows realistic rules. An episode ends either when the ball reaches the left wall or when a maximum of 125 time steps (cycles) has been played Farama Foundation, 2023.

State space. Each piston receives a local observation in the form of an RGB image. This observation shows the piston itself, its immediate neighbors (or the wall if it is on the edge), and the vertical space above, where the ball may appear. The local image has dimensions $(457, 120, 3)$ with pixel values in the range $[0, 255]$. In addition, the environment provides a global state consisting of the entire arena as an RGB image of dimensions $(560, 880, 3)$. The global state can be useful for centralized training methods, while the local observations reflect what each agent actually perceives.

Action space. Two variants are available. In the discrete version, each piston has three possible actions: move up, move down, or stay still, where each movement shifts the piston by four pixels. In the continuous version, the action space is a scalar in $[-1, 1]$: negative values move the piston down, positive values move it up, and the magnitude of the value determines the displacement (up to the same maximum as in the discrete case). In both cases, the action space is one-dimensional because pistons can only move vertically.

Reward structure. Rewards are fully cooperative, meaning all pistons receive the same reward signal. At each timestep, the reward consists of two parts: (1) a displacement-based reward proportional to the horizontal progress of the ball toward the left wall (positive if the ball moves left, negative if it moves right), and (2) a constant penalty of -0.1 that encourages agents to complete the task as quickly as possible. The shared reward structure makes coordination essential and introduces the credit assignment problem, since individual contributions cannot be easily distinguished.

Motivation and challenges. Pistonball was selected as the benchmark environment for this thesis because it provides a clear yet challenging setting for studying cooperative multi-agent reinforcement learning. Unlike single-agent tasks, success cannot be achieved by any one agent acting alone; instead, performance depends on the simultaneous and well-timed coordination of many simple agents. This makes Pistonball an ideal testbed for evaluating algorithms that combine decentralized observations with centralized training. At the same time, the environment is computationally efficient, easy to visualize, and configurable with respect to the number of pistons, the type of action space, and ball physics parameters.

—

2.6 Summary of Theoretical Background

This chapter introduced the fundamental concepts of reinforcement learning, focusing on the agent–environment interaction and the core elements that define the learning process. The problem was formalized through the Markov Decision Process (MDP), establishing the basis for sequential decision-making under uncertainty.

Key reinforcement learning algorithms were presented, highlighting the differences between value-based methods (e.g., Q-learning and SARSA), policy-based approaches (e.g., REINFORCE), and actor–critic architectures. Particular emphasis was given to Proximal Policy Optimization (PPO), as a stable and efficient method for policy optimization, combining practical performance with theoretical grounding.

In the multi-agent setting, the introduction of multiple interacting agents leads to additional challenges, with non-stationarity identified as a central issue, alongside credit assignment and partial observability. Different training paradigms were examined, illustrating key trade-offs: fully centralized approaches enable strong coordination but suffer from scalability issues, CTDE balances centralized learning with decentralized execution, while parameter sharing improves scalability by leveraging shared experience, at the cost of implicitly learned coordination.

Finally, the experimental environment was introduced, providing the practical setting in which these concepts and methods are applied.

Chapter 3

Methodology

3.1 Problem Formulation & Methodological Scope

The objective of this thesis is to examine whether a single shared neural policy can coordinate multiple piston agents in a cooperative physics-based environment without relying on explicit communication or role differentiation. Instead of assigning an independent policy to each piston, all agents are controlled by a single PPO policy that receives local observations and produces individual actions. This formulation directly raises the question of whether cooperative behavior can emerge purely through shared gradients and a common reward signal under decentralized execution.

Based on this premise, the central research question is formulated as follows: *How does collapsing multiple agents into a single shared PPO policy affect coordination and training stability in a stochastic multi-agent environment?* Two auxiliary questions naturally follow, namely whether coordinated behavior can emerge under partial observability and how stable such coordination remains across training and evaluation episodes.

In order to examine this hypothesis, the methodological scope of the study, a single shared neural policy controls all agents in the environment, rather than assigning independent policies to each agent. This design follows a parameter-sharing paradigm, where learning is shared across agents while execution remains decentralized, as each agent acts based solely on its local observation.

This choice is suitable for the environment examined, where agents are homogeneous, share the same action space, and operate under a common objective. Under these conditions, parameter sharing allows the learning process to generalize between agents, reducing redundancy, and enabling scalable training.

While this design has its advantages, it also introduces known limitations, such as gradient interference across agents and a reduced capacity for agent-specific specialization. These limitations are explicitly accepted as part of the experimental design.

3.2 Simulation Environment

To ground the research question in a concrete setting, the following section refreshes some key points of the simulation environment used for evaluation. All experiments are conducted in the `Pistonball-v6` environment, a 2D physics-based simulation in which several piston-like entities jointly act on a ball. Each piston receives a local pixel observation that captures only a small spatial window around its position, while the environment provides a shared global reward that encourages sustained ball motion. This setup creates a cooperative multi-agent control problem characterized by decentralized execution, partial observability and shared reward signals.

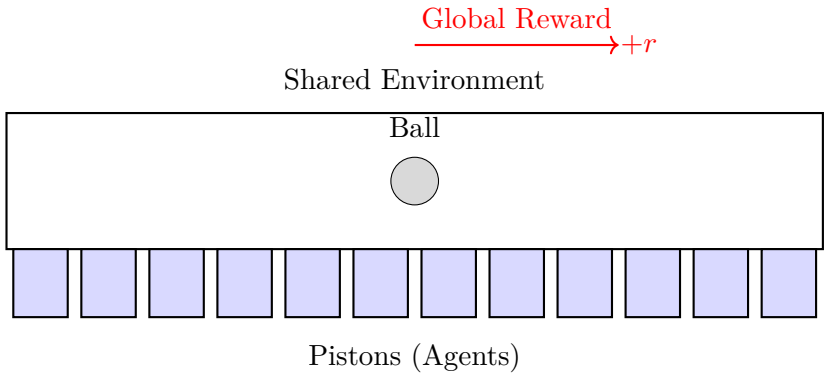

Figure 3.1: Schematic representation of the Pistonball environment. Multiple piston agents act within a shared environment, while a global reward encourages forward ball motion.

Even though Pistonball is commonly employed in multi-agent reinforcement learning settings, where multiple learning agents interact simultaneously, the present thesis adopts a different perspective. In this formulation, learning is treated as a shared-policy setup, where a single policy is applied independently to a local observation of each piston.

In other words, a single policy controls all pistons, while the remaining entities in the environment serve as physical context. This formulation isolates the effect of policy sharing and avoids additional mechanisms such as communication protocols, role assignment, or heterogeneous agent modeling.

Episodes end either when the ball exits the arena or when a fixed time horizon is reached. Performance in this environment is evaluated in terms of cumulative reward and persistence of ball motion over time. This allows the multi-agent system to be studied under a unified learning process.

3.3 PPO Training Framework and Shared Policy Architecture

In this work, learning is performed using the Proximal Policy Optimization (PPO) algorithm, as implemented in the Stable-Baselines3 framework. PPO optimizes a

stochastic policy through a clipped surrogate objective, allowing stable on-policy learning from rollout data.

At each timestep t, every piston i receives its own local observation o_t^i. The shared policy π_θ maps each observation to an individual action $a_t^i = \pi_\theta(o_t^i; \theta)$ using a common set of parameters θ. The experience generated by all pistons is collected in a shared training buffer and used jointly to update the policy. As a result, learning dynamics across agents is coupled through both shared rewards and shared gradients.

In our experiment, the environment can be interpreted as a Dec-POMDP, since multiple agents operate under partial observability and select actions based solely on local observations. During training, experience from all agents is aggregated and processed centrally, while at execution time, each agent acts independently based only on its own local observation.

Figure 3.2 illustrates how local observations from multiple agents are processed by a single shared policy to produce individual actions at each timestep.

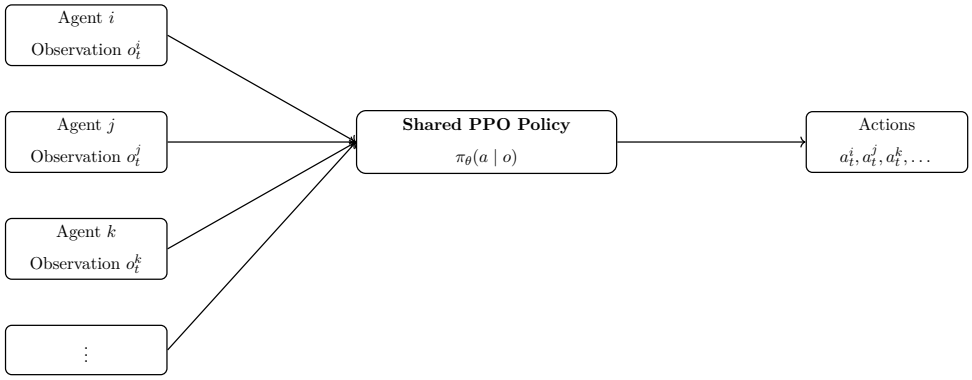

Figure 3.2: Multiple agents provide their local observations to a single shared PPO policy. The same policy parameters are used to compute individual actions for each agent.

The shared PPO policy follows a convolutional actor–critic architecture. Raw pixel observations are processed by a convolutional encoder, which extracts spatial features and compresses the high-dimensional input into a compact latent representation z_t^i. The actor network maps the latent representation z_t^i to a set of action scores (logits),

which are then transformed into a probability distribution over actions using the softmax function:

$$\pi_\theta(a_t^i \mid o_t^i) = \text{softmax}\Big(f_{\text{actor}}(z_t^i)\Big). \tag{3.1}$$

Both heads, actor and critic, share the same convolutional encoder but maintain separate output parameters. No decoder is used, as the encoder is optimized for control performance rather than reconstruction. Training is performed on-policy for a fixed number of timesteps, with periodic evaluation used to assess the stability and effectiveness of the learned cooperative behavior.

3.4 Environment Preparation Observation Processing

The experiments are conducted in the `pistonball-v6` environment, which emits pixel-based RGB observations and a single shared reward, while each piston receives only a localized visual slice of the arena. This introduces partial observability and enforces decentralized action selection, as no piston has direct access to the full environment state.

To make the environment compatible with Stable-Baselines3 and suitable for training a single shared PPO policy, a structured preprocessing pipeline was applied. Raw RGB frames are converted to grayscale to reduce dimensionality, resized to 64×64 to limit convolutional cost, and stacked over two frames to expose short-term temporal context. The resulting observations provide sufficient motion cues for control without introducing recurrent networks or explicit memory mechanisms.

The preprocessed observations from all pistons are then collected into a batch and forwarded through the shared convolutional encoder in a single vectorized pass. This design allows the shared policy to produce a separate action for each piston based solely on its own observation slice, while still benefiting from centralized gradient updates during training. The handling of multiple observations in this manner preserves decentralized execution—each piston acts independently according to its own

observation—without requiring explicit communication or agent-specific specialization.

Because PPO assumes a single-agent Gymnasium-style interface, the native PettingZoo parallel API is adapted through a conversion layer that standardizes outputs into the (obs, reward, done, info) format expected by Stable-Baselines3.Vectorization treats agent observations as parallel inputs, enabling batch processing rather than representing truly independent environments. This corresponds to decentralized execution, where learning is coupled across agents through shared gradients and aggregated experience, rather than through an explicit centralized controller.

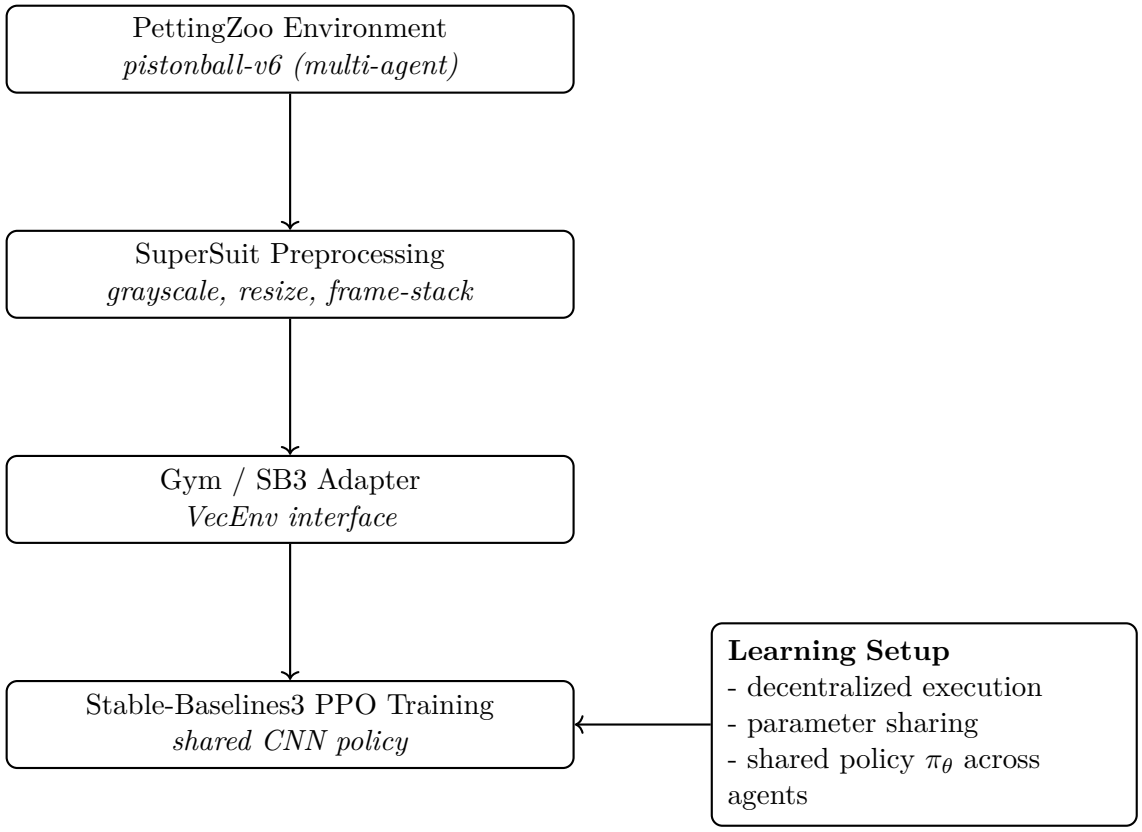

Figure 3.3: Processing and training pipeline from PettingZoo to Stable-Baselines3 PPO with a shared multi-agent policy.

Reward Structure and RL Interpretation

Progress toward the task objective is encoded through a shared global reward derived from the horizontal displacement of the ball within the environment. At each timestep t, all agents receive the same scalar reward defined as

$$r_i(t) = k \cdot \Big(x_{\text{ball}}(t) - x_{\text{ball}}(t-1)\Big), \qquad i = 1, \ldots, 12,$$

where $x_{\text{ball}}(t)$ denotes the horizontal position of the ball and k is a scaling constant. Positive rewards are therefore obtained when the collective action of the pistons results in forward ball motion, while backward movement yields negative feedback.

From a reinforcement learning perspective, this reward signal directly specifies the optimization target that the PPO algorithm seeks to maximize. By design, the reward is shared identically across all agents, eliminating the need for per-agent reward shaping and ensuring that learning is aligned with the true cooperative objective of the task. Since no agent receives individualized feedback, performance improvements can arise only from action patterns that benefit the system as a whole rather than isolated pistons. As a result, policy updates reinforce coordinated behavior via a single gradient signal aggregated across agents and time, instead of requiring explicit attribution of rewards to specific agents.

Under this shared-reward formulation, the PPO optimization objective can be written as

$$\max_{\theta} \; \mathbb{E}\left[\frac{1}{12}\sum_{i=1}^{12}\sum_{t=0}^{T} r_i(t)\right],$$

which corresponds to maximizing the shared return, since all agents receive identical rewards at each timestep.

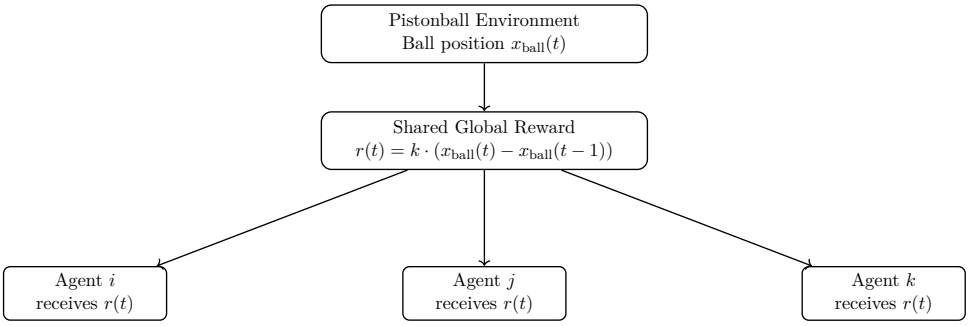

Figure 3.4: The reward signal is computed from the global ball displacement in the Pistonball environment and broadcast identically to all agents. This shared reward encourages coordinated behavior by aligning all agents with the same optimization signal.

3.5 Training Methodology

This section describes the procedure used to train the shared PPO policy, focusing on configuration choices, training execution, experiment design, and the iterative refinement process. The aim is to document how stable and reproducible models were obtained. Before large-scale training, the entire training pipeline was validated through short pre-runs to ensure correct interaction between the environment, preprocessing, and optimization components. This step ensured that subsequent performance differences reflected learning behavior rather than implementation-related issues.

3.5.1 PPO Configuration and Hyperparameters

The PPO algorithm is configured using a small set of hyperparameters that govern optimization dynamics, exploration behavior, and rollout structure. Rather than conducting an exhaustive hyperparameter search, the focus of this work is placed on identifying a stable and reproducible training regime suitable for cooperative multi-agent control under partial observability.

Hyperparameters are therefore treated as design variables that influence learning stability rather than quantities to be optimized in isolation. In particular, parameters related to the rollout length, exploration pressure, and update frequency are expected to affect convergence speed, variance, and robustness. A reference configuration is

defined to serve as a baseline for all experiments, while controlled variations are introduced incrementally during training. The impact of these variations is analyzed in the evaluation chapter, where their effect on performance and stability is examined empirically.

3.5.2 Training Protocol, Model Selection, and Iterative Refinement

Each training run is executed with a fixed computational budget and updated on-policy using PPO. Actions are sampled stochastically during training to support exploration and to avoid early convergence to unstable strategies. Training progress is monitored at regular intervals, meaning that performance is evaluated periodically during training rather than only after completion. This allows learning behavior to be tracked throughout the training process.

At the end of each run, two policy instances are retained: the final model corresponding to the last update step, and the best-performing model observed during training. This distinction is necessary because peak performance does not always coincide with the final iteration. In practice, some policies reached their highest performance early and later degraded due to instability. Retaining both models allows learning dynamics to be analyzed more faithfully in the evaluation chapter.

Beyond individual runs, a central question guides the training process: *how can improvements be distinguished from random fluctuations caused by stochastic training and initialization?* To address this question, training is not treated as a sequence of isolated experiments but is organized as an iterative refinement process. An initial baseline configuration is first validated under stochastic execution and across multiple random seeds to establish a stable reference behavior.

Subsequent iterations introduce controlled and targeted modifications, which are evaluated against the current baseline before any further changes are applied. This staged progression ensures that observed performance differences can be attributed to specific interventions rather than to randomness arising from action sampling or seed-dependent effects.

Two complementary refinement modes are employed within this framework. In surgical refinement, small and localized adjustments are applied to selected hyperparameters while preserving the overall training structure, allowing issues such as insufficient exploration or unstable updates to be addressed without disrupting previously learned behavior. In rebuild refinement, the PPO setup is reconstructed with structural changes, such as modified rollout horizons or batch configurations, while transferring learned weights when appropriate. This mode is used when the existing baseline cannot accommodate the desired improvement within its current configuration.

A candidate configuration is promoted to the baseline status only if it consistently shows improved performance under stochastic evaluation, reduced variability across episodes, and robustness across multiple random seeds. Configurations that rely on favorable initialization, exhibit high variance, or show signs of performance collapse are explicitly rejected. This disciplined training protocol ensures that the improvements reported reflect robust behavioral gains rather than chance fluctuations or transient effects.

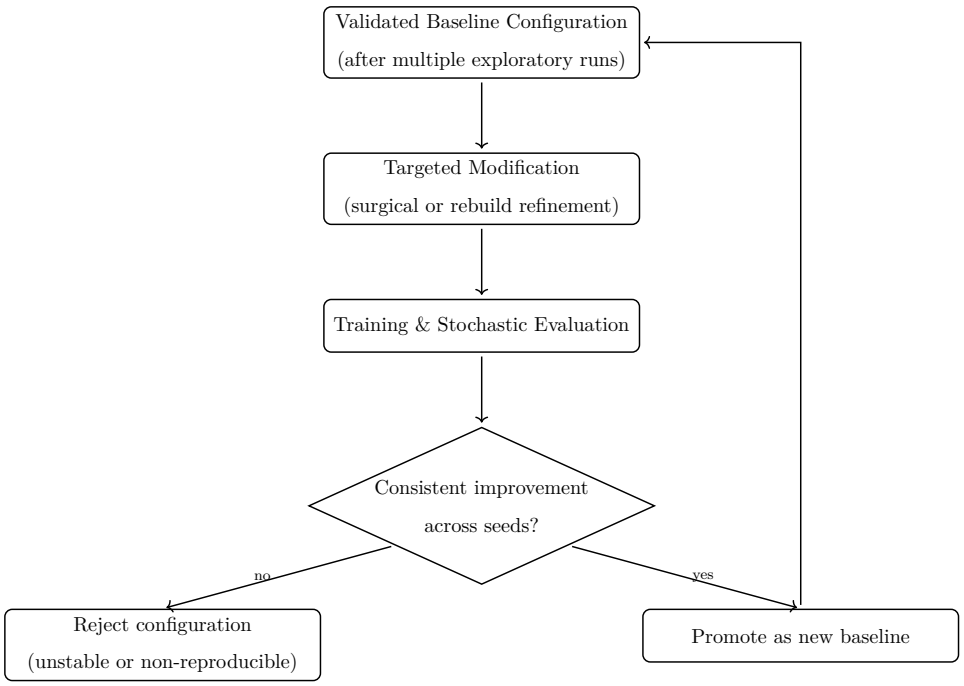

Figure 3.5: Iterative training protocol with baseline validation, controlled refinement, and promotion criteria.

3.5.3 Experiment Design and Reproducibility

The experimental process begins with a baseline configuration trained under fixed initialization conditions. Subsequent experiments introduce controlled modifications to a limited set of factors, including random seed, selected PPO hyperparameters, and, in specific cases, architectural changes. This design ensures that observed performance differences can be attributed to deliberate interventions rather than uncontrolled stochastic effects.

Reproducibility-oriented practices were incorporated through consistent configuration snapshots and structured run directories that encode the key experimental parameters. This makes each training run fully traceable and facilitates more reliable comparison between different configurations and training stages.

3.6 Evaluation Phase

The evaluation phase assesses the behavior of the learned policy under controlled conditions and determines whether the PPO agent has developed stable cooperative strategies. Since PPO is trained as a stochastic policy, the evaluation process considers two execution modes that reflect different aspects of performance: deterministic and stochastic action selection.

Deterministic vs Stochastic Execution

Deterministic evaluation uses $\arg\max_a \pi_\theta(a|o)$, selecting the most probable action in each timestep. This mode isolates the policy's "intended" behavior by restricting randomness and reveals how well the agent performs when following its most confident decisions. However, in a multi-agent environment such as *Pistonball*, this mode has limitations: the dynamics are non-stationary, other agents behave independently, and small divergences cannot be corrected. As a result, deterministic evaluation often underestimates real performance.

Stochastic evaluation samples from the full action distribution $a \sim \pi_\theta(\cdot|o)$. This reflects the actual behavior of the policy during training and provides a more realistic estimate of performance under uncertainty. In this mode, the agent can choose slightly different actions between episodes, revealing the degree to which cooperative behavior is stable rather than accidental. For this reason, stochastic evaluation is treated as the primary performance metric throughout this work.

Evaluation Procedure and Metrics

After training, both the final model (corresponding to the last update step) and the best-performing model (achieving the highest evaluation score during training) are evaluated. Evaluation runs are initially conducted over a moderate number of episodes per execution mode to obtain a first estimate of performance.

As training stabilizes and candidate baseline models are narrowed down, the number of evaluation episodes is increased to obtain more reliable performance estimates

and to reduce the effect of stochastic variability. This progressive evaluation strategy allows for both early-stage inspection and robust final assessment of learned behavior.

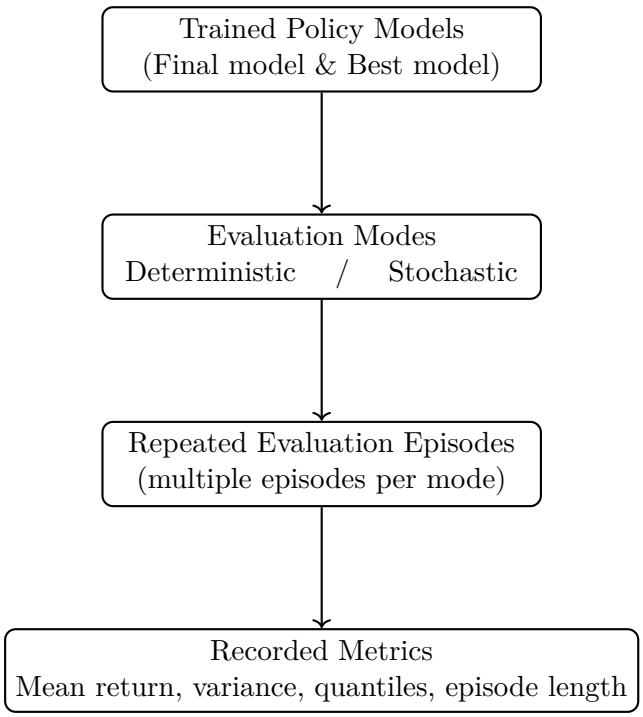

Figure 3.6: Evaluation pipeline and performance metrics used to assess policy stability and effectiveness.

All evaluation results are recorded in a structured manner to enable systematic comparison across models and training stages. Performance statistics and return distributions are aggregated across episodes and seeds, forming the basis for the analysis presented in the following chapter.

3.7 Summary

The experimental workflow was incremental and evidence-driven. Rather than searching for a single optimal configuration, the process focused on identifying a stable and reproducible-oriented setup, assessing robustness across seeds, and refining behavior through controlled fine-tuning. This approach aligns with the complexities of multi-agent reinforcement learning, where environment dynamics are non-stationary and policy stability is a prerequisite for meaningful evaluation.

Chapter 4

Results Analysis

4.1 Experimental Overview

We present the experimental results following the staged training methodology described in the previous chapter. The results are organized according to the main phases of the training process rather than as isolated runs.

First, results from the exploratory hyperparameter search are reported to show how unstable or poorly converging configurations are filtered out and how the baseline configuration is selected. The focus at this stage is on training stability rather than absolute performance.

The results of the baseline configuration evaluated on multiple random seeds are presented. This stage establishes a reference level of performance and variance under stochastic execution and serves as the comparison point for all subsequent results.

Finally, fine-tuning results are reported, illustrating how targeted modifications affect performance and stability relative to the baseline.

Across all stages, policies are primarily evaluated using the average episodic team reward, with variability across episodes and seeds used as an indicator of robustness. Unless stated otherwise, comparisons are based on stochastic evaluation.

4.2 Hyperparameter Exploration Strategy

Before fixing the final configuration, an extensive exploration phase was conducted. The purpose of this phase was not to exhaustively optimize the PPO algorithm, but to identify a region of the hyperparameter space where learning is stable and cooperative behavior emerges consistently.

The exploration focused primarily on the rollout length used for PPO updates (`n_steps`), the entropy coefficient controlling exploration pressure (`ent_coef`), and the overall training budget. Rollout lengths ranging from 1024 to 12288 steps were evaluated, entropy coefficients were varied between 0.0001 and 0.01, and training budgets typically ranged from approximately one million timesteps for baseline runs to shorter budgets during fine-tuning stages.

Early experiments with relatively short rollouts, such as `n_steps = 1024` combined with high entropy (`ent_coef = 0.01`), demonstrated that PPO was capable of discovering non-trivial behavior. In these cases, stochastic evaluation occasionally yielded moderate positive rewards, and some checkpoints achieved substantially higher performance. However, learning dynamics were highly variable, and results were sensitive to the specific checkpoint selected for evaluation. Similar instability was observed for intermediate rollout lengths such as `n_steps = 2048` with moderate entropy, where performance varied significantly across runs and seeds, particularly under deterministic evaluation.

At the other extreme, very long rollouts (`n_steps = 12288`) combined with very low entropy coefficients produced numerically stable policies but did not lead to consistent improvements in episodic reward. In several such configurations, deterministic evaluations clustered around negative mean values with relatively low variance, indicating convergence to conservative but suboptimal strategies.

Taken together, these observations suggest a trade-off between exploration and stability. Short rollouts with high entropy promoted exploration but resulted in high variance and unstable learning, while extremely long rollouts with minimal entropy led to stable yet under-performing policies.

Based on this analysis, subsequent experiments focused on an intermediate configuration that balances these effects, namely `n_steps = 4096` with a moderate entropy coefficient `ent_coef = 0.006`. This configuration was selected as the baseline for further evaluation across random seeds and for controlled fine-tuning.

Configuration	Observed Behavior	Main Limitation
1024 / 0.01	Strong exploration	High variance
2048 / moderate	Occasional improvements	Unstable evaluations
12288 / low	Stable behavior	Conservative policies
4096 / 0.006	Balanced performance	Selected baseline

Table 4.1: Qualitative observations from the hyperparameter exploration phase.

4.3 Baseline Configuration: `n_steps = 4096`, `ent_coef = 0.006`

Following the exploratory phase, the configuration with `n_steps = 4096` and `ent_coef = 0.006` was selected as a baseline for further analysis. This choice represents a compromise between sample efficiency and training stability: rollout lengths are sufficiently long to capture meaningful temporal patterns in the cooperative dynamics, while the entropy coefficient remains large enough to prevent premature convergence to overly deterministic behavior.

For seed 0, this baseline configuration exhibits a clear divergence between stochastic and deterministic evaluation. In stochastic execution, the policy achieves a mean episodic reward of approximately 18.55, with a median of around 13.60 and an average episode length of roughly 122.5 steps. In contrast, deterministic evaluation yields substantially lower performance, with a mean reward of approximately -1.86, high variability across episodes, and a median close to -9.38.

This discrepancy already illustrates a central observation of this work: under the same learned policy, stochastic action selection enables sustained cooperative beha-

vior, whereas deterministic execution tends to suppress it. Importantly, this effect is not specific to a single initialization; as shown in the following sections, it persists when results are aggregated across multiple random seeds and remains evident throughout subsequent fine-tuning stages.

4.4 Fine-Tuning Procedure on the Baseline Configuration

Once the baseline configuration was established, we proceeded with an iterative fine-tuning process aimed at improving performance while preserving training stability.The fine-tuning stages were initialized from previously trained weights and followed two complementary strategies. In the first, only selected hyperparameters such as the learning rate or entropy coefficient were adjusted, resulting in small, localized changes in the training dynamics. In the second, the PPO rollout structure was modified by changing the rollout length or batch configuration, while retaining the learned network weights.

For seed 0, three successive fine-tuning stages were applied on top of the baseline configuration with `n_steps = 4096` and `ent_coef = 0.006`. The final fine-tuned policy (run directory ending in `__20251116-185344`) exhibits a substantial improvement under stochastic evaluation. Specifically, the policy achieves a mean episodic reward of approximately 43.61, with a standard deviation of 41.59 and a median of 37.38. The average episode length is reduced to around 108.3 steps, compared to the maximum horizon of 125.

Relative to initial baseline performance under stochastic execution (mean reward ≈ 18.55), this corresponds to an improvement factor of approximately $2.35\times$. The reduction in episode length suggests that successful cooperative behavior is achieved earlier within each episode, indicating more decisive and coordinated action patterns.

An important observation is that the deterministic evaluation remains near zero or slightly negative, with a mean reward of approximately -0.98 and a median close to -13.03. This mirrors the behavior observed at the baseline stage and reinforces the

conclusion that effective cooperation in this environment relies on stochastic action selection.

The same fine-tuning procedure was applied independently for seeds 1, 2 and 3. In each case, the final fine-tuned run is treated as the representative policy for that seed and is used in subsequent cross-seed analysis.

4.5 Final Policies Across Seeds

Table 4.2 summarizes the performance of the final fine-tuned policies for seeds 0–3 under both stochastic and deterministic evaluation. All numbers are aggregated over 200 evaluation episodes.

Table 4.2: Performance of final fine-tuned policies across seeds (`n_steps = 4096`, `ent_coef = 0.006`).

Seed	Eval	Mean	Std	Median	Mean length
0	Stoch.	43.61	41.59	37.38	108.27
	Det.	−0.98	29.27	−13.03	120.28
1	Stoch.	9.14	23.96	6.44	124.15
	Det.	−3.89	18.76	−10.48	125.00
2	Stoch.	4.01	23.99	−0.48	123.96
	Det.	−6.42	18.44	−11.97	125.00
3	Stoch.	45.66	42.73	44.84	106.73
	Det.	0.14	29.72	−11.93	121.18

Two key patterns appear clearly:

1. **Stochastic policies achieve consistently positive mean rewards.** Across the four seeds, the average stochastic mean is approximately 25.6 points, with the strongest seeds (0 and 3) achieving mean rewards above 40 and medians around 40–45. For these seeds, even the 25th percentile of the stochastic reward distribution is positive, indicating that most episodes are successful.

2. **Deterministic policies clustered around zero or slightly negative mean rewards**. When the same policies are evaluated deterministically, the mean

rewards move close to 0 and often become negative. The average deterministic mean across seeds is around -2.8, and the medians are consistently negative.

In addition, for high-performing seeds 0 and 3 the mean episode length under stochastic evaluation is around 106–108 steps, clearly below the 125-step horizon. This suggests that in many runs the ball is pushed to the right side well before the time limit, which is consistent with successful cooperative behavior. In contrast, deterministic evaluations tend to remain close to the full horizon, especially for seeds 1 and 2, where the mean episode length is exactly or almost 125 steps.

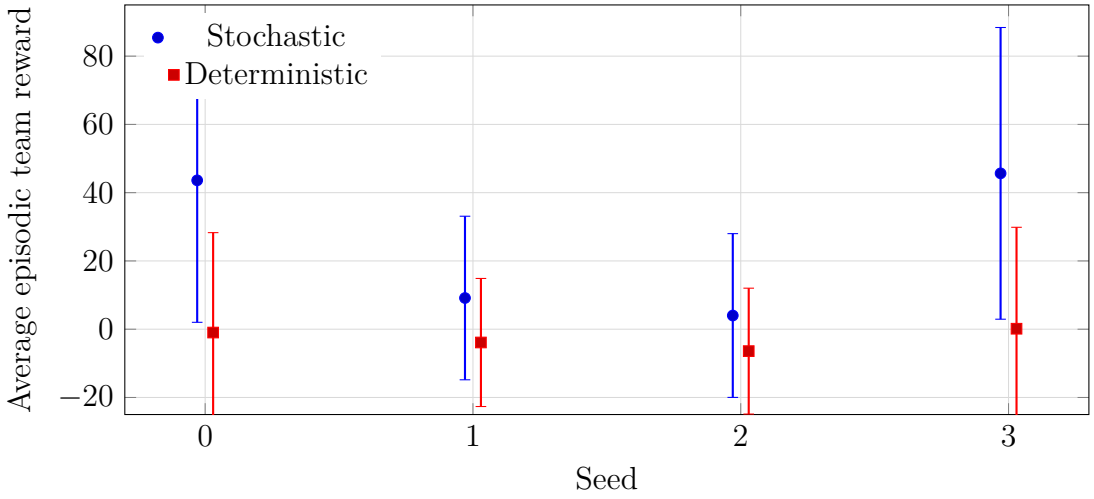

Figure 4.1: Mean episodic team reward (± standard deviation) of final fine-tuned policies across random seeds under stochastic and deterministic evaluation.

4.6 Improvements from Baseline to Final Policies

We now quantify how much performance improves when moving from the selected baseline configuration to the final fine-tuned policies. For each seed, we compare the baseline model trained with `n_steps = 4096` and `ent_coef = 0.006` against the corresponding final fine-tuned model, using the same evaluation protocol (200 episodes per mode).

Under stochastic evaluation, fine-tuning yields consistent gains for the high-performing seeds and more modest or mixed gains for the weaker seeds. In particular, seeds 0 and 3 show large increases in mean episodic reward, which is also reflected in shorter average episode lengths, indicating earlier task completion. Seeds 1 and 2 improve

less substantially, suggesting that fine-tuning cannot fully overcome unfavorable initialization effects in all cases.

Deterministic evaluation remains close to zero or slightly negative both at the baseline and at the final stage. This indicates that the primary effect of fine-tuning is to strengthen cooperative behavior under stochastic execution, rather than to produce a robust deterministic controller in this environment.

Overall, the baseline-to-final transition confirms the main role of fine-tuning in our pipeline: it amplifies cooperative behavior once a stable training regime is identified, but it does not eliminate the stochasticity dependency observed throughout the experiments.

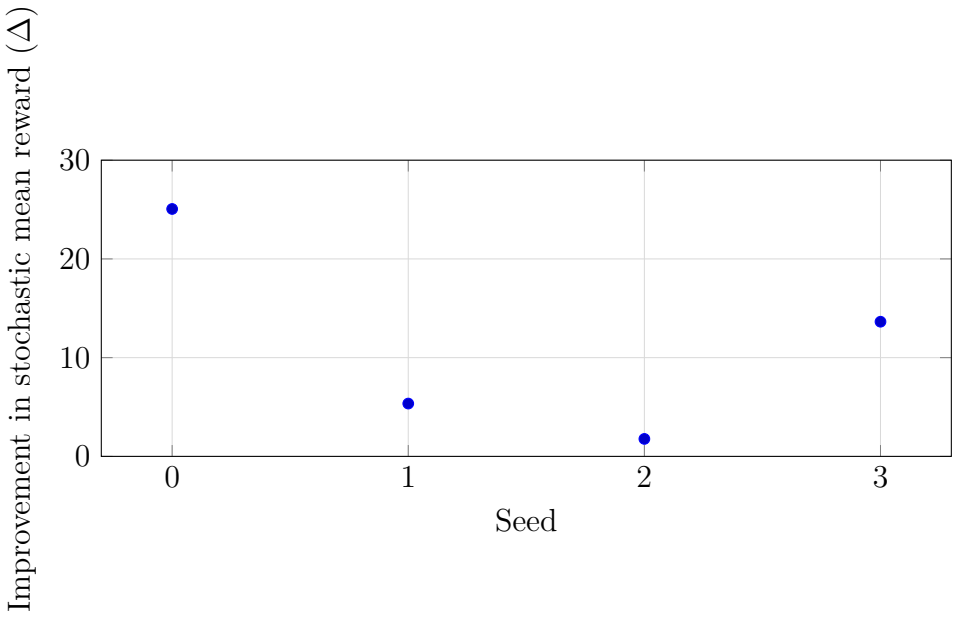

Figure 4.2: Improvement in stochastic mean episodic reward from the baseline policy (`n_steps = 4096`, `ent_coef = 0.006`) to the final fine-tuned policy across random seeds.

Taken together, the results confirm the role of stochastic execution and fine-tuning in enabling and stabilizing cooperative behavior under a shared policy.

4.7 Deterministic and Stochastic Evaluation in a Multi-Agent Context

Across all experiments, a consistent gap is observed between stochastic and deterministic evaluation. Policies that exhibit strong cooperative behavior under stochastic execution perform close to zero or negatively when evaluated deterministically. This pattern appears at the baseline stage, persists through fine-tuning, and remains stable across random seeds.

Rather than representing a training failure, this discrepancy reflects the cooperative and partially observable nature of the environment, in which agents share a single policy and must coordinate without access to global state information. Under these conditions, stochastic execution provides a more representative assessment of the learned policy's effectiveness. As such, all subsequent conclusions in this chapter are based primarily on stochastic evaluation results.

4.8 Summary of Findings

The experimental results presented in this chapter provide direct empirical answers to the research questions and the methodological assumptions introduced earlier. A systematic exploration of PPO hyperparameters identified an intermediate configuration (`n_steps = 4096`, `ent_coef = 0.006`) as a stable operating regime in which cooperative behavior can emerge reliably under a shared-policy formulation. This baseline offered a balance between sufficient temporal context for coordination and controlled exploration pressure.

Building on this configuration, iterative fine-tuning consistently improved stochastic performance across all random seeds. Although the magnitude of improvement varied, mean reward gains ranged from approximately 1.7× to 2.4×, indicating that once a stable coordination regime is reached, PPO can further reinforce cooperative dynamics through targeted refinement. In its strongest instances, the final fine-tuned policies achieved stochastic mean rewards above 40, accompanied by reduced episode

lengths, suggesting that the ball is often pushed to the goal side well before the time horizon is reached.

At the same time, the experiments highlight the sensitivity of training outcomes to random initialization and stochasticity. Performance varies across seeds, and successful cooperation does not emerge deterministically in all cases. Rather, effective coordination appears as an emergent outcome that depends on early training dynamics and is subsequently amplified through fine-tuning.

A consistent pattern across all experiments is the divergence between stochastic and deterministic evaluation. Policies that exhibit strong cooperative behavior under stochastic execution perform close to zero or slightly negative when evaluated deterministically. This observation confirms that deterministic performance alone is a misleading indicator of policy effectiveness in this cooperative multi-agent setting.

4.9 Reflection and Interpretation

The experimental results indicate that meaningful cooperative behavior can emerge even when all pistons are controlled by a single shared PPO policy under partial observability and decentralized execution. Despite the absence of explicit communication, role assignment, or agent-specific policies, several configurations were able to produce coordinated interactions that consistently pushed the ball forward and maintained collective motion over time.

At the same time, the experiments revealed that learning dynamics were highly sensitive to initialization and training conditions. Performance varied noticeably across random seeds, and policies that appeared promising during intermediate training stages did not always generalize consistently across evaluation settings. This variability suggests that convergence toward effective cooperation is not deterministic, but depends strongly on early policy updates and stochastic training dynamics.

The results also highlight the importance of fine-tuning after an initially cooperative regime has emerged. In several cases, continued training from well-performing checkpoints improved coordination stability and increased average episodic rewards.

However, fine-tuning alone was not sufficient to recover poor initial training trajectories, indicating that successful cooperation remains closely linked to the quality of the earlier learning phase.

An additional observation concerns the difference between stochastic and deterministic evaluation. Policies that performed relatively well under stochastic execution often degraded substantially when evaluated deterministically. This behavior suggests that stochasticity itself contributes to coordination by allowing the shared policy to adapt more flexibly to locally changing environmental states. Deterministic execution, by contrast, frequently produced rigid behavioral patterns that reduced adaptability and led to weaker collective performance.

Overall, the findings suggest that cooperative behavior in this setting emerges not through specialization or explicit coordination mechanisms, but through the interaction between shared policy optimization, common rewards, and stochastic action selection. The experiments therefore support the idea that parameter sharing can produce non-trivial coordination in cooperative multi-agent environments, while also revealing the sensitivity and instability that may accompany this formulation.

Chapter 5

Conclusions

This thesis investigated how a single PPO policy behaves in a cooperative multi-agent environment, using Pistonball-v6 as a case study. The objective was to examine how training dynamics, hyperparameter selection, fine-tuning, and evaluation methodology influence learning when a shared policy operates under partial observability and decentralized execution.

5.1 Discussion

The findings of this thesis suggest that cooperative behavior may emerge even in highly constrained shared-policy settings where agents operate using only local observations and decentralized execution. Although the achieved coordination was often unstable and sensitive to training conditions, the experiments nevertheless demonstrated that meaningful collective behavior can arise without explicit communication or predefined agent roles.

From a methodological perspective, the study highlights the importance of evaluation strategy in cooperative reinforcement learning. The observed differences between stochastic and deterministic execution indicate that policy behavior in partially observable environments may depend strongly on the manner in which evaluation is performed. This reinforces the idea that empirical results in reinforcement learning should be interpreted not only through final reward values, but also through stability, reproducibility, and behavioral consistency.

Overall, the experiments demonstrate that relatively simple shared-policy formula-

tions are capable of producing non-trivial cooperative behavior under strong constraints. Although the observed policies exhibited instability and sensitivity to training conditions, the results nevertheless support the use of parameter sharing as a meaningful framework for studying coordination in cooperative multi-agent reinforcement learning. More broadly, these findings are consistent with previous work on parameter sharing in cooperative MARL, where shared policies have previously been observed to produce coordinated behaviour under decentralized execution and limited local observations **gupta2017**.

5.2 Limitations and Future Work

This study is subject to several limitations that define the scope of its conclusions. First, the experimental analysis was restricted to a single environment, Pistonball-v6. Although this environment provides a challenging cooperative control problem, the observed behaviours cannot be assumed to generalize directly to tasks with different dynamics, reward structures, or agent interactions.

Second, the shared-policy formulation was not compared against alternative multi-agent approaches such as independent policies, centralized critics, or communication-based methods. This was a deliberate design decision, since the goal of the thesis was to investigate the internal dynamics of shared-policy PPO rather than to optimize performance relative to other MARL architectures.

Third, the evaluation focused primarily on episodic reward and episode duration. While these metrics align with the task objective, they do not fully capture finer-grained aspects of coordination such as synchronization patterns, behavioural diversity, or emergent agent roles.

Future work could extend this research by evaluating shared-policy learning in more heterogeneous environments where agents differ in observations or action capabilities. Another possible direction would be to investigate hybrid architectures that combine parameter sharing with limited forms of communication or memory. Finally, broader comparisons between stochastic and deterministic execution across multiple

cooperative environments could provide deeper insight into the role of stochasticity in multi-agent coordination.

Bibliography

Albrecht, Stefano V. and Subramanian Ramamoorthy (2012). 'A Survey of Approaches to Learning in Multiagent Environments'. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.3, pp. 292–314 (cit. on p. 1).

Bellman, Richard (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press (cit. on p. 15).

Bertsekas, Dimitri P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Englewood Cliffs, NJ: Prentice Hall (cit. on pp. 1, 15).

Farama Foundation (2023). *Farama Foundation Documentation*. URL: `https://farama.org` (cit. on pp. 37, 38).

Grondman, Ivo et al. (2012). 'A Survey of Actor–Critic Reinforcement Learning: Standard and Natural Policy Gradients'. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6, pp. 1291–1307 (cit. on p. 21).

Gupta, Jayesh K., Maxim Egorov and Mykel Kochenderfer (2017). 'Cooperative Multi-Agent Control Using Deep Reinforcement Learning'. In: *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* (cit. on pp. 34, 35).

Hernandez-Leal, Pablo et al. (2017). 'A Survey of Learning in Multiagent Environments: Dealing with Non-Stationarity'. In: *arXiv preprint arXiv:1707.09183* (cit. on p. 1).

Papoudakis, Georgios et al. (2019). 'Dealing with Non-Stationarity in Multi-Agent Deep Reinforcement Learning'. In: *arXiv preprint arXiv:1910.12010* (cit. on pp. 31, 34, 35).

Schulman, John, Philipp Moritz et al. (2016). 'High-Dimensional Continuous Control Using Generalized Advantage Estimation'. In: *International Conference on Learning Representations (ICLR)* (cit. on p. 26).

Schulman, John, Filip Wolski et al. (2017). 'Proximal Policy Optimization Algorithms'. In: *arXiv preprint arXiv:1707.06347* (cit. on pp. 1, 25, 27).

Sunehag, Peter et al. (2018). 'Value-Decomposition Networks For Cooperative Multi-Agent Learning'. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)* (cit. on p. 32).

Sutton, R. S. (1988). 'Learning to predict by the methods of temporal differences'. In: *Machine Learning* 3.1, pp. 9–44. DOI: 10.1007/BF00115009 (cit. on p. 16).

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. 2nd. Cambridge, MA: MIT Press (cit. on pp. 1, 5, 11, 12).

Tan, Ling et al. (2023). 'Research on weighted energy consumption and delay optimization algorithm based on dual-queue model'. In: *IET Communications* 18. DOI: 10.1049/cmu2.12710 (cit. on p. 28).

Terry, Justin K. et al. (2020). 'PettingZoo: Gym for Multi-Agent Reinforcement Learning'. In: *arXiv preprint arXiv:2009.14471* (cit. on pp. 36, 37).

Van Otterlo, Martijn and Marco Wiering (2012). *Reinforcement Learning and Markov Decision Processes*. Ed. by Marco Wiering and Martijn Van Otterlo. Springer, pp. 3–42 (cit. on p. 31).

Watkins, C. J. C. H. (1989). 'Learning from delayed rewards'. PhD thesis. King's College, Cambridge (cit. on p. 17).

Watkins, C. J. C. H. and P. Dayan (1992). 'Q-learning'. In: *Machine Learning* 8.3–4, pp. 279–292. DOI: 10.1007/BF00992698 (cit. on p. 17).

Williams, R. J. (1992). 'Simple statistical gradient-following algorithms for connectionist reinforcement learning'. In: *Machine Learning* 8.3 (cit. on p. 19).

Wong, Annie et al. (2022). 'Deep Multiagent Reinforcement Learning: Challenges and Directions'. In: *Artificial Intelligence Review* (cit. on pp. 30, 32–34).