Multi-Agent Deep Reinforcement Learning for Coverage Maximization

22125030 - Roopam Taneja
22114098 - Vraj Tamakuwala

Department of Computer Science and Engineering
Indian Institute of Technology Roorkee
Roorkee - 247667 (INDIA)

Submitted in partial fulfillment of the requirements for the
Degree of Computer Science and Engineering
in Computer Science Department

Supervisor : Prof. Rajdeep Niyogi

April, 2025

INDIAN INSTITUTE OF TECHNOLOGY
ROORKEE, ROORKEE

Candidate's Declaration

We hereby certify that the work which is being presented in this report entitled "Multi-Agent Deep Reinforcement Learning for Coverage Maximization" in partial fulfillment of the requirements for the award of the degree of Computer Science and Engineering and submitted in the Department of Computer Science and Engineering of the Indian Institute of Technology Roorkee is an authentic record of our own work carried out during the period from December, 2024 to April, 2025 under the supervision of Prof. Rajdeep Niyogi, Professor.

The matter presented in this thesis has not been submitted by us for the award of any other degree of this or any other Institute.

22125030 - Roopam Taneja
22114098 - Vraj Tamakuwala

This is to certify that the above statement made by the candidates is correct to the best of our knowledge.

Date:

Supervisor
(Prof. Rajdeep Niyogi)

Abstract

This report presents a comprehensive exploration of Reinforcement Learning (RL), progressing from foundational concepts to advanced multi-agent systems, culminating in the design and implementation of a custom learning framework for decentralized UAV coordination. We begin by studying the theoretical underpinnings of RL, Markov Decision Processes (MDPs), and extend into Multi-Agent Reinforcement Learning (MARL) through the lens of stochastic games and centralized training with decentralized execution (CTDE).

The core contribution of this work lies in addressing the long-term communication coverage problem using a team of Unmanned Aerial Vehicles (UAVs), modelled as learning agents in a partially observable environment. Our custom 2D grid environment simulates realistic constraints, including limited energy, partial observations, and communication range restrictions. The learning framework builds on the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm, augmented with Convolutional Neural Networks (CNNs) for spatial feature extraction and Conditional Value at Risk (CVaR) for risk-averse policy optimization.

We define a multi-objective reward structure that balances spatial coverage, geographical fairness, and energy efficiency. Through detailed experimentation and visual analysis, we demonstrate the effectiveness of our approach, showing strong correlations between fairness, energy-aware behavoiur, and coverage quality. This report not only showcases a novel application of MARL techniques to real-world scenarios but also provides a foundation for future work in scalable, robust, and interpretable autonomous agent design.

Our implementation can be found on GitHub.

Acknowledgements

We would like to express our sincere gratitude to all those who supported and guided us throughout the course of this project.

First and foremost, we thank our faculty mentor(s) and the department for providing us with the opportunity and resources to explore this topic in depth. Their valuable feedback and encouragement played a key role in shaping the direction and outcome of our work.

We also extend our appreciation to the authors of the research paper *"Distributed Energy-Efficient Multi-UAV Navigation for Long-Term Communication Coverage by Deep Reinforcement Learning"* [10], which served as the foundational reference for our study. Their insights inspired us to replicate, adapt, and further build upon their ideas in a new multi-agent learning context.

Lastly, we are grateful to each other for the consistent collaboration, shared learning, and combined effort that went into completing this project with integrity and enthusiasm.

22125030 - Roopam Taneja
22114098 - Vraj Tamakuwala

Date:

Contents

List of Figures **vii**

List of Algorithms **viii**

1 Introduction **1**

1.1 Understanding RL and Its Importance 1
1.2 Characteristics of RL 2
1.3 Some key terms 2
1.4 About RL Agents 4
1.5 Challenges in RL 5

2 Value-Based Reinforcement Learning **6**

2.1 Temporal Difference (TD) Learning 6
2.2 Q-Learning 8
2.3 Deep Q-Networks (DQN) 9
2.4 Double DQN (DDQN) 10
2.5 SARSA 12
2.6 Conclusion 13

3 Policy-Based Reinforcement Learning **14**

3.1 What is Policy-Based Learning? 14
3.2 REINFORCE Policy Gradient 15
3.3 Vanilla Policy Gradient (VPG) 15
3.4 Conclusion 16

4 Actor-Critic Methods **17**

4.1 Introduction 17
4.2 Working Mechanism 18
4.3 Types of Actor-Critic Algorithms 20
4.4 Proximal Policy Optimization (PPO) 22

5 DDPG and Multi-Agent RL **25**
5.1 Deep Deterministic Policy Gradient (DDPG) 25
5.2 Multi-Agent Reinforcement Learning (MARL) 28
5.3 Multi-Agent DDPG (MADDPG) 31

6 MADDPG Application **34**
6.1 Introduction 34
6.2 Problem Statement 35
6.3 Environment Design 37
6.4 Proposed Solution 39
6.5 Implementation Setup 43
6.6 Results and Analysis 45
6.7 Conclusion 48

Bibliography **49**

List of Figures

1.1 RL-Venn Diagram 1
1.2 Agent-Environment relation 3
1.3 Markov-State definition 4

2.1 Comparison of two images 6
2.2 DP vs TD backup 7
2.3 Epsilon-Greedy policy 8
2.4 Q-Learning loss function 9
2.5 Q-Learning update rule 9
2.6 Q-Learning vs DQN 10
2.7 DDQN update rule 11

4.1 Actor-Critic Architecture 17
4.2 Actor v/s Critic 18
4.3 A3C Architecture 21
4.4 PPO Clip function in action 22

5.1 DDPG Flowchart 26
5.2 Approaches to MARL 30
5.3 Overview of CTDE paradigm 30
5.4 MADDPG Framework in CTDE paradigm 31

6.1 Multi-UAV navigation coverage scenario [10] 35
6.2 Architecture of MADDPG Network [10] 39
6.3 CNN Architecture 41
6.4 State images while training 45
6.5 Total Coverage plotted against the Total Fairness measured at the end of each episode during training 46
6.6 Total Fairness plotted against the Average energy Efficiency measured at the end of each episode during training 46
6.7 Total Fairness per episode during training 47

List of Algorithms

1 Q-learning Algorithm 9
2 DDQN Algorithm 11
3 SARSA Algorithm 12
4 VPG Algorithm 16
5 PPO Algorithm 23
6 Deep Deterministic Policy Gradient 27
7 Multi-Agent Deep Deterministic Policy Gradient for N agents 33
8 Distributed Energy-Efficient Multi-UAV Navigation (Training Process) 40
9 Distributed Energy-Efficient Multi-UAV Navigation (Testing Process) 40

Chapter 1

Introduction

1.1 Understanding RL and Its Importance

Reinforcement Learning (RL) is a branch of machine learning that trains intelligent agents to make sequential decisions by interacting with an environment. Unlike supervised learning, where labeled data guide learning, RL operates through a trial-and-error process, using feedback as rewards or penalties. The agent takes actions in a given state, observes the resulting consequences, and refines its strategy to maximize cumulative rewards over time. This learning paradigm enables agents to dynamically adapt and improve their performance, making RL suitable for complex, real-world scenarios where predefined labels or instructions are impractical.

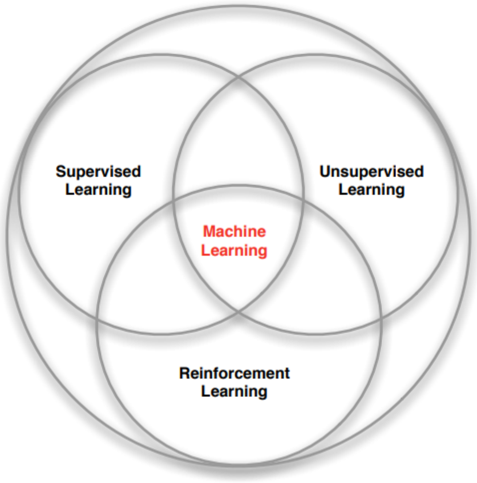


FIGURE 1.1: RL-Venn Diagram

The significance of RL lies in its ability to solve decision-making problems that involve long-term consequences. Many real-world applications, such as robotics, autonomous vehicles, game playing, and financial market strategies, require intelligent agents to make interdependent decisions rather than isolated ones. Since RL considers the delayed impact of actions, it enables agents to learn optimal policies that balance short-term rewards with long-term gains. This makes RL particularly effective for applications where planning and adaptation are essential.

Another key reason RL is widely used is its ability to handle high-dimensional and dynamic environments. Traditional rule-based systems struggle with uncertainty and variability, but RL agents can explore different strategies and adapt their behavior based on evolving conditions. This adaptability is crucial in areas such as healthcare (e.g., optimizing treatment plans), logistics (e.g., supply chain management), and telecommunications (e.g., network traffic optimization). RL is becoming an increasingly powerful tool for solving complex decision-making problems across diverse fields as computing power and algorithms advance.

1.2 Characteristics of RL

Several key features distinguish RL from other machine-learning approaches:

- ABSENCE OF SUPERVISORY SIGNALS: RL agents learn from a reward signal rather than direct supervision.
- DELAYED FEEDBACK: Rewards are not immediate; the consequences of actions may only become apparent after some time.
- TEMPORAL CONSIDERATIONS: The sequential nature of RL means that time plays a crucial role, with data being non-independent and identically distributed (non-i.i.d).
- ACTION-DEPENDENT DATA: An agent's actions influence the subsequent data it receives as the agent interacts with and affects the environment.

A classic example illustrating RL is a robot that navigates through a maze, where it must learn to make decisions that lead to the goal.

1.3 Some key terms

1. **The Concept of Rewards**
   In RL, a reward (R_t) is a scalar feedback signal that indicates the immediate benefit of an action taken at time t. The primary objective of an RL agent is to maximize the cumulative reward over time. This principle aligns with the Reward Hypothesis, which posits that all goals can be described as the maximization of expected cumulative reward.

2. **Sequential Decision Making**
   RL involves making a series of decisions to achieve long-term objectives. Key aspects include:
   - MAXIMIZING FUTURE REWARDS: Agents aim to select actions that optimize total future rewards, not just immediate gains.
   - LONG-TERM CONSEQUENCES: Actions may have repercussions that extend into the future, requiring foresight and planning.
   - DELAYED GRATIFICATION: Agents might need to forgo immediate rewards to achieve greater benefits later, exemplifying the trade-off between short-term and long-term gains.
3. **Agent and Environment Dynamics**
   An RL system comprises an agent and an environment:
   - AGENT: The decision-maker who selects actions based on a policy π derived from observations and rewards.
   - ENVIRONMENT: The external system with which the agent interacts, providing observations and rewards in response to the agent's actions.

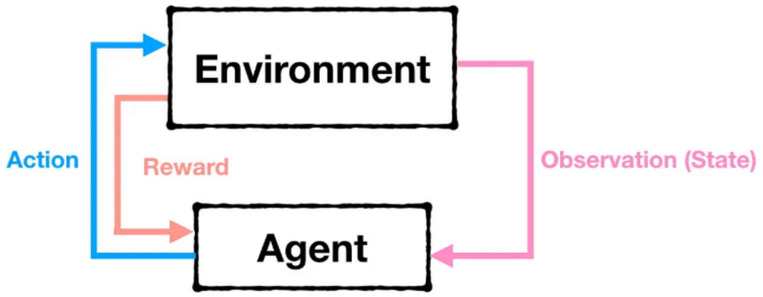
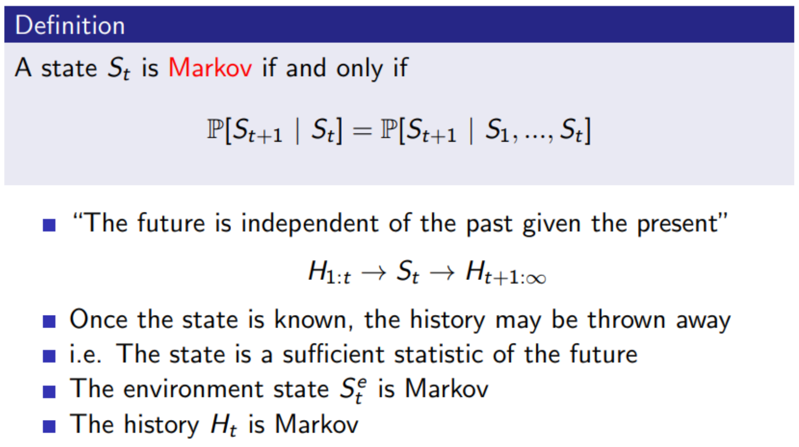


FIGURE 1.2: Agent-Environment relation

4. **History and State**
   HISTORY (H_t): A complete sequence of observations O_t, actions A_t, and rewards R_t up to time t.
   $$H_t = O_1, R_1, A_1, ..., A_{t-1}, O_t, R_t$$
   STATE (S_t): A function of the history that encapsulates all relevant information needed to determine future dynamics:
   $$S_t = f(H_t)$$
   States can be categorized as:

- ENVIRONMENT STATE (S_t^e): The actual state of the environment, which may be partially or fully observable by the agent.
- AGENT STATE (S_t^a): The agent's internal representation of the environment state is used to make decisions.
- INFORMATION STATE (MARKOV STATE): A state that contains all necessary information from history to predict future states and rewards, adhering to the Markov property.



Definition

A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

- "The future is independent of the past given the present"

$$H_{1:t} \rightarrow S_t \rightarrow H_{t+1:\infty}$$

- Once the state is known, the history may be thrown away
- i.e. The state is a sufficient statistic of the future
- The environment state S_t^e is Markov
- The history H_t is Markov

FIGURE 1.3: Markov-State definition

1.4 About RL Agents

Components of an RL Agent

An RL agent typically comprises several components:

- POLICY (π): A strategy that maps states to actions. Policies can be deterministic ($a = \pi(s)$) or stochastic ($\pi(a|s) = P[A_t = a|S_t = s]$).
- VALUE FUNCTION ($v(s)$): A function that estimates the expected cumulative reward from a given state, guiding the agent toward optimal actions.

$$v(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s]$$

- MODEL: An internal representation of the environment's dynamics used for planning and predicting future states and rewards.

Categorization of RL Agents

RL agents can be classified based on their learning strategies:

- VALUE-BASED: Agents that focus on estimating value functions to derive policies.
- POLICY-BASED: Agents that directly optimize the policy without explicitly using value functions.
- ACTOR-CRITIC: Agents that combine both value and policy-based approaches, utilizing an actor (policy) and a critic (value function) to improve learning efficiency.

Additionally, agents may be:

- MODEL-FREE: Agents that learn policies and/or value functions without an explicit model of the environment.
- MODEL-BASED: Agents that incorporate a model of the environment to facilitate planning and decision-making.

1.5 Challenges in RL

Several core challenges arise in RL:

1. **Learning vs. Planning:** The agent learns optimal policies through direct interaction with an initially unknown environment. In contrast, it uses a known model of the environment to compute optimal policies without real-world interaction.
2. **Exploration vs. Exploitation:** The agent seeks new environmental information to improve future decision-making by exploring the environment while utilising known information to maximize immediate rewards. Balancing exploration and exploitation is crucial, as excessive exploration can lead to suboptimal rewards, while insufficient exploration may prevent the discovery of better strategies.
3. **Prediction vs. Control:** Prediction is evaluating the future given a policy, while Control is optimizing the future by finding the best policy.

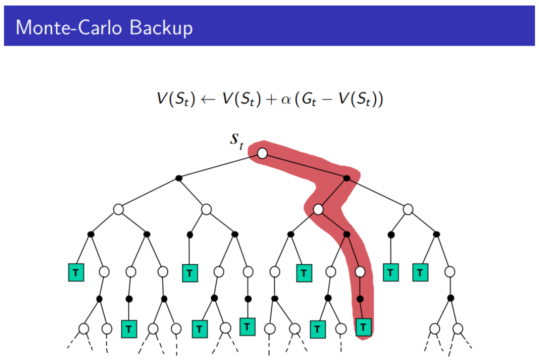
Chapter 2

Value-Based Reinforcement Learning

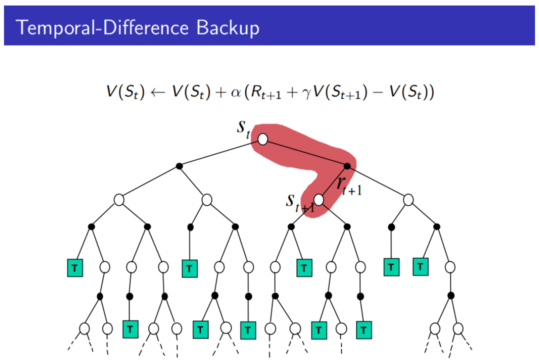
Value-Based Reinforcement Learning (RL) focuses on estimating the value of actions or states to inform decision-making. This approach involves learning value functions that predict the expected cumulative reward of states or state-action pairs, guiding agents toward optimal policies. Key methods in this paradigm include *Temporal Difference (TD) Learning*, *Q-Learning*, *Deep Q-Networks (DQN)*, *Double DQN (DDQN)*, and *SARSA*.

2.1 Temporal Difference (TD) Learning

The Temporal Difference (TD) algorithm is a key method in Reinforcement Learning (RL) that combines ideas from *Monte Carlo* methods and *Dynamic Programming (DP)* to learn optimal policies and value functions.[15]
Unlike Monte Carlo methods, which wait until the end of an episode to compute updates, TD methods update value estimates based on the next step. This means it uses its own current estimates (bootstrapping) to update itself. It does not require a complete model of the



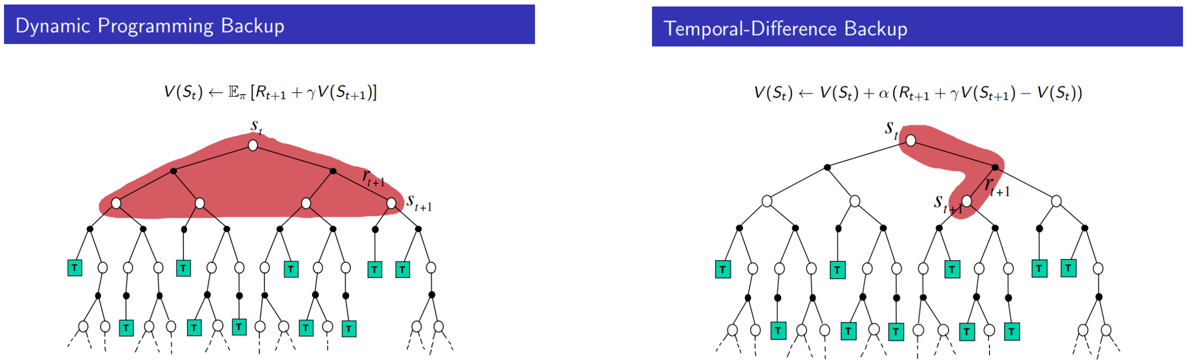
(A) Monte-Carlo backup


Temporal-Difference Backup
$V(S_t) \leftarrow V(S_t) + \alpha\,(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
s_t
r_{t+1}
s_{t+1}
T

(B) TD-backup

FIGURE 2.1: Comparison of two images

environment like DP. Instead, it learns directly from the agent's interactions with the environment. It updates value functions incrementally after each step, making it computationally more efficient than both of these methods. For a value function $V(s)$ (state value function),


Dynamic Programming Backup
$V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})]$
s_t
r_{t+1}
s_{t+1}
T
Temporal-Difference Backup
$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
s_t
r_{t+1}
s_{t+1}
T

(A) DP backup (B) TD-backup

FIGURE 2.2: DP vs TD backup

the TD update rule is:

$$V(s) \leftarrow V(s) + \alpha \left[R_{t+1} + \gamma V(s') - V(s)\right]$$

Here, $V(s)$ is the value estimate of state s, α is the learning rate, R_{t+1} is the reward received after transitioning from state s to state s', and γ is the discount factor. This method enables agents to learn directly from raw experience without requiring a model of the environment.

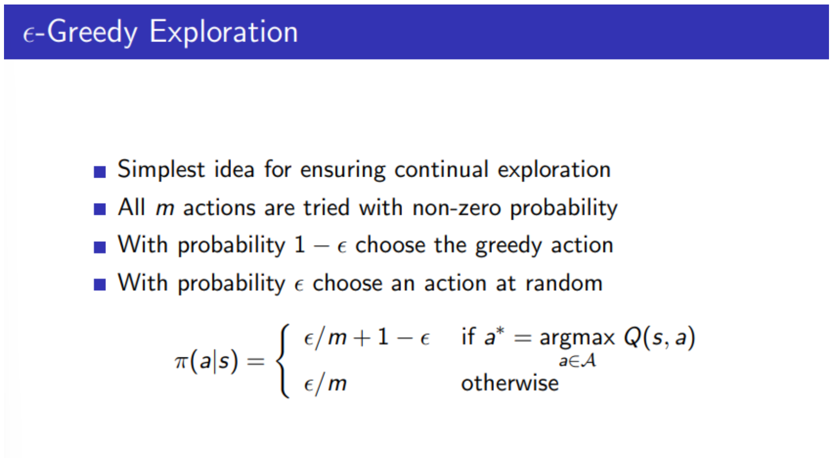
The term $\delta = r + \gamma V(s') - V(s)$ is called the *TD error*. It measures the difference between the current estimate and the updated value estimate. There are various types of TD-Learning algorithms.

- TD(0): The simplest form of TD learning that updates values based on a single step of experience.
- TD(λ): Extends TD(0) by considering multiple steps of experience through the eligibility trace, blending TD(0) and Monte Carlo methods.
- Q-Learning: A TD method used in off-policy learning, where updates are based on the optimal policy (not the agent's current policy).
- SARSA: A TD method used in on-policy learning, where updates are based on the agent's actual policy.

We'll see the later two in upcoming sections.

2.2 Q-Learning

Q-Learning is an off policy, model-free, value-based reinforcement learning algo which uses TD(0). Being model free implies that the agent does not know state transition probabilities or rewards in Q-learning. The agent only discovers that there is a reward for going from one state to another via a given action when it does so and receives a reward. Similarly, it only figures out what transitions are available from a given state by ending up in that state and looking at its options.
Current Q values can be stored in a table, called the *q-table*, and updated through several episodes. The behaviour policy, used to train agents, is ϵ - *Greedy Policy* [15] while the target policy, to test the agent, is *Greedy policy*. This ϵ - *Greedy Policy* encourages the agent to explore



ϵ-Greedy Exploration

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\text{argmax}}\ Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

FIGURE 2.3: Epsilon-Greedy policy

more, which on longer run may lead to more positive rewards.

We apply the Bellman Equation to estimate the best $Q(s,a)$ in the Q-learning algorithm. The procedure is described as following.

1. In time t, the Agent takes an action a_t in given current state s_t. Then, the Agent gets a reward, denoted R_{t+1}, when it arrives to next state s_{t+1}.

2. In according to $Q(s,a)$, we can pre-calculate the maximum Q-value in given state s_{t+1} by considering all possible actions it can take. The discount factor γ is for weighting the importance of maximum next-state Q-value.

3. Combine step 1 and step 2, the estimated best $Q(s,a)$, or TD-target, is completed.

The Q-Learning update rule is:

Here, $\max_{a'} Q(s',a')$ represents the maximum estimated value of the next state-action pair, assuming the agent acts optimally from state s'. This off-policy nature allows Q-Learning

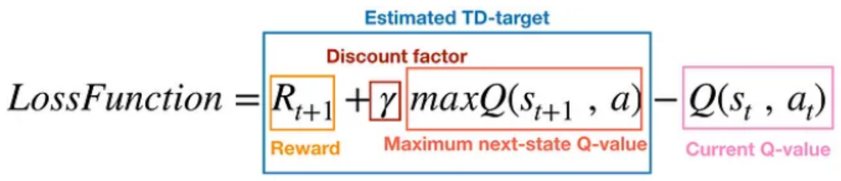

Estimated TD-target
Discount factor
$LossFunction = R_{t+1} + \gamma\, maxQ(s_{t+1}, a) - Q(s_t, a_t)$
Reward
Maximum next-state Q-value
Current Q-value

FIGURE 2.4: Q-Learning loss function

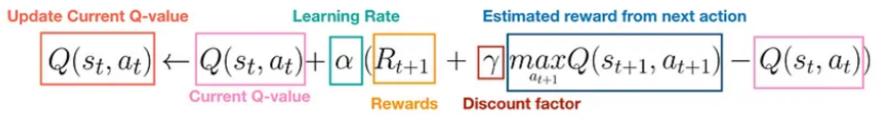

Update Current Q-value
Learning Rate
Estimated reward from next action
$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t))$
Current Q-value
Rewards
Discount factor

FIGURE 2.5: Q-Learning update rule

to learn the optimal policy while following an exploratory policy, enabling more aggressive learning than SARSA [1]. The underlying algorithm :

Algorithm 1 Q-learning Algorithm

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
repeat
  Initialize $S$
  repeat
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $A$, observe $R, S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$
  until $S$ is terminal
until episodes remain =0
```

Here, *Learning Rate (α)* is a hyper-parameter for controlling the convergent speed of updating procedure and *Discount Factor (γ)* is also a hyper-parameter for weighting the importance of estimated future reward. The value of discount factor is between 0 and 1. The more closer to 1, the more important the future reward is.

2.3 Deep Q-Networks (DQN)

Traditional Q-Learning struggles with high-dimensional state spaces due to the need for a comprehensive Q-table. Deep Q-Networks address this limitation by approximating the

Q-value function using deep neural networks. A DQN inputs the current state and outputs Q-values for all possible actions. Key innovations in DQN include *experience replay* and *target*

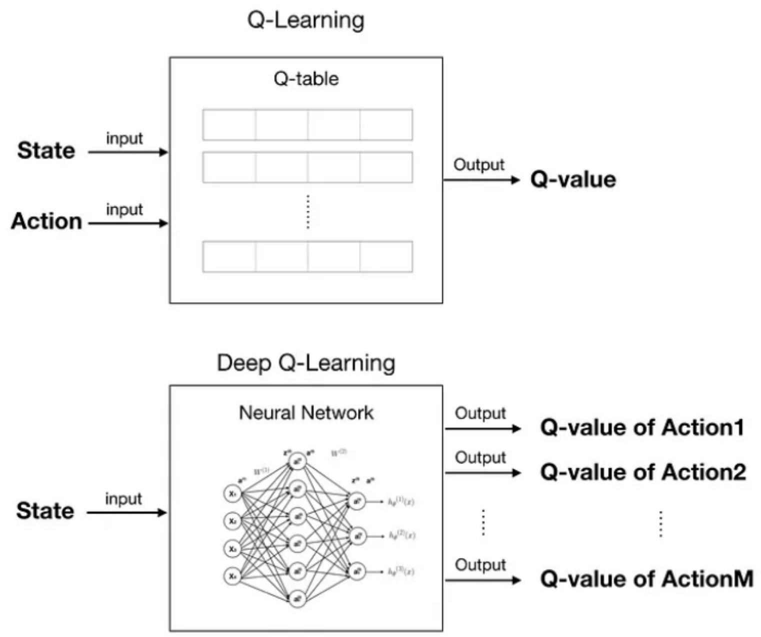


FIGURE 2.6: Q-Learning vs DQN

networks: **Experience Replay:** Stores agent experiences and samples mini-batches to break correlation between consecutive samples, improving learning stability. This process is key to the agent's learning in environments with high-dimensional state spaces. **Target Networks:** Uses a separate network to generate target Q-values, reducing oscillations and divergence during training.

These techniques enable DQNs to perform well in complex environments with large state spaces. [3]

2.4 Double DQN (DDQN)

In basic Q-learning, the optimal policy of the Agent is always to choose the best action in any given state. The assumption behind the idea is that the best action has the maximum expected/estimated Q-value. Unfortunately, the best action often has smaller Q-values than the non-optimal ones in most cases. According to the optimal policy in basic Q-Learning, the Agent tends to take the non-optimal action in any given state only because it has the maximum Q-value. Such problem is called the overestimations of action value (Q-value).

Double DQN addresses this by decoupling the action selection and evaluation steps. It uses two different action-value functions, Q and Q', as estimators. Even if Q and Q' are noisy, these noises can be viewed as uniform distribution. That is, this algorithm solves the problem of overestimations of action value.[14] The update procedure is slightly different from the basic version.

Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \underbrace{\boxed{Q'(s_{t+1}, \boxed{a})}}_{\text{estimated/expected Q-value}} - Q(s_t, a_t))$$

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = \boxed{Q'(s_{t+1}, \boxed{a})}$$

FIGURE 2.7: DDQN update rule

Q function selects the best action a with maximum Q-value of next state: $a = \max_a Q(s_{t+1}, a)$
Q' function calculates expected Q-value using a selected above: $q_{estimated} = Q'(s_{t+1}, a)$
Update Q function by using the expected Q-value of Q' function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q'(s_{t+1}, a) - Q(s_t, a_t))$$

Algorithm 2 DDQN Algorithm

```
Initialize $Q^A, Q^B, s$
**repeat**
  Choose $a$, based on $Q^A(s, \cdot)$ and $Q^B(s, \cdot)$, observe $r, s'$
  Choose (e.g., randomly) either UPDATE(A) or UPDATE(B)
  **if** UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s', a)$
    $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a)[r + \gamma Q^B(s', a^*) - Q^A(s, a)]$
  **else if** UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s', a)$
    $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a)[r + \gamma Q^A(s', b^*) - Q^B(s, a)]$
  **end if**
  $s \leftarrow s'$
**until** end =0
```

2.5 SARSA

SARSA, an acronym for *State-Action-Reward-State-Action*, is an on-policy reinforcement learning algorithm. The name SARSA reflects the sequence of events the agent experiences when transitioning from state S to S'.

- (S) State: Represents the environment's current condition.
- (A) Action: The decision made by the agent in the current state.
- (R) Reward: The feedback received after taking an action.
- (S') Next State: The state the environment transitions to after the action.
- (A') Next Action: The next action the agent decides to take in the new state.

In SARSA, the agent updates its action-value function, denoted as $Q(s,a)$, based on its action, adhering to its current policy. This characteristic classifies SARSA as an on-policy algorithm, meaning it evaluates and improves its current policy. The Q-value update rule in SARSA is expressed as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R_{t+1} + \gamma Q(s',a') - Q(s,a)\right]$$

In this equation, $Q(s,a)$ represents the action-value function for state s and action a, a' is the action taken in state s' according to the current policy, and the other terms are as previously defined. This update rule ensures that the Q-value for a given state-action pair is adjusted based on the reward received and the estimated Q-value of the subsequent state-action pair, promoting learning that is consistent with the agent's actual experiences. **[?]**

Algorithm 3 SARSA Algorithm

```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$
for each episode do
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  for each step of episode do
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$, $A \leftarrow A'$
  end for
end for=0
```

While both SARSA and Q-learning are temporal difference learning methods used to find optimal policies, they differ in their approaches:

- POLICY TYPE: SARSA is on-policy, updating its Q-values using the action actually taken by the current policy. In contrast, Q-learning is off-policy, updating its Q-values using the maximum possible reward of the next state, regardless of the agent's current policy.
- EXPLORATION VS. EXPLOITATION: Due to its on-policy nature, SARSA tends to be more conservative, as it accounts for the policy's exploration strategy during learning. Q-learning, being off-policy, can be more aggressive, as it assumes the agent will always choose the optimal action in the future.

These differences can lead to varying performance depending on the environment and the specific task at hand.

2.6 Conclusion

Value-Based RL methods provide a framework for learning optimal policies by estimating value functions. Starting with foundational algorithms like TD Learning and Q-Learning, advancements such as DQN and DDQN have enabled the application of these methods to complex, high-dimensional environments. Understanding these algorithms and their interconnections is crucial for developing effective RL agents.

Chapter 3

Policy-Based Reinforcement Learning

Unlike value-based approaches, which rely on estimating action-value functions $Q(a, s)$, policy-based methods learn the policy π directly, making them particularly effective for environments with continuous action spaces or stochastic policies.

3.1 What is Policy-Based Learning?

In policy-based learning, an agent maintains a policy function $\pi(a|s, \theta)$, parameterized by θ, that determines the probability of taking an action a in a given state s. The objective is to optimize the policy parameters θ such that the expected cumulative reward is maximized:

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t\right]$$

where γ is the discount factor and R_t is the reward at time step t. The gradient of $J(\theta)$ can also be expressed as:

$$\nabla_\theta J(\theta) = \mathbf{E}\left[\sum_{t=0}^{\infty} \psi_t \nabla_\theta log \pi_\theta(a_t|s_t)\right]$$

where ψ_t can take different forms depending on the algorithm used. It could be $(G_t = \sum_{t'=t}^{\infty} R_{t'})$ or with any baseline function [17] like $G_t - b(s_t)$ with $b(s_t)$ being a function independent of actions. To achieve this, policy gradient methods use the gradient ascent algorithm. The parameters θ are updated in the direction of the gradient of the expected return concerning the policy parameters. The update rule can be written as:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

where α is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the expected return with respect to the policy parameters and vary depending on the algorithm being used [2]. We'll discuss some of the major algorithms in upcoming sections.

3.2 REINFORCE Policy Gradient

The REINFORCE method is a foundational technique in policy gradient methods within reinforcement learning. It optimizes the policy by adjusting the policy parameters to maximize the expected cumulative reward. Here,

$$\psi_t = G_t = \sum_{t'=t}^{\infty} R_{t'}$$

It is a Monte Carlo method, meaning it samples complete trajectories from the environment without bootstrapping, unlike other methods such as Temporal Difference Learning. This approach makes it an on-policy method, where samples obtained under different policies cannot be used to improve the existing policy.

3.3 Vanilla Policy Gradient (VPG)

VPG is an on-policy algorithm that can be used for environments with discrete or continuous action spaces. Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right]$$

where τ is a trajectory and A^{π_θ} is the advantage function for the current policy [13]. The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

VPG trains a stochastic policy in an on-policy way. This means it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on initial conditions and the training procedure. The policy typically becomes progressively less random throughout training, as the update rule encourages it to exploit rewards already found. This may cause the policy to get trapped in local optima.

Algorithm 4 VPG Algorithm

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Estimate policy gradient : $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$.
    Compute policy update, $\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k$, using algorithms like standard gradient ascent or Adam.
    Fit value function by regression on mean-squared error:
    $\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$
    typically via some gradient descent algorithm.
end for=0
```

3.4 Conclusion

Policy-based reinforcement learning provides a robust framework for solving complex RL problems, particularly those involving continuous actions and stochastic environments. Directly optimising the policy enables more efficient learning in many real-world applications, paving the way for advanced reinforcement learning techniques such as Actor-Critic methods and Proximal Policy Optimization (PPO).

Chapter 4

Actor-Critic Methods

4.1 Introduction

In policy-based RL, the optimal policy is computed by manipulating policy directly, and value-based function implicitly finds the optimal policy by finding the optimal value function. Policy-based RL is effective in high dimensional & stochastic continuous action spaces, and learning stochastic policies. At the same time, value-based RL excels in sample efficiency and stability.

The main challenge of policy gradient RL is the high gradient variance. The standard approach to reduce the variance in gradient estimate is to use baseline function $b(s_t)$. There are lots of concern about adding the based line will invite the bias in the gradient estimate. It has been proved that the baseline doesn't bring the basis to the gradient estimate (refer this article by Daniel Takeshi for that proof).

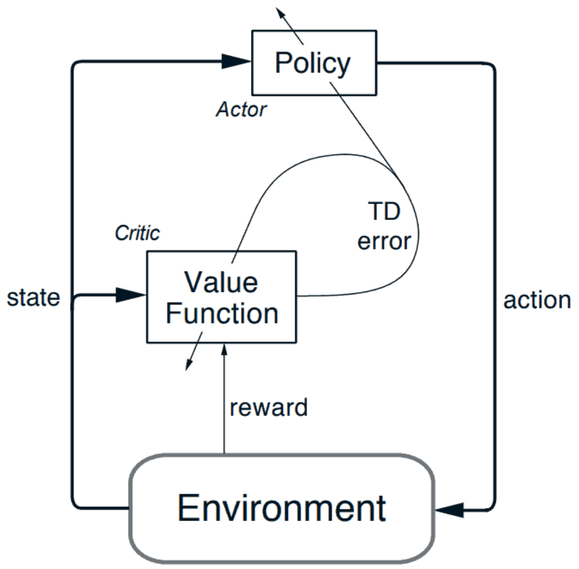


FIGURE 4.1: Actor-Critic Architecture

Actor-Critic Algorithm is a type of reinforcement learning algorithm that combines aspects of both policy-based methods (Actor) and value-based methods (Critic). This hybrid approach is designed to address the limitations of each method when used individually [7].

- ACTOR: Determines the action to take based on the current state by learning a policy function $\pi_\theta(a|s)$, where θ represents the parameters of the policy.
- CRITIC: Evaluates the action taken by the actor by estimating the value function ($V^\pi(s)$) or the action-value function $Q^\pi(s,a)$, providing feedback to the actor to improve future decisions.

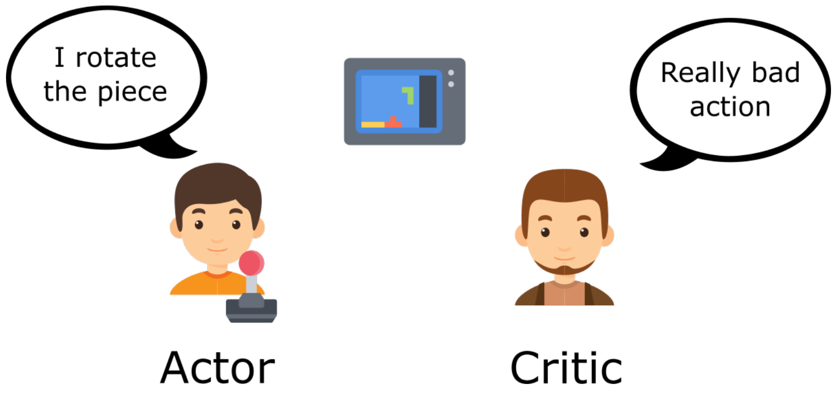


FIGURE 4.2: Actor v/s Critic

4.2 Working Mechanism

The Actor-Critic algorithm operates through the interaction of two primary components: the **actor**, responsible for policy representation, and the **critic**, which estimates value functions to guide the actor's learning. This synergy allows for more stable and efficient learning compared to pure policy-based or value-based approaches.

The **actor** maintains a parameterized policy $\pi_\theta(a|s)$, which outputs the probability of selecting action a given state s. The objective is to maximize the expected cumulative reward $J(\theta)$, which is defined as:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

where γ is the discount factor, r_t is the reward at time step t, and T represents the time horizon of the episode. By optimizing $J(\theta)$, the actor learns to select actions that yield higher long-term rewards.

The **critic** estimates the value function $V^\pi(s)$, which represents the expected return starting from state s and following policy π. Alternatively, it can estimate the action-value function $Q^\pi(s, a)$, which indicates the expected return when taking action a from state s and subsequently following policy π. The critic's role is to evaluate the actions taken by the actor and provide feedback for improvement.

To enhance policy learning, the actor updates its policy parameters θ using a policy gradient method. The policy gradient is computed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot Q^\pi(s, a)\right]$$

where $Q^\pi(s, a)$ represents the action-value function. This guides the actor in adjusting its policy to favor actions that yield higher expected returns.

A key component of the Actor-Critic framework is the *Temporal Difference (TD) error*, which quantifies the difference between predicted and observed values. The TD error is computed as:

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

This error serves as a feedback signal for both the actor and the critic, enabling continuous improvement. To refine its value estimation, the critic updates its parameters ϕ by minimizing the TD error using the following update rule:

$$\phi \leftarrow \phi + \alpha_c \delta_t \nabla_\phi V^\pi(s_t)$$

where α_c is the critic's learning rate. By iteratively reducing the TD error, the critic improves its estimation of the value function.

Simultaneously, the actor adjusts its policy parameters θ using the TD error as a feedback signal. The actor update rule is:

$$\theta \leftarrow \theta + \alpha_a \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

where α_a is the actor's learning rate. This step ensures that the policy evolves towards selecting actions that maximize long-term rewards [7].

Through iterative updates of both the actor and the critic, the Actor-Critic algorithm gradually refines the policy, enabling the agent to make progressively better decisions over time. This adaptive learning mechanism allows reinforcement learning agents to efficiently handle complex and dynamic environments.

4.3 Types of Actor-Critic Algorithms

Advantage Actor-Critic (A2C) and *Asynchronous Advantage Actor-Critic (A3C)* are two significant algorithms in the realm of reinforcement learning, both stemming from the actor-critic framework. They integrate policy-based and value-based methods to enhance learning efficiency and performance.

Advantage Actor-Critic (A2C) :
We can stabilize learning further by using the *Advantage function* ($A^{\pi}(s,a)$) as Critic instead of the Action-Value function.

The idea is that the Advantage function calculates how better taking that action at a state is compared to the average value of the state. It's subtracting the mean value of the state from the state action pair:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

In other words, this function calculates the *extra reward we get if we take this action at that state compared to the mean reward we get at that state* [4]. The extra reward is what's beyond the expected value of that state.

- If $A^{\pi}(s,a) > 0$: our gradient is pushed in that direction.
- If $A^{\pi}(s,a) < 0$: our gradient is pushed in the opposite direction (our action does worse than the average value of that state)

The problem with implementing this advantage function is that it requires two value functions - $Q(s,a)$ and $V(s)$. Fortunately, we can use the TD error as a good estimator of the advantage function, where $Q^{\pi}(s_t,a_t) = r_t + \gamma V^{\pi}(s_{t+1})$ which makes the overall Advantage function as:

$$A^{\pi}(s_t,a_t) = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$$

Asynchronous Actor-Critic Agent (A3C) :
The A3C algorithm is one of RL's state-of-the-art algorithms, which beats DQN in few domains (for example, Atari domain, as stated in a classic paper by Google Deep Mind).

Asynchronous stands for the principal difference of this algorithm from DQN, where a single neural network interacts with a single environment. On the contrary, in this case, we've got a global network with multiple agents having their own set of parameters. It creates every agent's situation interacting with its environment and harvesting the different and unique learning experience for overall training. That also deals partially with RL sample correlation, a big problem for neural networks, which are optimized under the assumption that input samples are independent of each other (not possible in games). Thus, it creates a

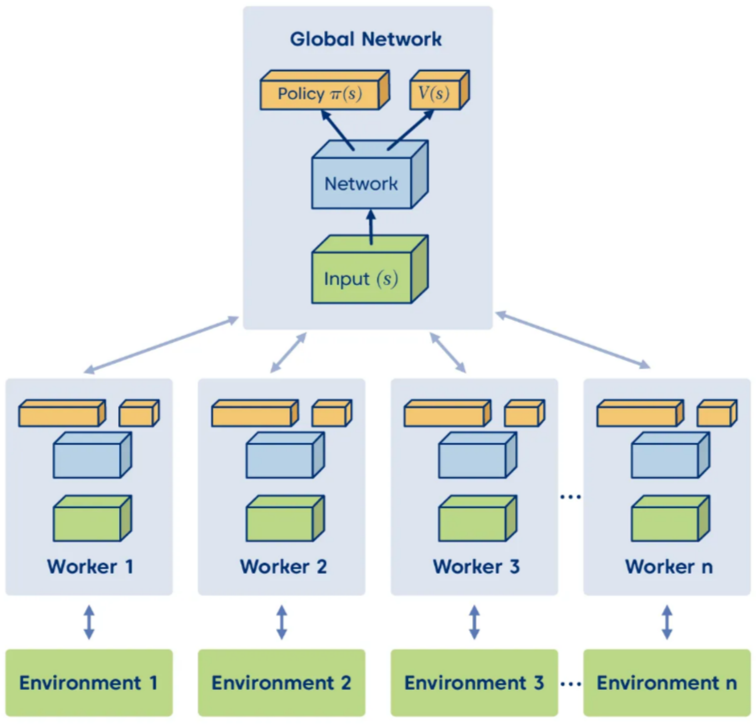


FIGURE 4.3: A3C Architecture

complementary situation for an agent to gain the best experience of fast learning.

Key Differences Between A2C and A3C :

- SYNCHRONIZATION: A2C performs synchronous updates, where all agents wait for each other to complete their interactions before updating the global parameters. In contrast, A3C allows asynchronous updates, enabling agents to update the global parameters independently without waiting for others.
- TRAINING STABILITY AND SPEED: A3C's asynchronous nature often leads to faster training due to diverse and decorrelated experiences from multiple agents. However, A2C's synchronous updates can provide more stability during training, as the updates are based on aggregated experiences from all agents.

Both A2C and A3C have been successfully applied in various domains, including playing Atari games and navigating 3D mazes, demonstrating their effectiveness in complex reinforcement learning tasks.

4.4 Proximal Policy Optimization (PPO)

The idea with Proximal Policy Optimization (PPO) is that we want to improve the training stability of the policy by limiting the change you make to the policy at each training epoch: *we want to avoid having too large of a policy update* [5]. Why?

- We know empirically that smaller policy updates during training are more likely to converge to an optimal solution.
- A too-big step in a policy update can result in falling “off the cliff” (getting a bad policy) and taking a long time or even having no possibility to recover.

So with PPO, we update the policy conservatively. To do so, we need to measure how much the current policy changed compared to the former one using a ratio calculation between the current and former policy. And we clip this ratio in a range $[1-\epsilon, 1+\epsilon]$, meaning that we remove the incentive for the current policy to go too far from the old one (hence the proximal policy term). A new function is designed to avoid destructively large weights updates:

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

This equation is called *PPO's Clipped Surrogate objective function*, where,

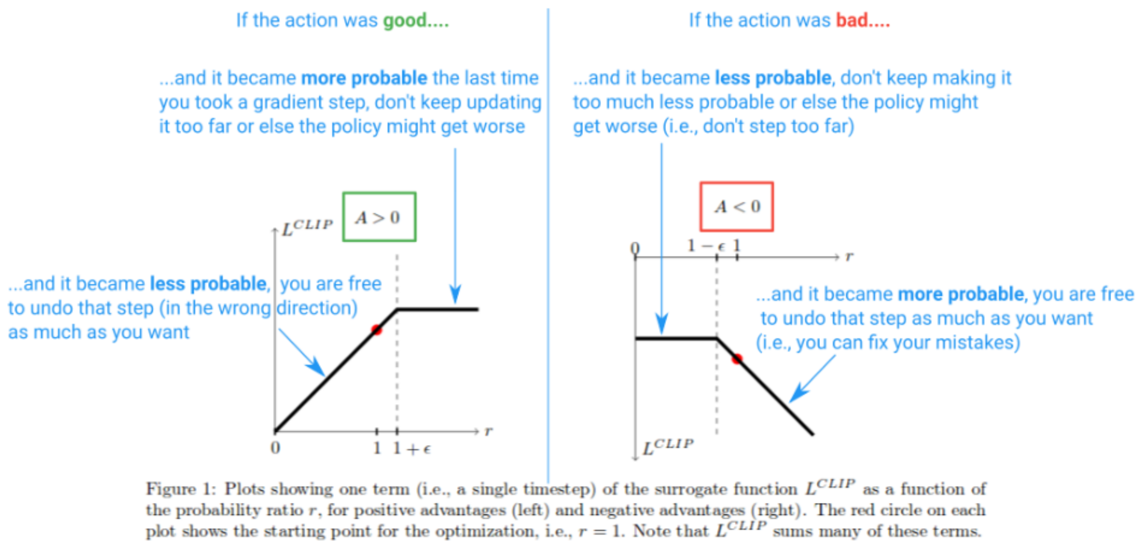


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

FIGURE 4.4: PPO Clip function in action

- *The Ratio function* $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ denotes the probability ratio between the current and old policy:

  - If $r_t(\theta) > 1$, the action a_t at state s_t is more likely in the current policy than the old policy.
  - If $r_t(\theta)$ is between 0 and 1, the action is less likely for the current policy than for the old one.

- With the *Clipped Surrogate Objective function*, we have two probability ratios, one non-clipped and one clipped in a range between $[1-\epsilon, 1+\epsilon]$, ϵ is a hyperparameter that helps us to define this clip range.
- Taking the minimum of the clipped and non-clipped objective means we'll select either the clipped or the non-clipped objective based on the ratio and advantage situation.

We only clip if the objective function would improve. If the policy is changed in the opposite direction such that $L^{CLIP}(\theta)$ decreases, $r(\theta)$ is not clipped (since there is min). This is because it was a step in the wrong direction (e.g., the action was good but we accidentally made it less probable). If we had not included the min in the objective function, these regions would be flat (gradient = 0, no update to theta) and we would be prevented from fixing mistakes. [6]

Thus, the clipping limits the effective change you can make at each step to improve stability, and the *min* allows us to fix our mistakes in case we screwed it up.

Algorithm 5 PPO Algorithm

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Update the policy by maximizing the PPO-Clip objective:
        $$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right), \quad (4.1)$$
    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:
        $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2, \quad (4.2)$$
    typically via some gradient descent algorithm.
end for=0
```

PPO allows you to run multiple epochs of gradient ascent on your samples without causing destructively large policy updates because of the Clipped Surrogate Objective function. This allows you to squeeze more out of your data and reduce sample inefficiency. This is not possible for vanilla policy gradient methods.

Chapter 5

DDPG and Multi-Agent RL

5.1 Deep Deterministic Policy Gradient (DDPG)

Traditional policy gradient methods, such as REINFORCE and Actor-Critic, use stochastic policies, meaning they sample actions from a probability distribution. However, stochastic policies may introduce unnecessary exploration noise in continuous action spaces. *Deterministic Policy Gradient (DPG)* [16] provides an alternative approach by directly learning a deterministic policy.

Deterministic Policy Gradient Theorem
In DPG, instead of learning a probability distribution over actions, we parameterize a deterministic policy function $\mu_\theta(s)$ that maps states directly to actions $a = \mu_\theta(s)$. The goal is to find the optimal policy that maximizes the expected return:

$$J(\theta) = \mathbb{E}_{s\sim\rho^\mu}\left[R(s)\right]$$

where ρ^μ represents the discounted state distribution under policy μ_θ. The deterministic policy gradient theorem states that the gradient of the expected return can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\mu}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)}\right]$$

This means that instead of sampling actions from a probability distribution, we directly optimize the policy using the action-value function $Q^\mu(s,a)$. The key benefit of this approach is that it eliminates the variance associated with stochastic policy gradients, making it more sample-efficient in continuous action spaces. [18]

Since DPG is deterministic, it lacks the inherent exploration mechanism of stochastic policies. To address this, an exploration noise process $\mathcal{N}_t$ is added to the deterministic policy. A common choice for $\mathcal{N}_t$ is the *Ornstein-Uhlenbeck (OU) noise* [12] process, which generates temporally correlated noise suitable for smooth control tasks.

Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [8] extends DPG by leveraging *deep neural networks* for function approximation, making it suitable for high-dimensional continuous control problems.

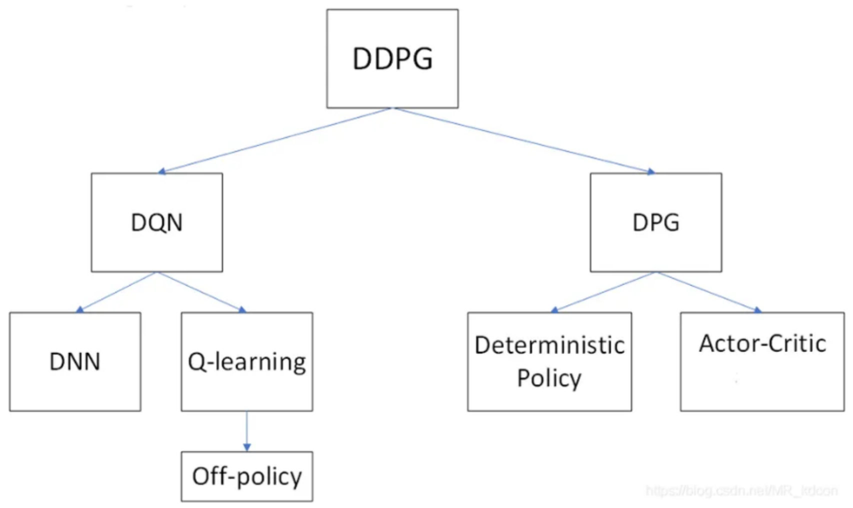


FIGURE 5.1: DDPG Flowchart

Actor-Critic Framework in DDPG
DDPG follows the Actor-Critic architecture, where:

- The **actor** network parameterized by θ^{μ} learns the deterministic policy $\mu_{\theta}(s)$. It outputs continuous action values. The actor is updated using the deterministic policy gradient:

$$\nabla_{\theta^{\mu}} J \approx \mathbb{E}_{s \sim \rho^{\mu}} \left[\nabla_{\theta^{\mu}} \mu_{\theta}(s) \nabla_{a} Q(s,a) \big|_{a=\mu_{\theta}(s)} \right]$$

- The **critic** network, parameterized by θ^{Q}, learns the action-value function $Q(s,a)$ using the Bellman equation:

$$Q(s,a) = \mathbb{E}\left[r + \gamma Q(s', \mu_{\theta}(s')) \right]$$

  The critic is updated by minimizing the Mean Squared Bellman Error (MSBE):

$$L(\theta^{Q}) = \mathbb{E}\left[\left(y - Q_{\theta^{Q}}(s,a) \right)^{2} \right]$$

  where $y = r + \gamma Q_{\theta^{Q}}(s', \mu_{\theta}(s'))$ is the target value.

Key Techniques in DDPG

DDPG incorporates two key components from Deep Q-Networks (DQN): Experience Replay and Target Networks.

- **Experience Replay:** To decorrelate consecutive samples and stabilize training, DDPG stores transitions (s, a, r, s') in a replay buffer and samples mini-batches for training.
- **Target Networks:** Separate target networks $Q_{\theta Q'}$ and $\mu_{\theta^{\mu'}}$ are used to compute target values, reducing instability due to rapid policy updates. They are updated slowly using:

  $$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

  where $\tau < 1$ controls the update rate.
- **Action Noise for Exploration:** Since DDPG uses a deterministic policy, noise is added to actions during training to encourage exploration.

Algorithm 6 Deep Deterministic Policy Gradient

```
Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute $a$ in the environment
  Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
  Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
  If $s'$ is terminal, reset environment state.
  if it's time to update then
    for however many updates do
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets $y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
      Update Q-function by one step of gradient descent using
      $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$
      Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
      Update target networks with $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$ and
      $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$
    end for
  end if
until convergence =0
```

DDPG has been widely applied in robotic control, autonomous driving, and simulated physics environments due to its ability to handle high-dimensional continuous action spaces. It is more sample-efficient than stochastic policy gradient methods. In addition, it uses off-policy learning, allowing data reuse from the replay buffer. However, it also suffers from instability and sensitivity to hyperparameters, requiring careful tuning for effective performance.

5.2 Multi-Agent Reinforcement Learning (MARL)

Real-world environments are inherently multi-agent in nature. Whether it is a team of drones coordinating to survey an area, a group of robots collaborating to assemble parts, or multiple self-driving cars navigating traffic, these systems require agents to learn and act in the presence of other agents. This has led to the emergence of **Multi-Agent Reinforcement Learning (MARL)**, a subfield of RL where multiple agents learn simultaneously in a shared environment.

The standard RL framework is based on a *Markov Decision Process (MDP)*, which assumes a single agent controlling all actions. However, in multi-agent systems, rewards and transitions are influenced by the joint actions of multiple agents. Therefore, MDPs are insufficient to model such systems. A more general framework is the **Stochastic Game**, also known as a *Markov Game*, which generalizes both MDPs and game-theoretic formulations [19].

Game Taxonomy

Game is a more general term to describe a problem that involves multiple agents or players.

- *Normal form* games model non-sequential games where agents take actions simultaneously, but the reward or payoff that they receive is dependent on the moves of the other agents. There is no evolving environment state however.
- An *Extensive form* game is a sequential game. There are multiple players who can take moves in the game, but not simultaneously.
- A *Stochastic game* is a generalisation of above two types of games. Here the moves are made simultaneously and the environment state evolves as a result of actions of agents. A stochastic game is also a generalization of a MDP in the sense that putting number of agents as 1 reduces it to a MDP.

A stochastic game is defined as a tuple:

$$G = (S, A_1, \ldots, A_n, r_1, \ldots, r_n, P, \gamma, \mu)$$

where:

- S : Set of environment states
- A_j : Action space for agent j
- $r_j : S \times A_1 \times \cdots \times A_n \times S \to \mathbb{R}$ Reward function for agent j
- $P : S \times A_1 \times \cdots \times A_n \to \Omega(S)$ Transition function that defines the probability of arriving in a state given a starting state and the actions chosen by all players
- γ : Discount factor
- $\mu : S \to [0, 1]$ Initial state distribution

The *Markov property* holds: the probability of transitioning to the next state and the rewards depend only on the current state and the joint action $\mathbf{a} = (a_1, \ldots, a_n)$. Each agent j follows its own policy π_j, and the joint policy is denoted as:

$$\pi = [\pi_1, \ldots, \pi_n]$$

Each agent aims to maximize its own expected cumulative return:

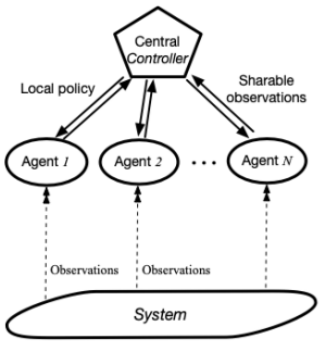
$$V^{\pi_j}(s) = \mathbb{E}_{\pi_j}\left[\sum_{t=0}^{\infty} \gamma^t r_j(s_t, \mathbf{a}_t, s_{t+1}) \,\middle|\, s_0 = s, \mathbf{a}_t = \pi(s_t)\right]$$

Types of Reward Structures

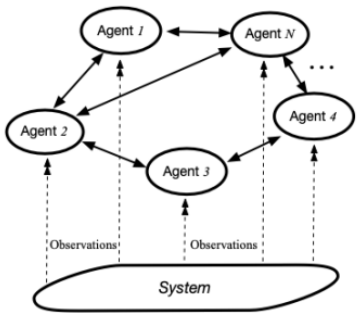
- **Zero-Sum Games:** The agents' rewards always sum up to zero.
- **Common-Reward Games:** All agents receive the same reward.
- **General-Sum Games:** No restrictions on individual reward functions.

Approaches to Learning in MARL

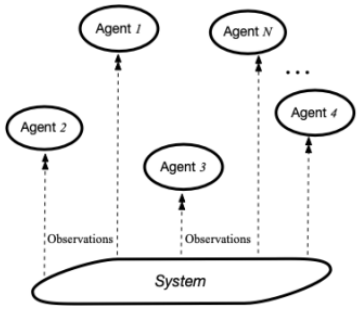
1. **Decentralized Learning:** Each agent learns its own policy independently, assuming the other agents are part of the environment[11].
   Advantages: One major advantage of decentralized learning is its simplicity in implementation. Since each agent acts independently, there is no need for inter-agent communication or coordination. This also reduces the input space each agent must handle, making the training process computationally more efficient.
   Challenges: However, decentralized learning introduces several challenges. The environment becomes non-stationary from each agent's perspective due to the constantly changing policies of other agents. This violates the assumptions of many traditional RL algorithms and can lead to unstable training. Furthermore, the agents may fail to generalize across symmetric states—cases where swapping agents should yield the same optimal action—since such symmetries are not inherently recognized.



(a) Centralized setting



(b) Decentralized setting with networked agents



(c) Fully decentralized setting

FIGURE 5.2: Approaches to MARL

2. **Centralized Learning:** A centralized controller gathers data from all agents and trains a joint or shared policy. Especially useful in homogeneous agent settings[19].

3. **Centralized Training with Decentralized Execution (CTDE):** **CTDE** enables stability during training while maintaining the scalability of decentralized execution[11]. Agents share full-state and joint-action information during training but act independently during execution. Example: Multi-Agent Actor-Critic

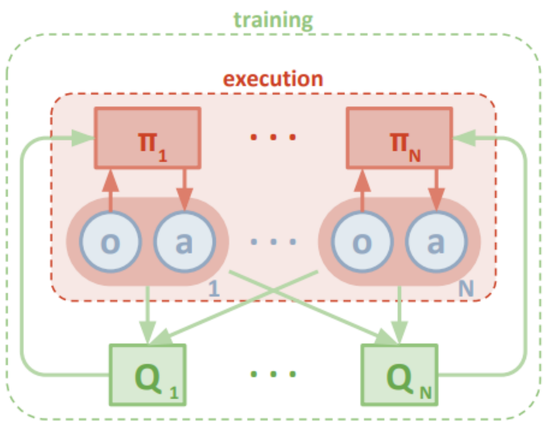


FIGURE 5.3: Overview of CTDE paradigm

An actor-critic algorithm under CTDE may use:

- **Actor:** $\pi_j(a_j|o_j)$ trains using local policy based on local observation (*Decentralized Execution*)
- **Critic:** $Q_j(o_1,\ldots,o_n,a_1,\ldots,a_n)$ evaluates the combined actions performed by all the agents globally using full action-state input (*Centralized Training*)

This setup helps mitigate non-stationarity during training and allows accurate credit assignment.

Summary
Multi-Agent Reinforcement Learning provides a powerful framework for modeling intelligent agents interacting in shared environments. With advancements like CTDE and centralized critics, MARL addresses the challenges of non-stationarity, coordination, and credit assignment. These developments are central to building robust multi-agent systems in domains such as swarm robotics, traffic control, and distributed sensing.

5.3 Multi-Agent DDPG (MADDPG)

In multi-agent environments, agents must learn to act optimally not only in response to the environment but also to the changing behaviors of other agents. While traditional deterministic method of *DDPG* [8] perform well in single-agent continuous control tasks, it struggles in multi-agent settings due to *non-stationarity* and *poor coordination*.

To address these issues, **Multi-Agent Deep Deterministic Policy Gradient (MADDPG)** was proposed by Lowe et al.[11]. It extends DDPG into a multi-agent framework using the *Centralized Training with Decentralized Execution (CTDE)* paradigm. Each agent maintains a decentralized actor (policy) that selects actions using only local observations. At the same time, the corresponding critic (Q network) is centralized and conditioned on the joint observations and actions of all agents during training.

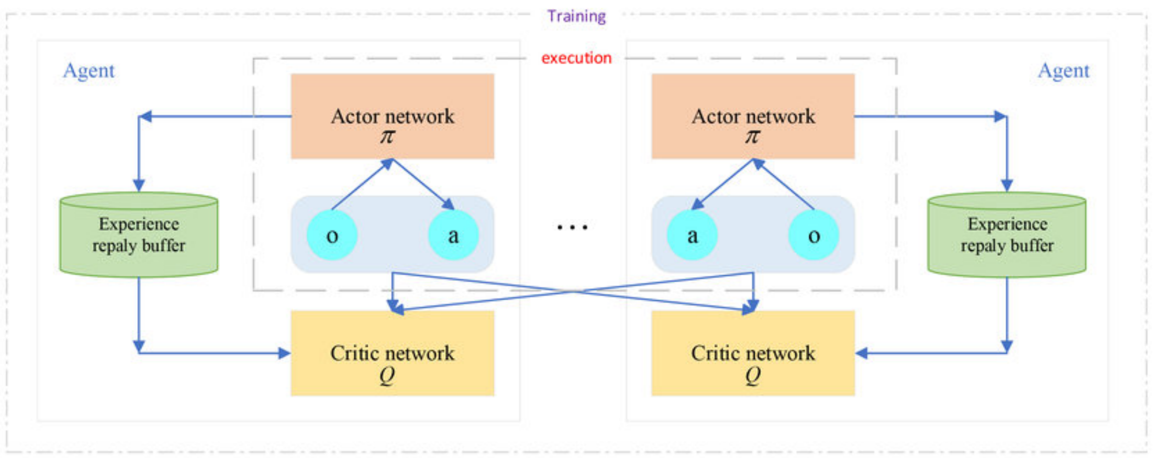


FIGURE 5.4: MADDPG Framework in CTDE paradigm

Consider an environment with N agents, each with its own policy $\pi_i(a_i|o_i)$, where o_i is the local observation for agent i and a_i is the action taken. The environment is modeled as a stochastic game with joint observations $\mathbf{o} = (o_1, \ldots, o_N)$ and joint actions $\mathbf{a} = (a_1, \ldots, a_N)$.

Each agent i has its own reward function $r_i(s, \mathbf{a})$, and aims to maximize its expected return:

$$J_i(\theta_i) = \mathbb{E}_{s,\mathbf{a}\sim\mathcal{D}}\left[\sum_{t=0}^{\infty} \gamma^t r_i(s_t, \mathbf{a}_t)\right]$$

The policy π_i is parameterized by θ_i, and the objective is to learn parameters that maximize J_i under the assumption that the other agents' policies are changing during training.

Actor-Critic Framework

To mitigate the non-stationarity caused by the evolving policies of other agents, MADDPG introduces a centralized critic Q_i for each agent i, which is conditioned on the global state and the joint action of all agents:

$$Q_i^{\pi}(s, a_1, \ldots, a_N) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_i(s_t, \mathbf{a}_t)\right]$$

The actor for agent i is updated via the deterministic policy gradient:

$$\nabla_{\theta_i} J_i = \mathbb{E}_{s,\mathbf{a}\sim\mathcal{D}}\left[\nabla_{\theta_i}\pi_i(a_i|o_i)\nabla_{a_i}Q_i(s, a_1, \ldots, a_N)\Big|_{a_i=\pi_i(o_i)}\right]$$

The critic is trained by minimizing the mean-squared Bellman error:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s,\mathbf{a},r,s'}\left[(Q_i(s,\mathbf{a}) - y)^2\right]$$

$$y = r_i + \gamma Q_i'(s', a_1', \ldots, a_N') \quad \text{with} \quad a_j' = \pi_j'(o_j')$$

Here, Q_i' and π_j' denote the target networks for the critic and policy, respectively, which are slowly updated to improve training stability.

Training and Execution

Training follows a typical off-policy learning setup using experience replay. Each agent stores its transitions (o_i, a_i, r_i, o_i') in a shared replay buffer. During training, mini-batches of joint experience are sampled, and both actors and critics are updated accordingly.
Despite the centralized critic during training, during execution each agent only uses its own actor $\pi_i(a_i|o_i)$, making the framework suitable for decentralized applications such as multi-robot control and autonomous UAV coordination.

Algorithm 7 Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to $M$ do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial state $\mathbf{x}$
    for $t = 1$ to max-episode-length do
        for each agent $i$, select $a_i = \mu_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. current policy and exploration
        Execute actions $\mathbf{a} = (a_1, \ldots, a_N)$ and observe reward $r$ and new state $\mathbf{x}'$
        Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
        $\mathbf{x} \leftarrow \mathbf{x}'$
        for agent $i = 1$ to $N$ do
            Sample a random minibatch of $S$ samples $(\mathbf{x}^j, \mathbf{a}^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
            Set $y^j = r_i^j + \gamma Q_i^{\mu'}(\mathbf{x}'^j, a_1', \ldots, a_N')|_{a_k'=\mu_k'(o_k^j)}$
            Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
            Update actor using the sampled policy gradient:
            $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^{\mu}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)|_{a_i=\mu_i(o_i^j)}$
        end for
        Update target network parameters for each agent $i$: $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
    end for
end for=0
```

Benefits of MADDPG

MADDPG offers several advantages over independently trained DDPG agents:

- It accounts for the joint action space during training, which improves stability and coordination.
- It adapts more effectively to non-stationary behaviors of other agents by conditioning the critic on full state-action information.
- It enables the use of function approximation in multi-agent settings where traditional tabular methods fail.

These qualities make MADDPG a strong baseline for continuous-action multi-agent reinforcement learning problems, particularly in partially observable real-world environments.

Chapter 6

MADDPG Application

6.1 Introduction

Unmanned Aerial Vehicles (UAVs) have emerged as a promising technology for providing flexible and on-demand communication coverage in areas lacking ground infrastructure. Their rapid deployment, high mobility, and elevated line-of-sight communication capabilities make them ideal for scenarios such as disaster response, large-scale events, and rural area connectivity.

Conventional wireless communication infrastructures are often static and ill-suited for dynamically changing environments. In contrast, UAVs mounted with communication equipment can act as mobile base stations (BSs), flying over regions of interest to provide network access to ground users. Notably, UAVs have been successfully proposed for public safety communication during natural disasters, such as earthquakes or floods, where terrestrial base stations may be damaged or overloaded.

However, deploying multiple UAVs introduces new challenges. These include maximizing long-term area coverage, ensuring fairness among service points or Points of Interest (PoIs), conserving limited battery energy, and maintaining inter-UAV connectivity without relying on centralized control. In practical applications, each UAV must make navigation decisions in real time while being aware of its limited sensing range and residual energy. Designing such a system demands autonomous coordination strategies that are robust, decentralized, and energy-efficient.

To address this, recent research has leveraged *Deep Reinforcement Learning (DRL)* to enable UAVs to learn control policies from environmental interactions. While earlier methods, such as DRL-EC3 [9], utilize centralized models to control all UAVs jointly, they often struggle with scalability and coordination. In contrast, this project adopts a distributed approach where each UAV is modeled as an independent learning agent, following Multi-Agent Reinforcement Learning (MARL) principles.

In particular, our work builds on the framework proposed in [10], where UAVs are controlled using a *Multi-Agent Deep Deterministic Policy Gradient (MADDPG)* network. We extend this by incorporating *Convolutional Neural Networks (CNNs)* for spatial encoding, and *Conditional Value at Risk (CVaR)* as a risk-averse training criterion along with the MADDPG network. This combination aims to improve long-term communication coverage by enabling cooperation among UAVs while accounting for fairness and energy efficiency in highly dynamic environments. Our work can be found on GitHub.

6.2 Problem Statement

We consider the problem of navigating a team of UAVs over a fixed two-dimensional area divided into K grid cells, each containing a Point of Interest (PoI), as shown in 6.1. The UAVs act as flying base stations (BS) to provide wireless communication coverage for ground users, aiming to ensure energy-efficient and geographically fair long-term coverage.

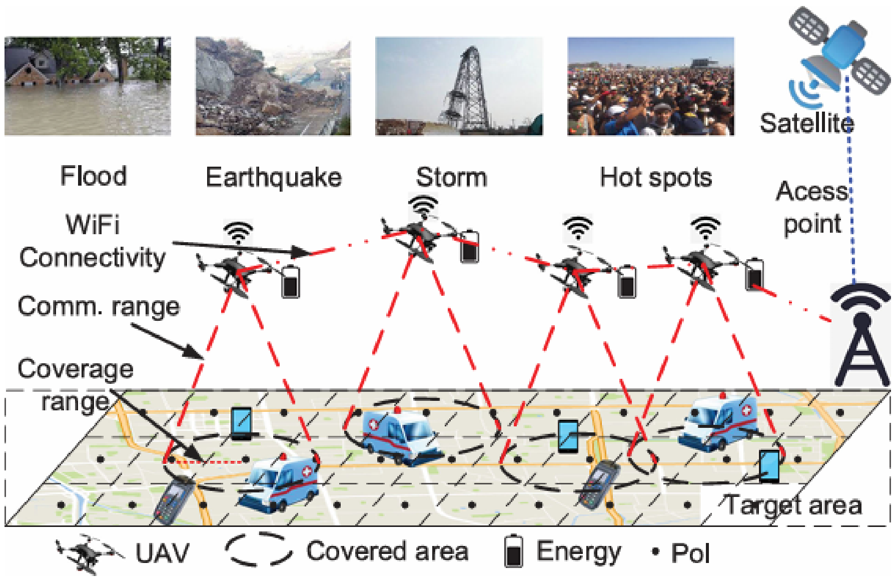


FIGURE 6.1: Multi-UAV navigation coverage scenario [10]

Let $N = \{1, 2, \ldots, N\}$ be the set of UAVs, each flying at a *fixed altitude*. UAVs are equipped with a communication range R and a coverage range R_0, where $R_0 \leq R$. During the mission of T timeslots (each of fixed duration), UAVs must decide their trajectory in terms of direction $\theta_i^t \in [0, 2\pi)$ and distance $l_i^t \in (0, l_{\max})$ or choose to hover at their current position ($\theta_i^t = 0, l_i^t = 0$). Their energy consumption e_i^t is determined either by movement (proportional to l_i^t) or hovering (a constant value).

We define the following metrics to evaluate UAV performance during the task:

1. Average Coverage Score

A PoI k is considered *covered* at timeslot t if it falls within the coverage area of any UAV. Let $w_t(k)$ be the number of timeslots up to time t that PoI k has been covered. The coverage score of PoI k is then:

$$c_t(k) = \frac{w_t(k)}{t}, \quad \forall k \in \{1, 2, \ldots, K\}$$

The average coverage score over all PoIs at timeslot t is:

$$c_t = \frac{1}{K} \sum_{k=1}^{K} c_t(k)$$

And at the end of the mission ($t = T$), the final average coverage score is c_T.

2. Geographical Fairness

Even with a high c_T, certain PoIs may never be covered, leading to unfairness. To capture the uniformity of coverage, *Jain's fairness index* is used:

$$f_t = \frac{\left(\sum_{k=1}^{K} c_t(k)\right)^2}{K \sum_{k=1}^{K} c_t(k)^2}$$

The fairness score at the end of the mission is denoted as f_T.

3. Energy Consumption

Each UAV selects a direction $\theta_i^t \in [0, 2\pi)$ and a normalized distance ratio $\rho_i^t \in [0, 1]$ in each timestep. The actual distance traveled is:

$$l_i^t = \rho_i^t \cdot l_{\max}$$

The energy consumed for UAV i at time t is calculated as:

$$e_i^t = \min\left(\eta \cdot l_i^t + E_h \cdot (1 - \rho_i^t),\ E_i^t\right)$$

where:

- η is the per-unit energy cost for movement,
- E_h is the energy cost for hovering (when $\rho_i^t = 0$),
- E_i^t is the remaining energy of UAV i at time t.

The UAV's energy is updated as:

$$E_i^{t+1} = E_i^t - e_i^t$$

And the total cumulative energy consumed by all UAVs over T timesteps is:

$$e_T = \sum_{t=1}^{T} \sum_{i=1}^{N} e_i^t$$

Objective

The joint objective of this multi-agent UAV system is to:

- Maximize c_T: the average spatiotemporal coverage score.
- Maximize f_T: the geographical fairness across PoIs.
- Minimize e_T: the cumulative energy cost over the mission.

This yields a complex multi-objective optimization problem under decentralized control. Each UAV must act based on limited observations while maintaining coverage and connectivity, under energy constraints. Our project addresses this challenge using a *Multi-Agent Deep Deterministic Policy Gradient (MADDPG)* framework enhanced with convolutional spatial encoding and risk-sensitive learning using CVaR.

6.3 Environment Design

To evaluate the performance of our learning framework in a realistic yet controlled setting, we designed a custom 2D grid-based simulation environment inspired by the problem formulation in [10]. The environment models the cooperative behaviour of a team of UAVs aiming to provide long-term, energy-efficient communication coverage to a set of Points of Interest (PoIs) spread across the area.

Environment Layout and Initialization

The environment is a 16×16 logical grid, mapped to an 80×80 visual grid with boundaries reinforced by virtual walls to restrict UAV movement. A fixed number of K data points (PoIs) are randomly initialized across the map and represent the ground users requiring coverage. Each PoI is visualized and tracked to monitor its spatiotemporal coverage status.

A team of N UAVs is initialized at random locations with full energy capacity $E_{\max}$. Each UAV operates independently with partial observability and can move within a maximum radius $l_{\max}$ per timestep or choose to hover in place. Walls and boundaries are visually rendered, and the system stores both state and heatmap images during training for visual analysis.

Observation and Action Space

Each agent i receives a local observation o_i represented as a three-channel 80×80 image:

- **Channel 0:** Global data map (PoI locations)
- **Channel 1:** Self UAV's position map (with energy encoding)
- **Channel 2:** Visit history over time (updated every step)

This observation structure is suitable for convolutional processing and enables each agent to make informed decisions based on spatial context.

Each agent outputs an action $a_i = [\theta_i, \rho_i] \in [0,1]^2$, where θ_i is mapped to a direction in $[0, 2\pi)$ and ρ_i represents a distance ratio in $[0,1]$, scaled by $l_{\max}$ to yield the true movement length.

Communication and Penalties

UAVs must maintain communication connectivity with at least one other UAV within a communication range R. If disconnected, a communication penalty p_c is applied. Similarly, attempting to cross environmental boundaries results in a wall penalty p_w, which is relatively higher in magnitude. Therefore, total penalty for an agent i will be $p_t^i = \sum_t (p_c + p_w)$.

Reward Function

It consists of two parts. One is penalty p_t^i, if UAV i flies out of the target area or loses connectivity to all of rest of UAVs within communication range R; another is time-varying energy efficiency, defined as

$$\Delta\eta_t = \frac{f_t \cdot \sum_{k=1}^{K} \Delta c_t(k)}{\sum_{j=1}^{N} \Delta e_t^i + \epsilon}$$

where $\Delta c_t(k) = c_t(k) - c_{t-1}(k)$ is the increase in average coverage and $\Delta e_t^i = e_t^i - e_{t-1}^i$ is the energy consumption difference from $t-1$ to t.
Therefore, net reward for agent i will be

$$r_t^i = \underbrace{\Delta\eta_t}_{\text{Coverage-Fairness-Energy}} + \underbrace{\lambda \cdot \mathcal{H}_t}_{\text{Exploration via Entropy}} - \underbrace{p_t^i}_{\text{Penalty of agent } i}$$

and $r_t = \{r_t^i | i \in N\}$

Where:

- $\mathcal{H}_t$ is the entropy of the UAV spatial distribution over the map,
- λ is the entropy factor promoting exploration.

- p_t^i is the total penalty accumulated by each agent

All rewards are normalized and clipped to ensure numerical stability during training. Each UAV shares a common reward with slight adjustments for individual constraints (e.g., wall hits, communication failure).

Terminal Condition

An episode terminates when all UAVs are depleted of energy, i.e., $E_i^t \leq \epsilon \cdot E_{\max}$ for all i.

6.4 Proposed Solution

MADDPG Architecture

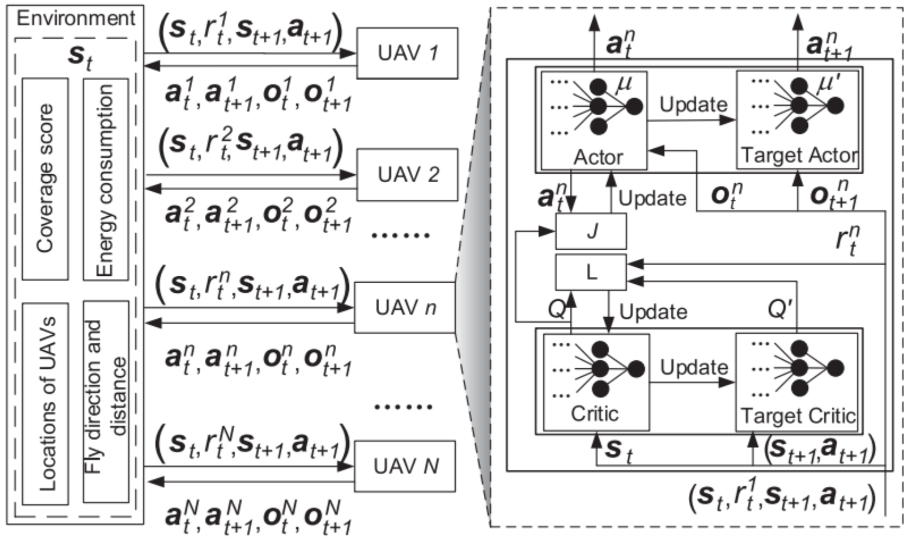


FIGURE 6.2: Architecture of MADDPG Network [10]

Our architecture comprises of a multi-agent model-free actor-critic model which is visually described in 6.2. The training algorithm has been described in detail below 8. Note that the Q^i used for Critic Network comes from the Quantile Critic Network, which is risk-adjusted Critic Network (discussed in later sections) based on CVaR logic.

Algorithm 8 Distributed Energy-Efficient Multi-UAV Navigation (Training Process)

```
for UAV $i := 1, \ldots, N$ do
    Randomly initialize critic network $Q^i(s_t, a_t \mid \theta^{Q^i})$ and actor network $\mu^i(o_t^i \mid \theta^{\mu^i})$ with weights $\theta^{Q^i}$ and $\theta^{\mu^i}$
    Initialize target networks $Q'^i(\cdot)$ and $\mu'^i(\cdot)$ with weights $\theta^{Q'^i} = \theta^{Q^i}$ and $\theta^{\mu'^i} = \theta^{\mu^i}$
end for
for episode $:= 1, \ldots, M$ do
    Initialize environment and receive an initial state
    for $t := 1, \ldots, T$ do
        For each UAV $i$, select $a_t^i = \mu^i(o_t^i) + \epsilon$ where $\epsilon$ is decaying Gaussian noise
        Execute actions $a_t = (a_t^1, \ldots, a_t^N)$, get new state $s_{t+1}$, performance metrics and reward $r_t = (r_t^1, \ldots, r_t^N)$ as per defined reward structure earlier.
        Store $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{B}$, and $s_t \leftarrow s_{t+1}$
        for UAV $i := 1, \ldots, N$ do
            Sample $H(<< B)$ random transitions $(s_j, a_j, r_j, s_{j+1})$ from $\mathcal{B}$
            Set target value $y_j^i = r_j^i + \gamma Q'^i(s_{j+1}, a_{j+1} \mid \theta^{Q'^i})|_{a_{j+1}^i = \mu'^i(o_{j+1}^i | \theta^{\mu'^i})}$
            Update critic network weights $\theta^{Q^i}$ by minimizing loss:
            $L(\theta^{Q^i}) = \frac{1}{H} \sum_{j=1}^{H} (y_j^i - Q^i(s_j, a_j \mid \theta^{Q^i}))^2$
            Update actor network weights $\theta^{\mu^i}$ using policy gradient:
            $\nabla_{\theta^{\mu^i}} J(\theta^{\mu^i}) \approx \frac{1}{H} \sum_{j=1}^{H} \nabla_{\theta^{\mu^i}} \mu^i(o_j^i \mid \theta^{\mu^i}) \nabla_{a_j^i} Q^i(s_j, a_j)|_{a_j^i = \mu^i(o_j^i | \theta^{\mu^i})}$
            Update target networks: $\theta^{Q'^i} = \tau\theta^{Q^i} + (1-\tau)\theta^{Q'^i}$, and $\theta^{\mu'^i} = \tau\theta^{\mu^i} + (1-\tau)\theta^{\mu'^i}$
        end for
    end for
end for=0
```

Following algorithm 9 describes testing process:

Algorithm 9 Distributed Energy-Efficient Multi-UAV Navigation (Testing Process)

```
for episode $:= 1, \ldots, M$ do
    Initialize environment and receive an initial state
    for $t := 1, \ldots, T$ do
        For each UAV $i$, select $a_t^i = \mu^i(o_t^i)$
        Execute actions $a_t = (a_t^1, \ldots, a_t^N)$, get new state $s_{t+1}$, performance metrics and reward $r_t = (r_t^1, \ldots, r_t^N)$ as per defined reward structure earlier.
        $s_t \leftarrow s_{t+1}$
    end for
end for=0
```

CNN based Spatial Encoding

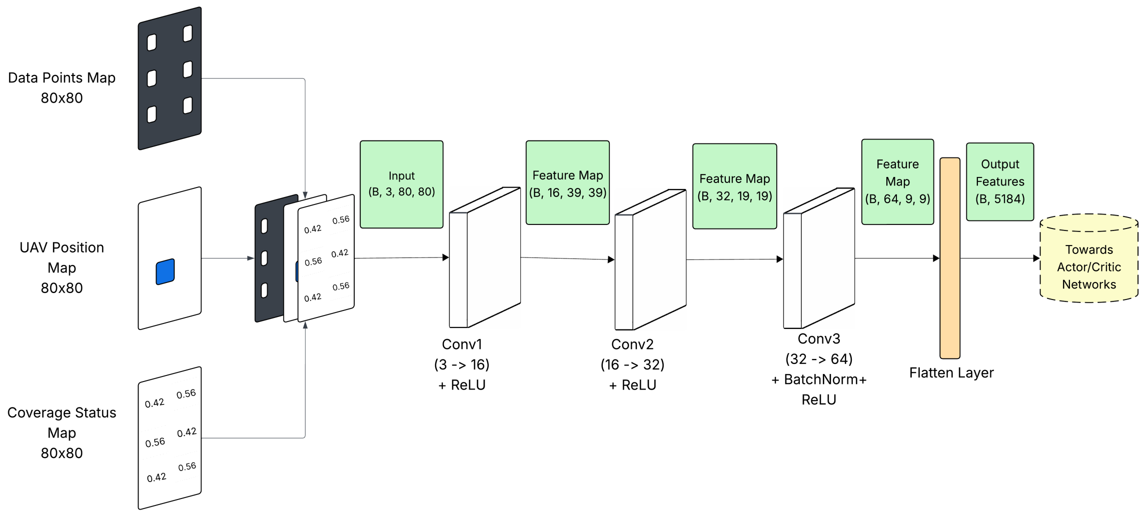


FIGURE 6.3: CNN Architecture

A *Convolutional Neural Network (CNN)* is employed to extract features from observations for each UAV agent. The CNN processes raw grid-based observations of dimensions $80 \times 80 \times 3$ as described earlier and extracts high-level spatial features.

The implemented CNN consists of three convolutional layers, each followed by a ReLU activation function, and a batch normalization layer after the final convolution. The architecture is designed for input images of size 80×80 with three channels:

- **Conv Layer 1:** 16 filters, 3×3 kernel, stride 2
- **Conv Layer 2:** 32 filters, 3×3 kernel, stride 2
- **Conv Layer 3:** 64 filters, 3×3 kernel, stride 2
- **Batch Normalization:** Applied to the output of the third convolution
- **Flatten:** The final feature map is flattened to a 1D vector

Each agent's observation is passed through its own CNN, which encodes the spatial information into a compact feature vector. These feature vectors are then used as inputs to the actor and critic networks within the MADDPG framework.

By leveraging CNNs for spatial encoding, the agents can effectively interpret complex environmental layouts and make informed decisions based on the encoded spatial context. This approach enables the agents to generalize across different spatial scenarios and improves their ability to coordinate in coverage and exploration tasks.

Incorporating Risk-Averse RL - CVaR

In many real-world scenarios—such as UAV (Unmanned Aerial Vehicle) navigation—it is not sufficient to optimize the average (expected) return alone. Rare events or *tail* outcomes (e.g., near collisions or other catastrophic episodes) can lead to unacceptable risks. To mitigate these risks, our framework incorporates a risk-sensitive criterion using Conditional Value at Risk (CVaR). CVaR focuses on the expected loss in the worst-case tail of the return distribution, ensuring that our agents learn policies that are robust even in adverse conditions.
CVaR is defined as the expected return (or loss) under the worst $\alpha-$percentile outcomes. Formally, for a given confidence level α (e.g., 0.95), CVaR represents the mean of all returns that lie in the worst $(1\check{}\alpha)$ fraction of outcomes. By optimizing policies with respect to both the expected return and CVaR, the learning algorithm penalizes rare but severe deviations, which can be crucial for safety-critical applications. In effect, minimizing risk (i.e., reducing CVaR) results in agents that are more cautious in uncertain or highly dynamic settings.
Given a random return distribution Z, the *Value at Risk* at level $\alpha \in (0, 1]$ is defined as the α-quantile:

$$\text{VaR}_\alpha(Z) = \inf\{z \in \mathbb{R} \mid \mathbb{P}(Z \leq z) \geq \alpha\}$$

The corresponding *Conditional Value at Risk* is the expected return under the lower tail of the distribution:

$$\text{CVaR}_\alpha(Z) = \mathbb{E}[Z \mid Z \leq \text{VaR}_\alpha(Z)]$$

To introduce risk-sensitive behaviour into our multi-agent training, we extend the standard MADDPG framework with a *Quantile-Based Critic Network* that enables the use of *Conditional Value at Risk (CVaR)* as a training objective. The motivation behind using CVaR is to reduce the likelihood of agents converging to high-variance policies that may yield poor performance under rare but critical scenarios (e.g., high energy consumption, loss of connectivity, or poor coverage).Unlike traditional Q-learning which optimizes the mean return, CVaR focuses on minimizing risk by training the agent to optimize its performance in the worst $\alpha \cdot 100\%$ of cases.

Critic Network Architecture: We modify the critic network to output a set of quantile estimates, rather than a single scalar Q-value. The new critic, implemented as `QuantileCriticNetwork`, accepts both state and action as input and outputs n_q quantile values:

$$Q_\phi(s, a) = [q_1, q_2, \ldots, q_{n_q}]$$

Here, q_i denotes the i^{th} quantile estimate of the return distribution for the state-action pair (s, a), and n_q is the number of quantiles sampled (typically set to 50 in our experiments). The architecture of this critic consists of two hidden layers followed by an output layer with n_q units:

- **Input:** Concatenated state and action vectors
- **Hidden Layers:** Two fully connected layers with ReLU activations
- **Output Layer:** A linear layer with n_q units

CVaR-based Target Computation: During training, we sort the predicted quantiles in ascending order and compute the mean of the lowest $\alpha \cdot n_q$ values to approximate CVaR_α:

$$\hat{Q}^{\text{CVaR}}(s,a) = \frac{1}{\lceil \alpha \cdot n_q \rceil} \sum_{i=1}^{\lceil \alpha \cdot n_q \rceil} q_{(i)}$$

This $\hat{Q}^{\text{CVaR}}$ is then used in place of the standard Q-value in the Bellman target during critic training. The rest of the training pipeline, including actor updates and target networks, follows the standard MADDPG formulation.

Tuning the Risk Sensitivity: The hyperparameter $\alpha \in (0,1]$ controls the degree of risk aversion. A lower value of α results in a stricter policy that prioritizes safety and robustness but may slow down convergence. A higher α leads to more lenient, mean-like behaviour and faster convergence but potentially riskier actions.

Impact: By training agents to optimize CVaR rather than expected return, our framework encourages cautious and energy-aware navigation strategies. UAVs avoid high-risk trajectories and learn policies that generalize better under uncertainty and energy constraints. This enhancement improves system stability, reduces variance in episode returns, and promotes smoother exploration-exploitation tradeoffs.

6.5 Implementation Setup

The implementation models a cooperative multi-UAV coverage problem, where multiple UAVs must efficiently cover a set of data points in a grid environment. The environment simulates UAV movement, energy consumption, communication constraints, and coverage tracking.
The environment is implemented in `env.py`, which manages the UAVs, data points, energy constraints, communication, and reward structure.
The MADDPG algorithm is implemented in `maddpg/maddpg_uav.py`, with supporting modules for CNNs (`cnn.py`), agents (`agents.py`), and replay buffer (`buffer.py`).

Utilities for data point generation, logging, and plotting are provided in `utils/`, including `input.py`, `logger.py`, and `plot_logs.py`.
Training and testing workflows are managed by `train.py` and `test.py`, which handle initialization, logging, model saving/loading, and result visualization.

Input Module

The `input.py` module provides functionality to generate data points for the UAV coverage environment. It supports image-based input, where a grayscale image is processed and non-zero pixels are interpreted as data points to be covered by UAVs. This allows for flexible and realistic scenario creation by simply providing different images. Alternatively, initial coordinates of data points can also be fed directly.

Logging Module

The `logger.py` module handles experiment logging. It records key metrics such as episode rewards, coverage percentage, energy consumption, and other relevant statistics during training and testing. The logs are saved in a structured format (e.g., CSV or JSON), making it easy to analyze and visualize results later.

Plotting Module

The `plot_logs.py` module provides tools to visualize the logged experiment data. It reads the log files generated by `logger.py` and produces plots for metrics. These visualizations help in evaluating the performance and behaviour of the multi-UAV system across different runs.

Training Scenarios

During training, the system runs for a user-defined number of episodes, with each episode consisting of up to 300 steps. In each step, all UAV agents select actions, interact with the environment, and collect experiences. The MADDPG agents update their networks every few steps, and model checkpoints are saved periodically. Key metrics such as coverage, fairness, and energy efficiency are logged for analysis. Training can be resumed from saved models if needed.

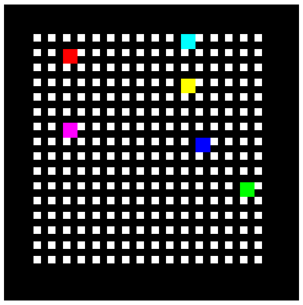
Testing Scenarios

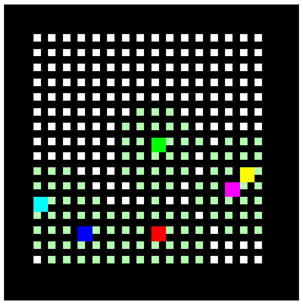
For testing, the trained models are loaded and evaluated over a specified number of episodes, each with up to 300 steps. Agents act according to their learned policies. The environment records performance metrics and saves state images and heatmaps at regular intervals for. Results are logged and visualized to assess the effectiveness of the trained policies.

6.6 Results and Analysis

State Visualisation

In addition to the training curves, we captured representative state images during the training episodes:


(A) Initial positions (random distribution) of all uavs before training starts


(B) State image at the end of a randomly selected training episode

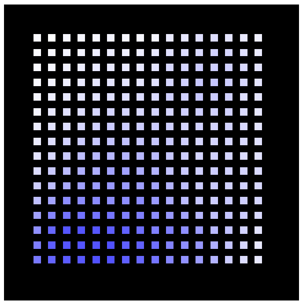

(C) Heatmap of a state randomly selected while training

FIGURE 6.4: State images while training

1. **Initial State Image:** Figure 6.4a shows the initial position of all uavs at the start of training. This position remains fixed before start of each episode while training but changes in case of testing.
2. **Random Episode State:** Figure 6.4b presents a snapshot from a randomly selected training episode. This image captures the positions of multiple UAVs and their corresponding sensing regions (shown by the *green* region). The spread and overlap visible in the image highlight how the agents coordinate to maximize coverage while managing redundant observations.
3. **Heatmap State Image:** One of the heatmaps (see Figure 6.4c) illustrates the spatial distribution of UAV coverage. Areas with higher intensity indicate regions that have been observed more frequently. This visualization confirms that the agents strategically focus on less-covered regions over time, thereby improving overall coverage.

Various Plots

To analyze the relationships among the different performance metrics (coverage, fairness, and energy efficiency), we include the following empirical plots obtained during training.

1. Fairness vs. Coverage

The plot 6.5 shows a almost linear and positively correlated trend. As UAVs increase their

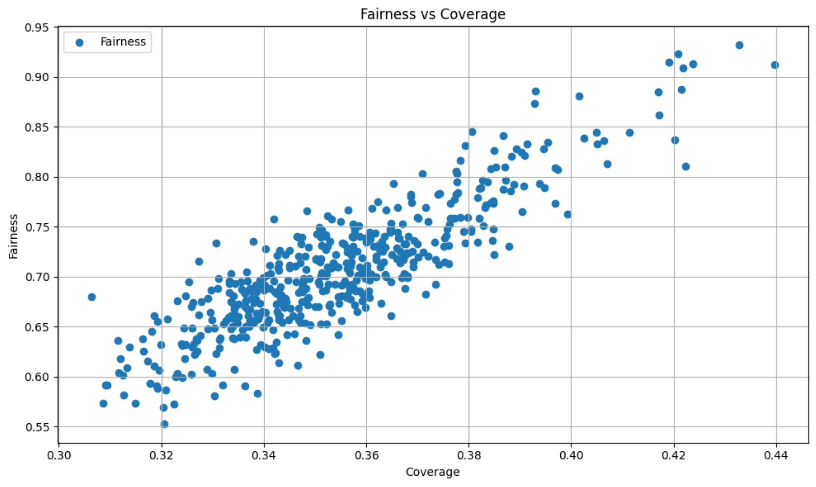


FIGURE 6.5: Total Coverage plotted against the Total Fairness measured at the end of each episode during training

coverage across more PoIs, fairness improves because more PoIs are uniformly serviced. A high coverage inherently encourages agents to distribute themselves more evenly, minimizing neglected zones.

2. Fairness vs. Energy Efficiency

A linearly increasing trend is observed in plot 6.6. This is due to the fact that when UAVs

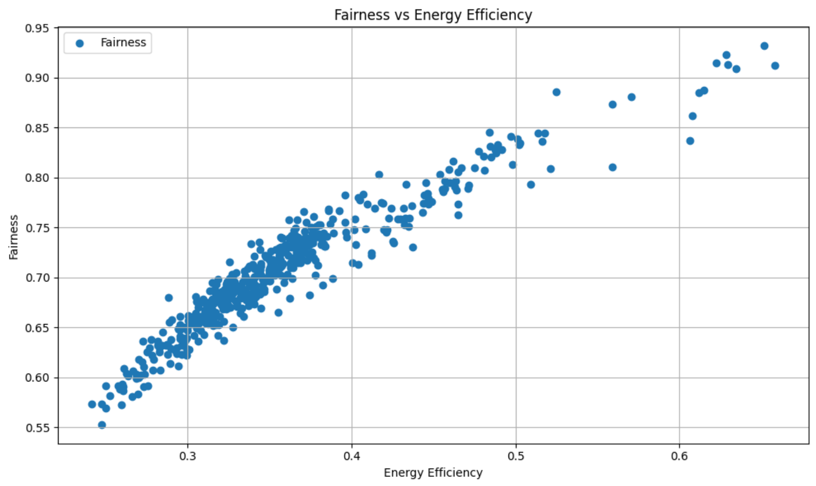


FIGURE 6.6: Total Fairness plotted against the Average energy Efficiency measured at the end of each episode during training

adopt energy-efficient strategies (e.g., reduced unnecessary movement, strategic hovering), they tend to stay longer in their zones of responsibility. This leads to better temporal coverage balance, hence enhancing fairness. Efficient agents avoid overlapping coverage or competition, promoting distributed fairness.

3. Fairness vs. Episode Number

The plot 6.7 shows a *hockey stick* curve—starting with high fairness in initial episodes (peaking around 0.95), followed by a flat tail averaging around 0.70. Initially, high exploration (due to high entropy) allows UAVs to randomly cover diverse PoIs, resulting in high fairness. As training progresses and the CVaR-based risk-averse policy becomes dominant, agents prioritize safer and more certain zones, reducing exploration. This strategic risk aversion leads to stabilizing behaviours but may overlook less-visited PoIs, causing fairness to plateau.

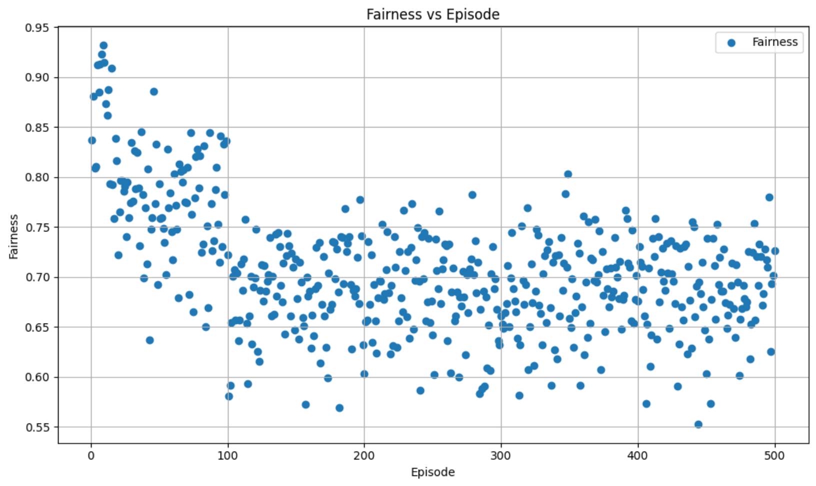


FIGURE 6.7: Total Fairness per episode during training

These plots demonstrate the interplay between exploration, risk sensitivity, and fairness, highlighting the nuanced tradeoffs that arise in decentralized UAV coordination.

Test Result Summary

After training, the model was evaluated on various unseen test scenarios with Risk-Averse (CVaR) Metric set to 0.5. The average test results are summarized below:

- Mean Coverage Percentage: 33.0369%
- Fairness Index: 33.1693%
- Energy Efficiency Metric: 13.5578%

These results indicate that the trained multi-UAV system not only achieves good coverage and coordinated behaviour but also successfully maintains robustness against risky maneuvers through the CVaR adjustment.

6.7 Conclusion

In this chapter, we presented an application of Multi-Agent Reinforcement Learning on energy-efficient and fair multi-UAV coverage using a decentralized, risk-aware deep reinforcement learning framework. The core of our approach is built upon the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm, enhanced through two significant additions: convolutional neural networks (CNNs) for spatial encoding and Conditional Value at Risk (CVaR) to introduce risk-sensitive policy learning.

We began by formalizing the problem of long-term communication coverage in a 2D space, where a fleet of UAVs must provide fair and sustained coverage to spatially distributed Points of Interest (PoIs). We then designed a custom simulation environment that models realistic UAV behaviour including energy consumption, communication constraints, and spatial decision-making based on local observations. The reward function was crafted to balance fairness, coverage improvement, and energy usage, while penalizing disconnecting or boundary-violating behaviour.

Our extension to the standard MADDPG included the use of CNNs to interpret spatial features from image-based observations and a quantile-based critic network that enabled CVaR computation. By tuning the α parameter, we could shift the learning paradigm between aggressive exploration and conservative, risk-averse behaviour. This significantly impacted the agents' ability to maintain fairness and energy efficiency over time.

Empirical analysis through various plotted metrics—such as fairness vs. coverage and fairness vs. energy efficiency—confirmed the positive impact of our approach. We also observed that the fairness initially peaked due to high exploration but stabilized to a more risk-sensitive level as CVaR-driven policies matured.

Overall, the project demonstrated the potential of combining MARL with risk-averse learning and spatial encoding to address real-world challenges in multi-UAV coordination. This framework can be extended to more complex and dynamic environments, including 3D navigation, moving PoIs, or real-time deployment on physical UAVs in the future.

Bibliography

[1] Q-learning: A value-based reinforcement learning algorithm. *Medium*, 2019. URL https://medium.com/intro-to-artificial-intelligence/q-learning-a-value-based-reinforcement-learning-algorithm-272706d835cf.

[2] Policy gradients in a nutshell. *Towards Data Science*, 2023. URL https://medium.com/towards-data-science/policy-gradients-in-a-nutshell-8b72f9743c5d.

[3] Reinforcement learning, 2023. URL https://en.wikipedia.org/wiki/Reinforcement_learning.

[4] Hugging Face. Deep reinforcement learning with advantage actor-critic (a2c). Hugging Face Blog, 2023. URL https://huggingface.co/blog/deep-rl-a2c.

[5] Hugging Face. Deep reinforcement learning with ppo. Hugging Face Blog, 2023. URL https://huggingface.co/blog/deep-rl-ppo.

[6] Hugging Face. Visualizing deep reinforcement learning policies. Hugging Face Deep RL Course, 2023. URL https://huggingface.co/learn/deep-rl-course/unit8/visualize.

[7] GeeksforGeeks. Actor-critic algorithm in reinforcement learning. GeeksforGeeks, 2023. URL https://www.geeksforgeeks.org/actor-critic-algorithm-in-reinforcement-learning/.

[8] Timothy P. Lillicrap et al. Continuous control with deep reinforcement learning. *arXiv preprint*, arXiv:1509.02971, 2015. URL https://arxiv.org/pdf/1509.02971.pdf.

[9] Chi Harold Liu, Zheyu Chen, Jian Tang, Jie Xu, and Chengzhe Piao. Energy-efficient uav control for effective and fair communication coverage: A deep reinforcement learning approach. *IEEE Journal on Selected Areas in Communications*, 36(9):2059–2070, 2018.

[10] Chi Harold Liu, Xiaoxin Ma, Xudong Gao, and Jian Tang. Distributed energy-efficient multi-uav navigation for long-term communication coverage by deep reinforcement learning. *IEEE Transactions on Mobile Computing*, 19(6):1274–1286, 2020. URL https://ieeexplore.ieee.org/document/8676325.

[11] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, volume 30, 2017. URL https://arxiv.org/abs/1706.02275.

[12] NESTML. Ornstein-uhlenbeck noise tutorial. 2024. URL https://nestml.readthedocs.io/en/latest/tutorials/ornstein_uhlenbeck_noise/nestml_ou_noise_tutorial.html.

[13] OpenAI. Vanilla policy gradient (vpg). Spinning Up Documentation, 2023. URL https://spinningup.openai.com/en/latest/algorithms/vpg.html.

[14] Qempsil0914. Deep q-learning part 2: Double deep q-network (double dqn). *Medium*, 2023. URL https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2.

[15] David Silver. Reinforcement learning teaching material, 2025. URL https://www.davidsilver.uk/teaching/. Accessed: 2025-02-02.

[16] David Silver et al. Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on Machine Learning*, 32(1):387–395, 2014. URL https://proceedings.mlr.press/v32/silver14.html.

[17] Daniel Takeshi. Going deeper into reinforcement learning: Fundamentals of policy gradients. Daniel Takeshi Blog, 2017. URL https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/.

[18] OpenAI Spinning Up. Deep deterministic policy gradient (ddpg) algorithm. 2023. URL https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

[19] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, 2021. URL https://arxiv.org/abs/1911.10635.