NORTHWESTERN UNIVERSITY

Deep Reinforcement Learning for Optimizing Multi-Agent Patrolling Systems in a Graph

A DISSERTATION

SUBMITTED TO THE SCHOOL OF MATHEMATICAL SCIENCES OF

UNIVERSITY COLLEGE CORK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Data Science and Analytics

By

Nicolas Martinet

EVANSTON, ILLINOIS
United States

August 2023

Contents

1 Introduction to Multi-Agent Patrolling **5**
- 1.1 Introduction to Intelligent Autonomous Systems . . . 5
- 1.2 Multi-Agent Patrolling Challenges . . . 6
- 1.3 Mathematical Foundation & Abstraction . . . 7
- 1.4 Simulations and Experimental Graph Structures . . . 8
- 1.5 Overview of the thesis . . . 8

2 Introduction to Reinforcement Learning **10**
- 2.1 Fundamental Concepts of Reinforcement Learning . . . 10
  - 2.1.1 Reinforcement Learning Paradigm . . . 10
  - 2.1.2 Components of MDP . . . 10
  - 2.1.3 Reward and Policy . . . 12
  - 2.1.4 Rewards, Returns, and Value Functions . . . 12
  - 2.1.5 The Bellman Equations . . . 15
  - 2.1.6 Optimality . . . 15
- 2.2 Basics of Q-Learning . . . 17
  - 2.2.1 Updating rule . . . 17
  - 2.2.2 Exploration, Exploitation Trade-Off . . . 18
  - 2.2.3 Q-Learning Algorithm . . . 19
  - 2.2.4 Limitation . . . 20
- 2.3 Deep Learning . . . 21
  - 2.3.1 Forward propagation . . . 21
  - 2.3.2 Training and Backward Propagation . . . 22
- 2.4 Deep Q-Learning (DQL) . . . 24
  - 2.4.1 Discrete State Space . . . 24
  - 2.4.2 DQL Algorithm . . . 24
- 2.5 Proximal Policy Optimization (PPO) . . . 28
  - 2.5.1 Use of the Advantage Function in PPO . . . 30
  - 2.5.2 Temporal Difference Learning . . . 31
  - 2.5.3 Generalized Advantage Estimation (GAE) . . . 31

3 Exploring Multi-Agent Reinforcement Learning **33**
- 3.1 Decentralized Multi Agent MDP Model . . . 33
  - 3.1.1 Multi-Agent Markov Decision Process (MMDP) . . . 33
  - 3.1.2 Partial Observability: The Dec-POMDP Model . . . 34

3.2 Asynchronous Independent Approach for Multi-Agent Planning 35
3.2.1 Individualized Actions with Shared Policy 36
3.2.2 Local Observations with Global Contribution 36
3.2.3 Benefits of the Asynchronous Independent Approach 36
3.3 Centralized Global Approach for Multi-Agent Planning 37
3.3.1 Overview 37
3.3.2 Constructing the Multi-Action Policy 38

4 Challenges that arise in a Multi-Agent Patrolling Problem 41
4.1 Reward Sparsity 41
4.1.1 Agent Flip-Flop 41
4.1.2 Random Walk Approach 41
4.2 Restrained Action Swap (RAS) 44
4.2.1 Mitigating Reward Sparsity with the Restrained Action Swap Scheme 44
4.2.2 Action Reward pair mismatch 44
4.3 Reward Strategies 46
4.3.1 Challenges with Direct Idle Time-based Rewards 46
4.3.2 Ranking Approach 47
4.3.3 Normalization Approach 48
4.3.4 Centered Idle Time Approach 48
4.3.5 Average Idle Time Approach 49
4.3.6 N-Worst Idle Time Approach 49
4.3.7 Final Average Idle Time Approach 49
4.4 Observation Strategies 49
4.4.1 Ranking as Observations 50
4.4.2 Normalization as Observations 50
4.4.3 Centered Idle Times as Observations 50
4.4.4 Utilizing a Matrix Representation for Agent-Node Affiliation 51

5 Numerical Results 52
5.1 Synchronous Global Approach with PPO 52
5.2 Asynchronous Individual Approach with DQL 54
5.3 Conclusion 55

Acknowledgements

I am profoundly grateful to Dr. Eric Wolsztynski of University College Cork for his valuable guidance in organizing my research and for his continuous advice and support throughout the summer.

My sincere appreciation goes to Prof. Qi Zhu of Northwestern University for granting me the opportunity to be a part of the IDEAS lab. I deeply value the trust and support he and the team provided during my summer tenure.

Special thanks are due to Anthony Goeckner of Northwestern University for our collaboration. The constructive and insightful discussions we shared over the summer have been greatly enriching.

I would also like to thank Xinliang Li and Yueyuan Sui of Northwestern University for their collaboration.

Notation

$V \to W$	Application from set V to set W
$G = (V, E)$	Graph with vertex set V and edge set E
$\|V\|$	Cardinality of V, here the number of nodes
$\|\| \cdot \|\|_2$	Euclidean norm (L2 norm)
$d(v_1, v_2)$	Distance between vertices v_1 and v_2
ζ	Idle time
τ	Trajectory
π	Policy
$\tau \sim \pi_\theta$	Trajectory τ generated by policy π_θ
$Pr(a_t = a \| s_t = s)$	Conditional probability
$\max_a(\cdot)$	Maximum over all actions a
$\mathbb{R}^{n \times p}$	Matrix of real numbers with n rows and p columns
σ	Activation function
$\frac{\partial f}{\partial x}$	Partial derivative of f over x.
∇	Gradient operator, $\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$
$E[\cdot]$	Expectation
$Var(\cdot)$	Variance, $Var(X) = E[(X - E[X])^2]$
$\mathcal{S}$	State space
$\mathcal{A}$	Action space
$\mathcal{R}$	Reward function
γ	Discount factor
θ	Parameters of the policy
a_t	Action at time t
s_t	State at time t
r_t	Reward at time t
o_t	Observation at time t

Chapter 1

Introduction to Multi-Agent Patrolling

1.1 Introduction to Intelligent Autonomous Systems

Intelligent autonomous systems are machines, devices, or software that can operate independently and perform tasks without the need for human intervention. They introduce multifaceted challenges in design and implementation, encapsulate the need for perceptive observation, adaptive navigation, and strategic planning. These systems must be outfitted with advanced sensory equipment, such as visual cameras or LIDAR technology, to observe and interpret their complex surroundings.

The navigation aspect requires the development of algorithms for dynamic steering control, optimal pathfinding, and obstacle avoidance, ensuring smooth and purposeful movement within diverse terrains. Implementing those systems opens the door to real-world applications ranging from urban mobility solutions to exploratory missions in uncharted landscapes.

Patrolling planning, the focal point of this dissertation, delves into the art of autonomous decision-making, task sequencing, and patrol coordination. It's not just about merely moving from one point to another; it involves designing agents that can learn, adapt, and decide the order of tasks in a dynamic environment filled with barriers such as walls. This also encompasses the development of distributed control systems where multiple agents can collaborate to patrol an environment, ensuring areas of interest are not left unattended.

Specific applications under consideration include all-terrain robots monitoring hiker safety in rugged landscapes, and unmanned aerial vehicles (UAVs) in industrial settings detecting malfunctions, leaks, or unauthorized intrusions. These scenarios necessitate an intricate balance of autonomous path planning and multi-agent coordination, optimizing the division of responsibilities, and ensuring a systematic and thorough examination of areas of concern.

The ambition of this research is in employing Deep Reinforcement Learning to forge a balanced integration of smart perception, allowing agents to work together in exploring an environment and making sure no areas of interest are overlooked.

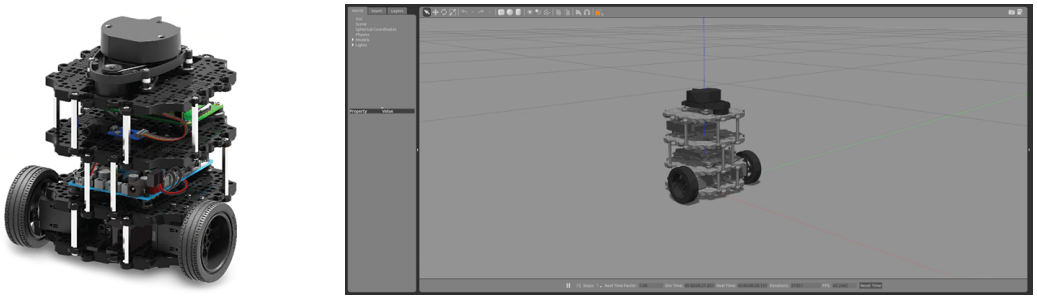


Figure 1.1: TurtleBot3 robot used in IDEAS Lab and its 3D model in ROS

1.2 Multi-Agent Patrolling Challenges

In the context of multi-agent patrolling, the challenges extend far beyond simple navigation from point a to point b. The primary concern here is to enable agents to intelligently decide their next destination, with an emphasis on task allocation and coordination among multiple agents. This problem introduces several layers of complexity, making it a unique and intriguing subject of study.

One significant aspect of this complexity lies in the communication between agents. Depending on the scenario, communication can be either partial or complete, affecting the way agents share information and make decisions. Agents might have different characteristics, such as speed, that add to the difficulty of maintaining coherence in their collaborative efforts.

Another major facet of this problem is the distinction between centralized and decentralized control. In a centralized system, a single entity (such as a control center) may synchronize all agents, leading them to act in unison. This method can offer more straightforward coordination but may be less flexible in responding to dynamic changes in the environment. Conversely, in a decentralized system, agents take actions independently and asynchronously. While this can provide more adaptability, it also introduces substantial challenges.

In asynchronous, independent action scenarios, one agent's decision might lead to a modification of the environment that impacts the decision-making process of others. Such unpredictable environmental changes can cause agents to misinterpret their surroundings, as they cannot foresee how other agents' actions will alter the environment. This leads to a complex interplay between agents, where the actions of one can ripple through the system, affecting the decisions and effectiveness of others.

To address these problems, research must delve into advanced algorithms for decision-making, task allocation, communication protocols, and agent modeling. Solutions must consider agent heterogeneity, varying levels of communication, and the balance between centralized and decentralized control. Moreover, it must take into account the potential for asynchronous actions to create a cascading effect through the environment, requiring robust strategies that can adapt to these dynamic and interdependent conditions.

In essence, the multi-agent patrolling problem represents a multifaceted challenge, intertwining intelligent decision-making, complex coordination, and adaptive responsiveness to an ever-changing environment.

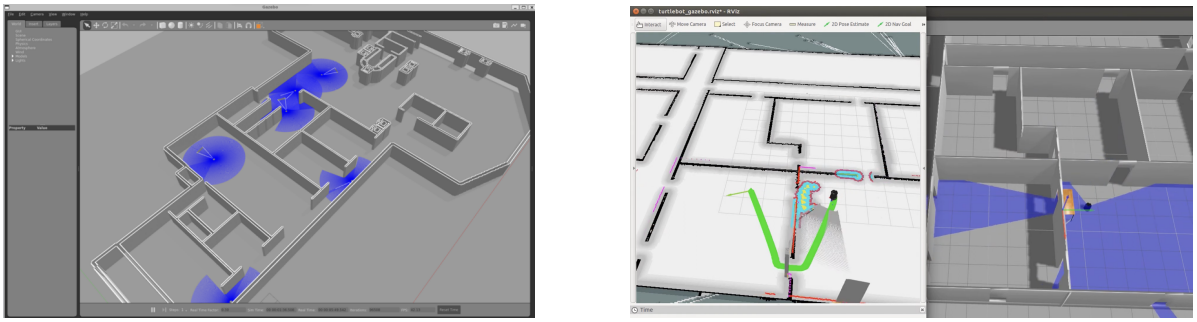


Figure 1.2: Multi-Agent simulation with ROS

1.3 Mathematical Foundation & Abstraction

The environment for multi-agent patrolling is abstracted into a graph $G = (V, E)$, where V represents the vertices (points of interest) and E symbolizes the edges connecting these points.

Each vertex $v \in V$ is represented by a two-dimensional coordinate in the vector space $\mathbb{R}^2$. Let $v = (x, y)$ denote the coordinates of a vertex v, where x and y are the horizontal and vertical positions, respectively, in the plane.

The distance function $d : V \times V \to \mathbb{R}$ is defined to measure the distance between two vertices $v_i, v_j \in V$, where $v_i = (x_i, y_i)$ and $v_j = (x_j, y_j)$. This distance is calculated using the Euclidean norm, leading to the following expression:

$$d(v_i, v_j) = ||v_i - v_j||_2 = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Here, the notation $|| \cdot ||_2$ refers to the 2-norm or Euclidean norm, which computes the ordinary straight-line distance between two points in a plane.

This mathematical definition enables precise and consistent measurements of distances between points of interest in the environment. By representing the vertices as two-dimensional coordinates and defining the distance function in terms of the Euclidean norm, we establish a theoretical foundation that can be systematically applied in algorithms and simulations.

Furthermore, the concept of idle time is introduced to capture the duration since a vertex's last visit.

Definition: Idle Time

Let $G = (V, E)$ be a graph representing the patrolling environment, where V denotes the set of vertices (nodes) and E the set of edges. The idle time, denoted $\zeta(v)$, of a vertex $v \in V$, represents the time elapsed since the last visit by an agent to that vertex. Formally, if $t(v)$ denotes the most recent time vertex v was visited and t is the current time, then:

$$\zeta : V \to \mathbb{R}^+$$

$$\zeta(v) = t - t(v)$$

where t represents the current time, and $t_{\text{last visit}}$ is the timestamp of the last visit to vertex v. The range of ζ in $\mathbb{R}^+$ ensures that the idle time is always non-negative.

These mathematical frameworks and concepts enable a more structured and thorough examination of the issue. By converting the environment from a multifaceted multi-dimensional Euclidean space into a graph and utilizing exact mathematical methods, we can simplify the process. While we use ROS to simulate the robot, the graph-based approach allows us to utilize more straightforward tools like Python and NetworkX.

1.4 Simulations and Experimental Graph Structures

For our experimentation, we employ two distinct graph structures to validate and test our algorithms.

1. **4-node Graph:** This is a relatively simple graph comprising of 4 nodes. Being almost fully connected, its primary purpose is to serve as a preliminary testing ground. With its simplicity, this graph allows us to validate the fundamental correctness of the algorithm, ensuring agents can navigate and operate efficiently within basic environments.

2. **Cumberland Graph:** A more intricate structure, the Cumberland graph represents a realistic and complex environment with a total of 40 nodes. This graph serves a dual purpose. Firstly, it tests the scalability of our algorithm, ensuring its applicability in larger, more intricate environments. Secondly, by closely emulating real-world scenarios, the Cumberland graph allows for a more in-depth assessment of the algorithm's practicality and effectiveness.

By testing on both the simplistic 4-node graph and the complex Cumberland graph, we aim to provide a comprehensive evaluation of our algorithm's capabilities. From fundamental correctness to scalability and real-world applicability, this dual-graph approach ensures a robust assessment of our solution's overall performance.

1.5 Overview of the thesis

In the initial chapter, we provide an introduction to Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL), starting with elementary algorithms such as Q-Learning and

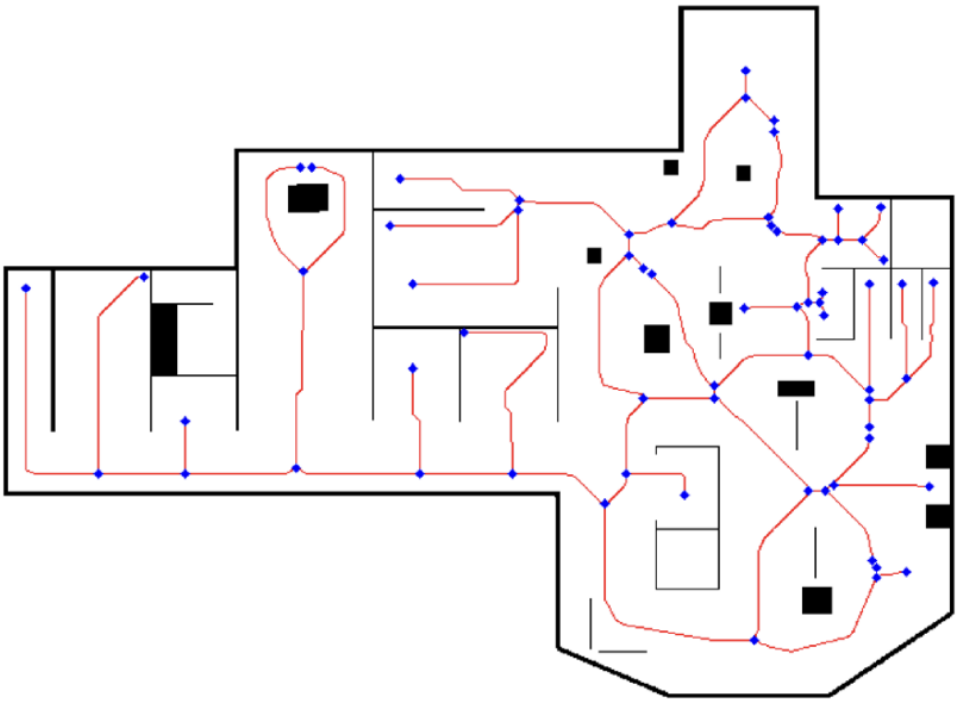


Figure 1.3: Cumberland Environment [6]

gradually delving into more advanced concepts like Deep Q-Learning (DQL) and Proximal Policy Optimization (PPO).

In a subsequent chapter, we discuss Multi Agent Reinforcement Learning and the challenges encountered while addressing the patrolling problem in a graph-based environment. These challenges range from observation and action spaces definition, agent coordination, to rewards shaping.

In the concluding chapter, we present the most promising numerical results obtained from our experiments. Although our approach demonstrates satisfactory performance on simple graphs, there is room for improvement particularly for large environments, where the complexity of the problem would necessitates new ideas. Despite the challenges faced, our research provides a foundation for future work in this area.

Chapter 2

Introduction to Reinforcement Learning

2.1 Fundamental Concepts of Reinforcement Learning

2.1.1 Reinforcement Learning Paradigm

Unlike Supervised Learning, where the model is provided with labeled data, or Unsupervised Learning, where data lacks explicit labels, in Reinforcement Learning (RL) the agent learns through trial and error, receiving feedback in the form of rewards or penalties. A distinguishing feature of RL is that the data used for training is not pre-collected or pre-labeled. Instead, it's generated dynamically as the agent interacts with its environment. This dynamic generation of training data means that the agent learns from its own actions and the subsequent feedback from the environment.

However, this approach presents certain challenges, especially in real-world scenarios. For instance, if we were training an RL agent to drive a car, we couldn't simply let the agent crash multiple times until it learns to drive correctly. This would be unsafe and economically infeasible. These risk highlights the need for careful strategies in RL training, such as the use of simulations or safety constraints.

2.1.2 Components of MDP

Central to the study of Reinforcement Learning (RL) is the concept of a Markov Decision Process (MDP). An MDP serves as a mathematical foundation, outlining the interaction between an agent and its environment over discrete time steps. This framework captures the dynamics of many decision-making situations where outcomes are partly random and partly under the control of the decision-maker (or the agent). An MDP characterizes this environment using a 4-tuple (S, A, P_a, R_a):

- S is **The Set of States**. For any given state-action pair $(s, a) \in S \times A$, there exists a subset of states $S(s, a) \subset S$ reachable by the agent.
- A is **The Set of Actions**. For any given state $s \in S$, there exists a subset of actions $A(s) \subset A$ available to the agent.

- $P_a(s', s)$: **The Transition Function** captures the transition dynamics of the environment. The transition function is defined as:

$$P_a(s'|s) = \Pr(s_{t+1} = s'|s_t = s, a_t = a)$$

It represents the probability of transitioning to state s' in the next time step when action a is taken in state s at the current time step.

- $R_a(s, s')$: **The Reward Function** provides the expected reward after transitioning from state s to state s' due to action a.

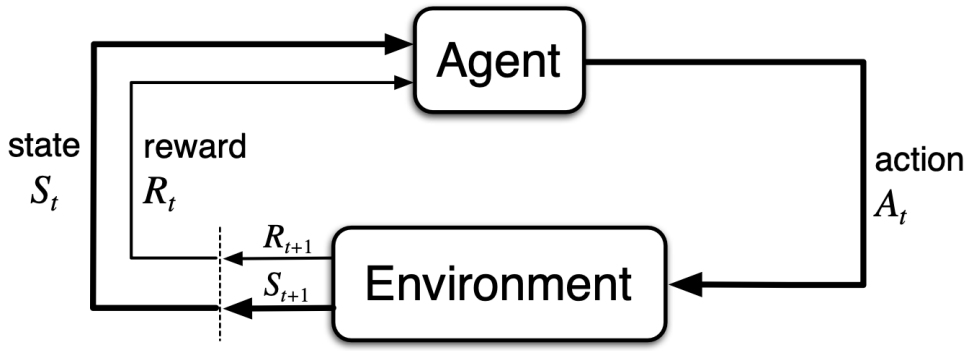


Figure 2.1: The agent–environment interaction in a Markov decision process [8]

By understanding these components, an agent can learn optimal actions in each state to achieve the desired outcomes in a given task.

The Markov Property

The Markov property of a Markov Decision Process (MDP) asserts that the future, given the present, is independent of the past. In other words, the state captures all relevant information such that the system's future dynamics are independent of the sequence of states and actions that led to the current state.

$$P(s_{t+1}|s_0, a_0, \ldots, s_t, a_t) = P(s_{t+1}|s_t, a_t)$$

One way to understand the flow of interaction in the MDP is by defining τ the trajectory through sequences of states, actions, and rewards:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots)$$

Furthermore, the interaction might have a finite horizon or continue indefinitely. A finite horizon implies that there's a terminal time step T, beyond which the interaction doesn't proceed. In such cases, the trajectory would look like:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T)$$

To illustrate, consider a car navigating through an environment. If the car successfully reaches its destination, it may receive a final reward, $r_T = +10$. Conversely, if the car crashes along its journey, it could incur a significantly negative reward, such as $r_T = -100$, indicating a highly undesirable outcome.

2.1.3 Reward and Policy

Rewards play a crucial role in reinforcement learning as they signal the agent about its performance. The agent's primary objective is to take actions that maximize the cumulative reward over time, thus achieving its goal. This decision-making strategy of the agent is governed by a policy, denoted as π. The policy is essentially a guide for the agent that dictates which action to take given the current state.

A policy, π, can be formally defined in two primary ways: deterministic and stochastic.

- **Deterministic Policy:** A deterministic policy is a mapping from the state space S to the action space A. That is, for every state $s \in S$, the policy π specifies a particular action $a \in A$ that the agent should take. It can be defined as follows:

$$\pi : S \to A$$

$$\pi(s) = a$$

  Here, the action selected by the policy $\pi(s)$ is always the same for any given state s. There's no randomness in the action selection process, making it straightforward but potentially less flexible.

- **Stochastic Policy:** A stochastic policy, on the other hand, defines a probability distribution over the action space A for each state $s \in S$. It provides the probability of selecting each action $a \in A$ when in state s. This type of policy can be expressed as:

$$\pi : S \times A \to [0, 1]$$

$$\pi(a|s) = \Pr(a_t = a|s_t = s)$$

  This approach introduces randomness into the decision-making process, which can be beneficial in environments with uncertainties or when exploring various strategies to find an optimal solution.

The choice between deterministic and stochastic policies largely depends on the nature of the environment and the specific goals of the learning task. While deterministic policies can be efficient in stable and predictable settings, stochastic policies often prove advantageous in dynamic or non-deterministic scenarios, allowing the agent to explore and adapt.

2.1.4 Rewards, Returns, and Value Functions

The agent's objective is to maximize the cumulative rewards over time. However, in some environments, the reward structure may not be immediately informative. Consider an environment such as the Frozen Lake from the reinforcement learning framework Gymnasium.

Figure 2.2: Frozen Lake environment (Gymnasium)

There exists only one state-action pair that triggers a positive reward, signifying the goal for the agent. The reward is received when the agent, positioned at the bottom right, decides to move right to get the gift. If the agent falls, the episode ends with a score of 0, indicating no rewards received. One challenge is that, initially, the agent is positioned far away from the goal and lacks any indicator of its distance from the reward.

To address this, we introduce the notion of the *return* G_t, which is the discounted sum of rewards from time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

Here, $\gamma \in [0, 1]$ is the discount factor. It represents the agent's consideration for future rewards. A value of γ close to 0 makes the agent myopic, focusing on immediate rewards, whereas a value close to 1 makes it far-sighted, emphasizing long-term rewards.

However, relying solely on G_t can be problematic as it corresponds to a specific trajectory τ. To generalize over possible trajectories, we introduce the *state-value function* $V^\pi(s)$.

Definition : The State-Value Function

The state-value function, $V^\pi(s)$, for a policy π, represents the expected return when starting from state s and following π thereafter. It is given by:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Where G_t is the return at time t and S_t is the state at time t.

Here, the expectation is taken considering all the possible trajectories the agent can follow from state s, while adhering to the policy π. It's important to note that $V^\pi(s)$ is dependent on π as the agent's behavior, and hence the return, varies with the policy. Understanding the value functions helps in learning the optimal policy that maximizes the expected return, leading to more effective decision-making by the agent. The state-value function, V, can

also be described using the action-value function, Q:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t|S_t = s] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s)\mathbb{E}_{\pi}[G_t|S_t = s, A_t = a] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(a|s)Q^{\pi}(s,a) \end{aligned} \tag{2.1}$$

Figure 2.3: Estimated State-Value V on Frozen Lake simulation

Furthermore, to evaluate how good an action is in a particular state, we can define the *action-value function* $Q^{\pi}(s,a)$:

Definition : The Action-Value Function

The action-value function, $Q^{\pi}(s,a)$, for a policy π, represents the expected return when starting from state s, taking action a, and following π thereafter. It is given by:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|S_t = s, A_t = a]$$

Where G_t is the return at time t, S_t is the state at time t, and A_t is the action taken at time t.

While $V(s)$ provides a value of being in state s, it remains silent about which specific action to choose next. In contrast, $Q(s,a)$ for each $a \in A(s)$ quantifies the value of selecting action a when in state s. When an agent finds itself in state s, it benefits from Q by directly comparing the values associated with each action $a \in A(s)$, thereby facilitating an informed decision.

The action-value function, Q, can be elucidated through the lens of the state-value function, V.

$$
\begin{aligned}
Q^{\pi}(s,a) &= \mathbb{E}_{\pi}[G_t|S_t=s,A_t=a] \\
&= \underbrace{\mathbb{E}_{\pi}[R_{t+1}|S_t=s,A_t=a]}_{\text{Immediate reward}} + \gamma \underbrace{\mathbb{E}_{\pi}[G_{t+1}|S_t=s,A_t=a]}_{\text{Discounted future reward}} \\
&= \sum_{\substack{s'\in\mathcal{S} \\ r\in\mathcal{R}}} p(s',r|s,a)\left(r+\gamma\mathbb{E}[G_{t+1}|S_{t+1}=s']\right) \\
&= \sum_{\substack{s'\in\mathcal{S} \\ r\in\mathcal{R}}} p(s',r|s,a)[r+\gamma V^{\pi}(s')]
\end{aligned}
\tag{2.2}
$$

Even though the learning of $Q^{\pi}(s,a)$ offers significant benefits it tends to be more computationally demanding and needs more data for effective learning compared to $V^{\pi}(s)$. This complexity arises because obtaining a reliable estimate for $V^{\pi}(s)$ needs the data to encompass the state space sufficiently, but to get a reliable estimate for $Q^{\pi}(s,a)$, the data must cover all possible (s,a) combinations, and not just all states s.

2.1.5 The Bellman Equations

The Bellman Equations

One fundamental relationship is derived by substituting the equation for $Q^{\pi}(s,a)$ from (2.2) into the equation for $V^{\pi}(s)$ from (2.1), which gives us:

$$V^{\pi}(s) = \sum_{a}\pi(a|s)\sum_{s',r}p(s',r|s,a)[r+\gamma V^{\pi}(s')]$$

Conversely, by incorporating the $V^{\pi}(s)$ expression from (2.1) into the $Q^{\pi}(s,a)$ equation from (2.2), we arrive at:

$$Q^{\pi}(s,a) = \sum_{s',r}p(s',r|s,a)\left(r+\gamma\sum_{a'}\pi(a'|s')Q^{\pi}(s',a')\right)$$

The Bellman Equations provide an elegant way to relate the value of a state (or state-action pair) to the values of its successor states (or state-action pairs), serving as a foundational tool in reinforcement learning for estimating value functions.

2.1.6 Optimality

In Reinforcement Learning (RL), the notion of optimality is central to our understanding and design of algorithms. At its heart, RL is about learning how to act optimally in an environment to achieve the highest possible cumulative reward. The quest for optimal behavior requires understanding and defining what "optimal" really means in this context.

The policy, denoted as π, provides a mapping from states to actions. It's essentially the agent's strategy; in every state, it suggests an action for the agent to perform. However some lead to better outcomes in certain environments, and the challenge is to find the policy that performs the best, given the dynamics of the environment and the agent's objectives.

The optimal state-value function, V^*, provides a glimpse into this optimality. For each state, it indicates the maximum expected cumulative reward an agent can achieve, starting from that state and acting optimally thereafter:

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.3}$$

Likewise, the optimal action-value function, Q^*, expands on this idea by determining the expected return for taking a particular action in a state and then following the best possible policy:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{2.4}$$

The main objective in this quest for optimality is to identify the optimal policy, π^*. This policy suggests the most favorable action in any state to maximize the expected cumulative reward over time.

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{2.5}$$

In essence, π^* encapsulates the best strategy for the agent, given its understanding of the environment. For every state $s \in S$, π^* will dictate the action a that promises the highest future rewards. This is a powerful idea, suggesting that if an agent follows π^*, it is guaranteed to act in the most favorable manner, always choosing actions that lead to the most rewarding outcomes in the long run.

Determining the optimal policy, π^*, is challenging due to the expansive state and action spaces and the unpredictable nature of many environments. The role of RL algorithms is to navigate these complexities to find π^*. Among the various algorithms developed for this purpose, we will first delve into Q-learning in the next chapter.

With the formulation of the optimality equation as stated previously, the Bellman equation can be transformed into the Bellman Optimality Equations.

The Bellman Optimality Equations

The optimal value function $V^*(s)$ and the optimal action-value function $Q^*(s,a)$ are intertwined through the following relationships:

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a)\,(r + \gamma V^*(s'))$$

These equations provide a recursive definition for determining the optimal values of states and actions in a given environment.

Let prove those new equations. Given that $V^*(s) = \max_\pi V^\pi(s)$, we can conclude that the optimal policy π^* will always select the most favorable action. Consequently, under the optimal policy π^*:

$$\pi^*(a'|s') = \begin{cases} 1 & \text{if } a' = \arg\max_{a''} Q^*(s', a'') \\ 0 & \text{otherwise} \end{cases}$$

This implies:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ &= \max_\pi \sum_{a \in \mathcal{A}(s)} \pi(a|s) Q^\pi(s, a) \\ &= \max_a Q^*(s, a) \end{aligned}$$

Given the optimality formulation, we can derive the optimal action-value function, $Q^*(s, a)$. Specifically, $Q^*(s, a)$ is defined as the maximum expected return when starting in state s, taking action a, and thereafter following an optimal policy. This leads to the following relationship:

$$\begin{aligned} Q^*(s, a) &= \max_\pi Q^\pi(s, a) \\ &= \max_\pi \sum_{s', r} p(s', r|s, a) \left(r + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s', a') \right) \\ &= \sum_{s', r} p(s', r|s, a) \left(r + \gamma \max_{a'} Q^*(s', a') \right) \\ &= \sum_{s', r} p(s', r|s, a) \left(r + \gamma V^*(s') \right) \end{aligned}$$

□

2.2 Basics of Q-Learning

2.2.1 Updating rule

Given the Bellman optimally equation for Q-values under a policy π:

$$Q^*(s, a) = \sum_{s', r} p(s', r|s, a) \left(r + \gamma \max_{a'} Q^*(s', a') \right)$$

One of the significant differences from the theory is that we typically do not have access to the full transition model $p(s', r|s, a)$. However, by interacting with the environment, we can sample from this model and gather empirical data regarding state transitions and rewards.

A fundamental component of Q-learning is the update rule which adjusts our current estimate of the Q-values based on new experiences. The equation that encapsulates this rule is:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a')\right) \tag{2.6}$$

The term α in the equation, often referred to as the learning rate, plays a crucial role in the algorithm. It determines the weight given to the new sample relative to the current Q-value estimate.

2.2.2 Exploration, Exploitation Trade-Off

Agent's primary objective is to learn the optimal strategy by navigating through the environment. However, to achieve this, the agent faces a pivotal dilemma: Exploration vs. Exploitation.

Exploration: When an agent is introduced to an environment, its knowledge about the state transitions and rewards is often minimal or non-existent. In such scenarios, the agent must explore various actions in different states to gather knowledge. Exploration involves taking actions that are not necessarily believed to be optimal but can provide new information about the environment.

Exploitation: As the agent interacts more with the environment and accumulates knowledge about state transitions and rewards, it starts developing a sense of which actions are likely to be optimal in various states. Exploitation involves taking actions that the agent believes, based on its current knowledge, will yield the highest rewards.

The key challenge is to decide how fast the agent should transition from an explorative strategy to an exploitative one. This is known as the exploration vs. exploitation tradeoff. One popular method to address this dilemma is the ϵ-greedy strategy. In the ϵ-greedy approach, the agent chooses a random action with probability ϵ (exploration) and chooses the best-known action with probability $1-\epsilon$ (exploitation). Initially, ϵ can be set to a high value (e.g., 1) to prioritize exploration. As the agent learns more about the environment, ϵ can be reduced to shift the emphasis toward exploitation.

The reduction in ϵ can be achieved in various ways:

- **Linear Decay:** This involves decrementing ϵ linearly over time. A simple model for this can be:

  $$\epsilon_t = \epsilon_0 - \delta t$$

  where δ is the decay rate.

- **Exponential Decay:** This involves an exponential reduction in ϵ. It can be modeled as:

  $$\epsilon_t = \epsilon_0 e^{-\lambda t}$$

  where λ is the decay constant.

The correct choice of decay strategy and its parameters depends on the specific characteristics and requirements of the environment and task.

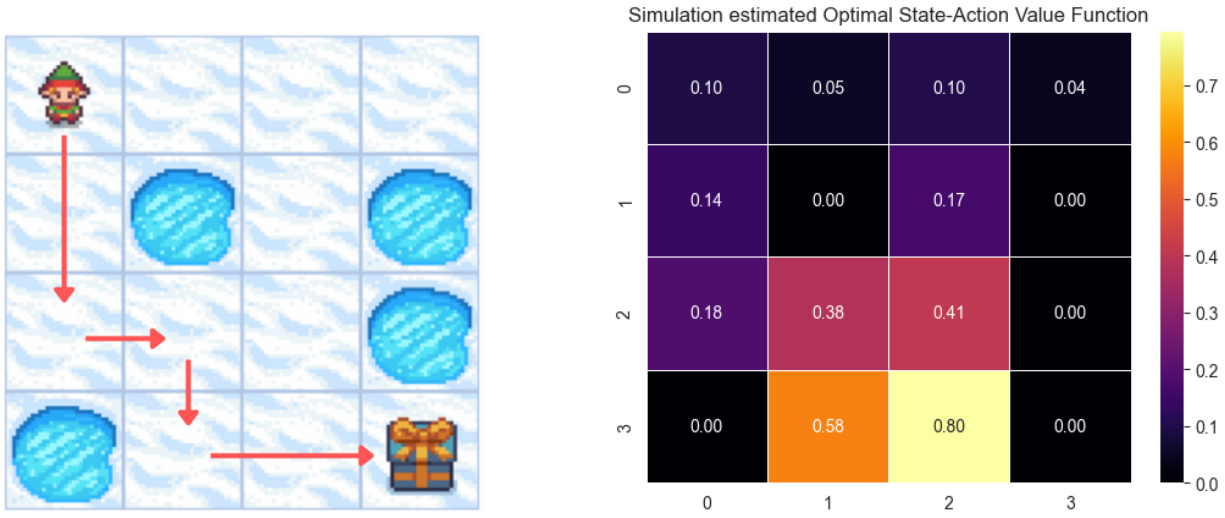


Figure 2.4: State-Value based optimal path on Q-Learning Simulation of Frozen-Lake

2.2.3 Q-Learning Algorithm

Initially, we set up a matrix $Q \in \mathbb{R}^{n \times p}$ of dimensions $n \times p$ where n is the number of states, represented as $|S|$, and p is the number of possible actions, represented as $|A|$. Essentially, this matrix is the Q-value table, where each entry $Q(s, a)$ corresponds to the Q-value of taking action a in state s.

$$Q = \begin{bmatrix} Q(s_1, a_1) & Q(s_1, a_2) & \dots & Q(s_1, a_p) \\ Q(s_2, a_1) & Q(s_2, a_2) & \dots & Q(s_2, a_p) \\ \vdots & \vdots & \ddots & \vdots \\ Q(s_n, a_1) & Q(s_n, a_2) & \dots & Q(s_n, a_p) \end{bmatrix} \tag{2.7}$$

In the beginning, the agent's interactions with the environment are largely explorative due to a high value of ϵ (close to 1). This means the agent will predominantly take random actions to learn about the environment. As experiences are accumulated and the Q-value matrix is updated (based on the rule in Eq. (2.6)), the value of ϵ is reduced, allowing the agent to exploit its knowledge more frequently by taking actions that are believed, based on the updated Q, to yield the highest rewards.

In the simulations executed using the Q-Learning algorithm, we observe an upward trend in the proportion of successful episodes. Note that the *Frozen Lake* environment was configured with the `is_slippery=True` parameter. This implies that the agent, when navigating the icy environment, has a significant probability of not landing in the intended subsequent position . Therefore the optimal state-value function obtained is not 1 for the state adjacent to the terminal state, but rather 0.8 due to the uncertainty associated with navigating on slippery ice.

Using these state-values, we can trace the optimal route from the starting state to the terminal state. This can be accomplished by consistently choosing the adjacent state with the superior state-value, continuing this process until the terminal state is attained.

Furthermore, when the environment is simulated with the `is_slippery=False` setting,

Algorithm 1 Q-Learning

procedure TakeAction(s)
    Adjust the value of ϵ according to the current step.
    $u \sim Unif(0, 1)$
    if $u > \epsilon$ **then**
        Take random action $a_t \in A(s)$ ▷ Exploration
    else
        Take action $a_t = \arg\max_{a \in A(s)} Q(s, a)$ ▷ Exploitation
    return a_t, s_{t+1}

Instantiate null matrix $Q \in \mathbb{R}^{n \times p}$ where $n = |S|$ and $p = |A|$
for $n = 1$ **to** N **do**
    $s_1 \leftarrow$ ResetEnvironment()
    for $t = 1$ **to** 100 **do**
        $a_t, s_{t+1} \leftarrow$ TakeAction(s_t)
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) \times (1 - \alpha) + \alpha(r + \gamma \max_a Q(s_{t+1}, a))$

the agent achieves high performance. It's worth mentioning that the displayed line plot has been smoothed using a window size of 1000. This was necessary because, without smoothing, the raw values oscillate solely between 0 and 1, offering limited insight into the underlying trends.

2.2.4 Limitation

Traditional Reinforcement Learning (RL) methods demonstrate good performance when utilizing tabular functions, such as matrices. However, there exists a prominent limitation in these techniques. In the scenarios presented thus far, the state space was assumed to be discrete. In most real-world scenarios, the state space is continuous (like coordinates in Euclidean space). In some situations, even the action space is continuous like determining the precise angle by which to turn a steering wheel, rather than merely choosing directions like left or right.

Such continuous spaces render the previously discussed methods impractical. Using tables to represent value functions or policies for continuous states and actions would require infinite entries, making it infeasible to implement. The challenges posed by continuous spaces, coupled with the need for more scalable and generalizable solutions, lead us to explore a more advanced paradigm: Deep Reinforcement Learning. This will be the focal point of our discussion in the next chapters.

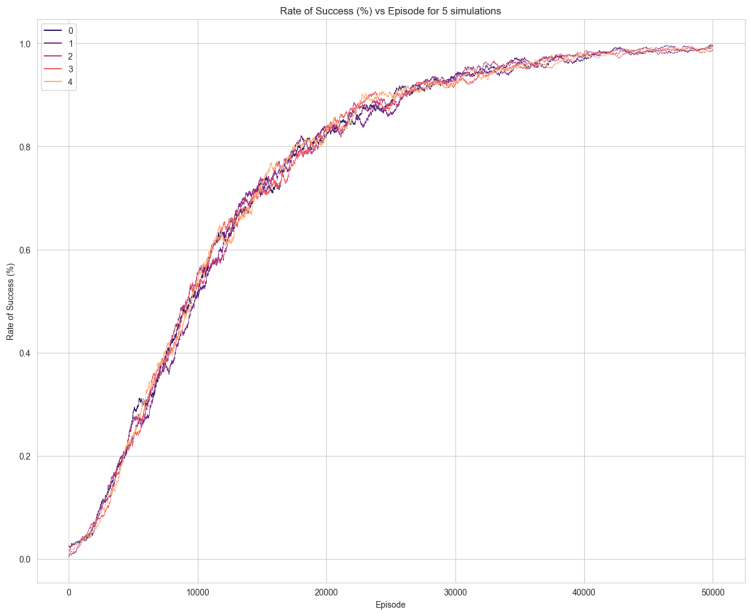


Figure 2.5: Q-Learning simulation

2.3 Deep Learning

2.3.1 Forward propagation

Deep Learning involves the use of neural networks with multiple layers (deep architectures) to analyze various kinds of data. A neural network is composed of layers of interconnected nodes or *neurons*. Each connection has an associated weight, which is adjusted during training. The basic unit, the neuron, computes a weighted sum of its inputs and passes it through an activation function.

$$y = \sigma\left(\mathbf{w} \cdot \mathbf{x} + b\right) \tag{2.8}$$

Where:

- $\mathbf{w}$ is the weight matrix.
- $\mathbf{x}$ is the input vector.
- b is the bias.
- σ is the activation function, e.g., ReLU, sigmoid, tanh.

Consider a deep neural network with L layers. For each layer l, let $\mathbf{w}^{[l]}$ represent the weight matrix and $\mathbf{b}^{[l]}$ the bias vector. The activation function is denoted by σ.

The output of layer l can be defined recursively as:

$$\mathbf{z}^{[l]} = \mathbf{w}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \tag{2.9}$$

$$\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]}) \tag{2.10}$$

Where:

- $\mathbf{a}^{[l]}$ is the activation (output) of the l^{th} layer.
- $\mathbf{z}^{[l]}$ is the pre-activation value.
- $\mathbf{a}^{[0]} = \mathbf{x}$, the input to the network.

By iterating through all the layers using the above equations, the output of the network is $\mathbf{a}^{[L]}$.

2.3.2 Training and Backward Propagation

Neural networks, often expressed in terms of their parameters θ (which include both weights and biases), are optimized to find the best fit for a given dataset. Central to this optimization process is the definition and minimization of a loss function.

For each training example $(x^{(i)}, y^{(i)})$, the per-example loss is denoted by $L(x^{(i)}, y^{(i)}, \theta)$. The overall objective during training is to minimize the expected loss across all training examples. The cost function $J(\theta)$ can be defined as the average loss over the dataset:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}}\left[L(x, y, \theta)\right]$$

The optimization algorithm, gradient descent, works by iteratively updating the parameters in the direction that minimizes $J(\theta)$. The gradient of $J(\theta)$ is given by:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

Here, m is the number of training examples. The parameter θ is then updated in the opposite direction of this gradient to reduce the loss. The primary objective is to minimize a loss function $L(x, y, \theta)$, which quantifies the discrepancy between the model's predicted output and the actual target value. The parameters θ, which include both weights and biases, are iteratively fine-tuned to minimize this loss.

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha\nabla_\theta L(x, y, \theta_{\text{old}})$$

Here, α signifies the learning rate, and $\nabla_\theta L(x, y, \theta_{\text{old}})$ represents the gradient of the loss function concerning the parameters θ.

At the beginning, the policy parameters θ are generated randomly, and an episode τ is completed by following a policy π_θ, which is defined as:

$$\tau \sim \pi_\theta$$

The objective function $J(\pi_\theta)$ can be the expected cumulative reward and is defined as:

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

where γ is the discount factor, T is the time horizon, and r_t represents the reward at time t.

The policy parameters θ are then updated in the direction that maximizes this objective function. The update equation is:

$$\theta_{\text{new}} = \theta_{\text{old}} + \alpha \nabla_\theta J(\pi_\theta)$$

Here, α is the learning rate, and $\nabla_\theta J(\pi_\theta)$ is the gradient of the objective function with respect to the policy parameters θ.

Backpropagation, short for "backward propagation of errors," is a widely-used algorithm in training feedforward neural networks for supervised learning tasks. It computes the gradient of the loss function with respect to each weight by employing the chain rule of calculus.

Chain Rule

The essence of backpropagation can be attributed to the chain rule of differentiation. If we have a composite function given by $y = f(u)$ and $u = g(x)$, then the derivative of y with respect to x can be expressed as:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Application in Neural Networks

Consider a neural network with layers indexed by l. Let the loss function be denoted by L. For a given layer l, the pre-activation is $z^{[l]}$ and the activation is $a^{[l]}$. The weight and bias for this layer are $w^{[l]}$ and $b^{[l]}$, respectively.

The goal is to compute the gradients:

$$\frac{\partial L}{\partial w^{[l]}}$$

and

$$\frac{\partial L}{\partial b^{[l]}}$$

Using the chain rule, the gradient of the loss with respect to the weight is:

$$\frac{\partial L}{\partial w^{[l]}} = \frac{\partial L}{\partial z^{[l]}} \cdot \frac{\partial z^{[l]}}{\partial w^{[l]}}$$

Here, $\frac{\partial L}{\partial z^{[l]}}$ (often denoted by $\delta^{[l]}$) is the error term for layer l. It can be computed recursively starting from the final layer. For the final layer, the error term is simply the derivative of the loss function with respect to its input. For hidden layers, it's calculated as:

$$\delta^{[l]} = (w^{[l+1]T} \delta^{[l+1]}) \odot \sigma'(z^{[l]})$$

where $\odot$ denotes element-wise multiplication and σ' is the derivative of the activation function.

Once $\delta^{[l]}$ is known, we can compute:

$$\frac{\partial L}{\partial w^{[l]}} = \delta^{[l]} a^{[l-1]T}$$

and

$$\frac{\partial L}{\partial b^{[l]}} = \delta^{[l]}$$

These gradients can then be used in optimization algorithms, like gradient descent, to update the weights and biases of the network.

2.4 Deep Q-Learning (DQL)

2.4.1 Discrete State Space

The Lunar Lander task is a classical reinforcement learning environment provided by the Gymansium library. The main goal is to navigate the lander to softly touch down on a specific landing pad. This is done by firing the main and side engines appropriately to control its descent and horizontal position.

Let the input space S and the output space A be defined as:

$$S = \left\{ (x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, \text{Leg}_1, \text{Leg}_2) \mid x, y, \dot{x}, \dot{y}, \theta, \dot{\theta} \in \mathbb{R}, \text{Leg}_1, \text{Leg}_2 \in \{0, 1\} \right\}$$
$$A = \{0, 1, 2, 3\}$$

where

- x, y: Coordinates of the lander.
- $\dot{x}, \dot{y}$: Linear velocities in the x and y directions, respectively.
- θ: Angle of the lander with respect to the upright position.
- $\dot{\theta}$: Angular velocity of the lander.
- $\text{Leg}_1, \text{Leg}_2$: Booleans representing whether the left and right legs of the lander, respectively, are in contact with the ground.

The neural network f is then a function that maps from the input space to the output space:

$$f \approx Q : S \to A$$

Here, f is trying to estimate the function $Q(s, a)$, where $s \in S$ represents the state of the lander and $a \in A$ represents the action taken.

2.4.2 DQL Algorithm

Deep Q-Learning (DQL) introduces several key differences compared to traditional Q-Learning, enhancing its efficiency and stability.

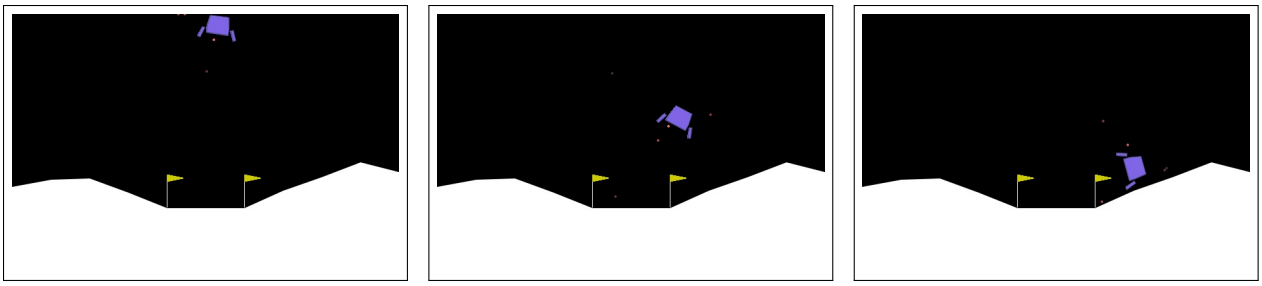


Figure 2.6: Unsuccessful Lunar Lander Episode (Gymnasium)

Utilization of Neural Networks: Unlike Q-Learning, which uses a tabular representation for the Q-values, DQL leverages neural networks to approximate the Q-function. The Q-function is represented as $Q(s, a; \theta)$, where s is the state, a is the action, and θ are the parameters of the neural network.

Experience Replay Buffer: Experience Replay is a crucial innovation in Deep Q-Learning (DQL). In traditional learning, the agent learns from each experience once and then forgets it. In contrast, DQL employs a replay buffer $\mathcal{D}$ to store past experiences or transitions. Each experience (s_t, a_t, r_t, s_{t+1}) is stored in the replay buffer $\mathcal{D}$, where s_t is the current state, a_t is the action taken, r_t is the received reward, and s_{t+1} is the next state. During training, mini-batches of experiences are randomly sampled from $\mathcal{D}$. This random sampling breaks the temporal correlations between consecutive samples, improving stability and convergence. Experiences in the buffer can be reused multiple times, allowing the agent to learn more from the same data.

In the Deep Q-Learning algorithm with Target Network (DQL), two networks are employed: Q_{network} and Q_{target}.

- $Q_{\textbf{network}}$: It is utilized for action selection according to the ϵ-greedy strategy, and it is updated continually throughout the learning process. This network helps generate the episode by taking the next action based on the current policy, balancing exploration and exploitation.

- $Q_{\textbf{target}}$: The target Q-network, denoted by Q_{target}, is updated less frequently than the main Q-network, denoted by Q_{network}, to enhance stability in the learning process. The update is performed periodically using the weights from both networks. If θ represents the weights of the main Q-network, and θ_{target} represents the weights of the target Q-network, the update is done as follows:

$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{network}} + (1 - \tau)\theta_{\text{target}}$$

  where τ is a blending factor that lies between 0 and 1. This approach of softly blending the weights is used to ensure a smoother transition, thereby helping in improving the stability of the learning process.

The use of Q_{target} improves stability in training by decoupling the target from the parameters being updated. By keeping the target values fixed over a number of updates, the

Algorithm 2 Deep Q-Learning with Target Network (DQL)

procedure TakeAction(s)
    Adjust the value of ϵ according to the current step.
    $u \sim Unif(0, 1)$
    if $u > \epsilon$ **then**
        Take random action $a_t \in A(s)$ ▷ Exploration
    else
        Take action $a_t = \arg\max_{a \in A(s)} Q_{\text{network}}(s, a)$ ▷ Exploitation
    return a_t, s_{t+1}

Initialize neural network Q_{network} for approximating Q function
Initialize target network Q_{target} with weights $\theta_{\text{target}} = \theta_{\text{network}}$
Initialize replay buffer $\mathcal{D}$
Define γ, the discount factor, and τ, the target network mixing factor
for $n = 1$ **to** N **do**
    $s_1 \leftarrow$ ResetEnvironment()
    for $t = 1$ **to** T **do**
        $a_t, s_{t+1} \leftarrow$ TakeAction(s_t)
        Store transition (s_t, a_t, r, s_{t+1}) in replay buffer $\mathcal{D}$
        if $n \equiv 0 \pmod{f_{train}}$ **then**
            Sample mini-batch of transitions of size d from $\mathcal{D}$
            Compute target $y_j = r + \gamma \max_a Q_{\text{target}}(s_{t+1}, a)$
            Update Q_{network} by minimizing $\mathcal{L} = \frac{1}{b} \sum_{j=1}^{b} (y_j - Q_{\text{network}}(s_t, a_t))^2$
        if $n \equiv 0 \pmod{f_{target\,train}}$ **then**
            $\theta_{\text{target}} \leftarrow \tau \theta_{\text{network}} + (1 - \tau)\theta_{\text{target}}$

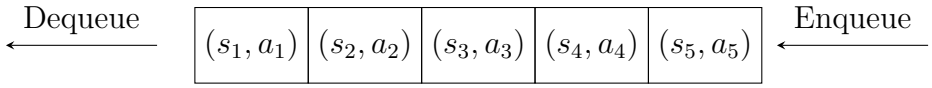


Figure 2.7: Queue representation in Replay Buffer $\mathcal{D}$. New elements are enqueued from the right, and the oldest elements are dequeued from the left.

learning process becomes more stable, mitigating the risk of oscillations or divergence in the learning process. Without Q_{target}, the network may face challenges. For example, the target values would change at each step since they would depend on the constantly updated Q_{network}, causing the learning process to chase a non-stationary target.

By incorporating Q_{target}, the DQL algorithm mitigates these risks, leading to a more robust and stable convergence.

In the beginning stages of the training process (lines 10 to 15), the Replay Buffer $\mathcal{D}$ is initially empty or too small to provide meaningful training samples. Therefore, these lines are ignored, and the algorithm focuses on populating $\mathcal{D}$ with initial experiences.

- **Filling the Replay Buffer**: Before learning starts, the algorithm performs actions in the environment and stores the transitions in $\mathcal{D}$, thereby filling it with sufficient data for training.
- **Learning Phase**: Once $\mathcal{D}$ is filled to a predefined capacity, the learning phase begins. At this point, the algorithm starts to sample mini-batches of transitions for training, as described in lines 10 to 15.
- **Queue Structure**: The Replay Buffer $\mathcal{D}$ operates as a queue, adhering to a First-In-First-Out (FIFO) principle. When $\mathcal{D}$ reaches its capacity, the oldest transitions are removed to make room for new ones. The operation of enqueue and dequeue in a queue can be mathematically represented as:

$$\text{enqueue}(q, x) : q \leftarrow q \cup \{x\} \text{ if } |q| < \text{capacity} \tag{2.11}$$

$$\text{dequeue}(q) : q \leftarrow q \setminus \{\min q\} \tag{2.12}$$

  where q represents the queue, x is the new element, and capacity is the predefined size of the queue.

In typical applications of the Deep Q-Learning algorithm, certain parameters are chosen based on empirical insights and the nature of the environment. Specifically, the discount factor γ is often set to a value close to 1, such as 0.99, to emphasize the importance of long-term rewards. A common batch size for training is 512, as this provides a good balance between computational efficiency and the diversity of experiences in each training step.

The algorithm may wait for a substantial number of steps, such as 10000, before starting the learning process. This delay ensures that the replay buffer $\mathcal{D}$ is filled with a rich set of experiences, providing a more representative sample for training. The size of $\mathcal{D}$ is usually chosen to be several times larger than a single episode, often 3 to 5 times bigger. This large buffer size ensures a good complete sample of experiences for training.

Especially for problems where the initial state of the environment may have specific characteristics that don't provide useful data for training (e.g., in our problem where all nodes have the same idle time at the beginning), this approach ensures that the algorithm has access to diverse and informative experiences before learning begins. It helps in training a more robust and well-generalized model.

2.5 Proximal Policy Optimization (PPO)

Unlike DQL, which is value-based and relies on estimating the action values Q, the Proximal Policy Optimization (PPO) algorithm is a policy-based method. In PPO, the neural network directly models the policy $\pi_\theta(a|s)$, where θ represents the parameters of the network.

The neural network outputs a vector of unnormalized scores $\mathbf{z}$, representing the last layer of the network for action a in state s. These scores are then passed through a softmax logistic function to obtain a proper probability distribution:

$$\pi_\theta(a|s) = \frac{e^{z(a,s)}}{\sum_{a'} e^{z(a',s)}}$$

Where $z(a, s)$ is the output of the neural network for action a in state s, and the denominator ensures that the probabilities sum to one.

The resulting policy is stochastic, allowing for exploration of the action space. The ultimate goal is to update $\pi_\theta(a|s)$ to optimize the objective function $J(\pi_\theta)$, which represents the expected cumulative reward:

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

Where γ is the discount factor, T is the time horizon, and r_t represents the reward at time t. By directly modeling the policy and optimizing this objective function, PPO can provide a more flexible and potentially more efficient approach to learning in complex environments compared to value-based methods like DQL.

To update the policy $\pi_\theta(a|s)$ and maximize the expected cumulative reward, we need to find the direction of the steepest ascent in the policy parameter space. The parameters are then updated according to the following rule:

$$\theta_{\text{new}} = \theta_{\text{old}} + \alpha\nabla_\theta J(\pi_\theta)$$

$$\Delta\theta = \theta_{\text{new}} - \theta_{\text{old}} = \alpha\nabla_\theta J(\pi_\theta)$$

We want to derive the gradient of the objective function $J(\pi_\theta)$, defined as:

Property : Gradient of the Objective Function

The gradient of the objective function $J(\pi_\theta)$ with respect to the policy parameters θ is given by:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau)\right]$$

Here, τ represents a trajectory sampled from the policy π_θ, and $G(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ is the expected cumulative reward for that trajectory. The expression reflects how the policy parameters should be updated to maximize the expected cumulative reward. It forms the basis for policy optimization algorithms such as PPO, allowing the model to learn better actions to take in different states.

Let $p(\tau|\theta)$ the probability of the trajectory τ given the policy parameters θ. It is an essential concept in policy gradient methods, encapsulating the likelihood of the actions taken according to the policy π_θ across the trajectory.

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau)\right] \\
&= \nabla_\theta \int p(\tau|\theta) G(\tau) d\tau \\
&= \int \nabla_\theta p(\tau|\theta) G(\tau) d\tau \\
&= \int \frac{p(\tau|\theta)}{p(\tau|\theta)} \nabla_\theta p(\tau|\theta) G(\tau) d\tau \\
&= \int p(\tau|\theta) \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)} G(\tau) d\tau \\
&= \int p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) G(\tau) d\tau \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log p(\tau|\theta) G(\tau)\right]
\end{aligned}$$

The relationship between $p(\tau|\theta)$ and $\pi_\theta(a|s)$ can be understood by considering how a trajectory is generated by a policy. A trajectory τ consists of a sequence of states and actions $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$, and the probability of this trajectory under the policy π_θ is given by:

$$p(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Here, $p(s_0)$ is the initial state distribution, $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under the policy π_θ, and $p(s_{t+1}|s_t, a_t)$ is the transition probability from state s_t to s_{t+1} given action a_t.

Taking the logarithm on both sides, we get:

$$\log p(\tau|\theta) = \log p(s_0) + \sum_{t=0}^{T-1} (\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t))$$

Differentiating both sides with respect to θ, we obtain:

$$\nabla_\theta \log p(\tau|\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Since $\nabla_\theta \log p(s_0)$ and $\nabla_\theta \log p(s_{t+1}|s_t, a_t)$ are both zero (as they do not depend on θ). □

2.5.1 Use of the Advantage Function in PPO

Proximal Policy Optimization (PPO) modifies the objective function by using the advantage function A instead of the total rewards R. This modification is rooted in the desire to quantify how much better a specific action is compared to the average action in a particular state. Below, we will delve into the reasoning and implications of this change.

Definition : Adventage Function

The advantage function is defined as:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

where $Q(s_t, a_t)$ is the action-value function and $V(s_t)$ is the value function. The action-value function represents the expected return of taking action a_t in state s_t and following policy π thereafter, while the value function represents the expected return of following policy π from state s_t.

Using the advantage function instead of the total rewards aims to achieve the following:

- **Relative Measurement:** The advantage function measures the relative benefit of an action compared to the average action in a specific state. This encourages the policy to focus on the specific quality of actions rather than the absolute rewards.
- **Stabilizing Training:** By centering the rewards around the value function, the advantage function can reduce the variance in the policy gradient estimates, potentially leading to more stable and efficient training.
- **Enhanced Exploration:** By quantifying how much better an action is relative to others, the advantage function can guide the policy to explore more advantageous actions, improving the overall decision-making process.

In PPO, the objective function changes from using G with total rewards to using the advantage function A. The gradient of the objective function becomes:

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

This change emphasizes learning actions that are advantageous relative to the average action in each state, aligning with the principles and goals of policy optimization.

2.5.2 Temporal Difference Learning

Temporal Difference (TD) Learning is a model-free method used in reinforcement learning to estimate the value function of a policy, V^π, or the action-value function, Q^π.

The action-value function for a given policy π is defined as:

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi, s_t=s, a_t=a}\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\right]$$

Using the Bellman equation, we can rewrite $Q^\pi(s,a)$ in a recursive form:

$$Q^\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left(r + \gamma\sum_{a'}\pi(a'|s')Q^\pi(s',a')\right)$$

However, this equation involves an expectation over all possible next states and rewards, which is often intractable to compute.

To overcome this issue, we can consider only the one next state that was actually observed at each step, effectively taking an individual sample from the distribution. This allows us to rewrite the equation as:

$$Q^\pi(s,a) = r + \gamma\mathbb{E}_{a'\sim\pi(s')}\left[Q^\pi(s',a')\right]$$

Here, we replace the outer sum with the expectation over the action following the policy π, and the value inside the expectation becomes the sample that was actually observed. This forms the basis for the Temporal Difference update.

The TD error, which represents the difference between successive estimates, can be defined as:

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

TD Learning uses this error to update the estimate iteratively. By considering only the sample that actually occurred, TD Learning provides a way to estimate the action-value function with high variance but without the need for modeling the entire distribution over next states.

2.5.3 Generalized Advantage Estimation (GAE)

Generalized Advantage Estimation (GAE) provides a way to balance bias and variance when estimating the advantage function. The advantage function is defined as the difference between the action-value function Q^π and the state-value function V^π:

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

GAE introduces a hyperparameter $\lambda \in [0, 1]$ and defines the advantage estimation as:

$$A^{\pi}_{\text{GAE}}(s_t, a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$ is the Temporal Difference (TD) error.

Understanding λ in GAE

The parameter λ controls the trade-off between bias and variance in the advantage estimation:

- When $\lambda = 0$, the advantage estimation relies only on the immediate TD error, leading to high bias and low variance.
- When $\lambda = 1$, the advantage estimation relies on an infinite sum of future TD errors, leading to low bias and high variance.

The Temporal Difference Error in GAE

The TD error δ_t in GAE plays a crucial role in determining the contribution of each step in the trajectory to the advantage estimation:

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

Here, r_t is the reward at time t, and V^{π} is the value function under policy π. This error represents the difference between the estimated value of the next state and the current state, adjusted by the reward received.

Computing GAE

The computation of A^{π}_{GAE} involves iterating through a trajectory and applying the above formula to calculate the generalized advantage for each state-action pair. The resulting advantage estimates can be used in various policy gradient methods to optimize the policy.

GAE provides a flexible method to estimate the advantage function by introducing a trade-off between bias and variance. By carefully selecting the hyperparameter λ, it's possible to obtain an advantage estimation that suits the specific needs of the task and algorithm.

Chapter 3

Exploring Multi-Agent Reinforcement Learning

3.1 Decentralized Multi Agent MDP Model

3.1.1 Multi-Agent Markov Decision Process (MMDP)

Multi-Agent Markov Decision Process (MMDP) extends the standard Markov Decision Process (MDP) to accommodate multiple agent systems. Unlike a standard MDP where there is only a single agent making decisions, the MMDP framework recognizes that in many realistic scenarios, more than one agent may be present, each making their own decisions.

MMDP can be seen as an extension of the traditional MDP, encapsulating the complexities that arise when multiple agents interact within an environment.

MMDP opens the door to various applications including distributed robotics, traffic control, energy management, and collaborative filtering. This chapter aims to delve into the principles of MMDP, its mathematical formulation, solution methods, and practical applications, providing insights into how multiple agents learn and adapt through interaction with each other and the environment.

An MMDP characterizes a multi-agent environment using the tuple $(S, A_1, A_2, \ldots, A_n, P_{\vec{a}}, R_{\vec{a}})$:

- S is **The Set of States**. It represents the environment in which the agents interact.
- $A_1, A_2, \ldots, A_n$ are **The Sets of Actions** for agents 1 through n, respectively. For any given state $s \in S$, there exists a subset of actions $A_i(s) \subset A_i$ available to agent i.
- $P_{\vec{a}}(s', s)$: **The Transition Function** captures the transition dynamics of the environment for a given joint action $\vec{a}$. The transition function is defined as:

  $$P_{\vec{a}}(s'|s) = \Pr(s_{t+1} = s'|s_t = s, \vec{a}_t = \vec{a})$$

  It represents the probability of transitioning to state s' in the next time step when the joint action $\vec{a}$ is taken by all agents in state s at the current time step.

- $R_{\vec{a}}(s, s')$: **The Joint Reward Function** provides the expected reward vector for all agents after transitioning from state s to state s' due to the joint action $\vec{a}$.

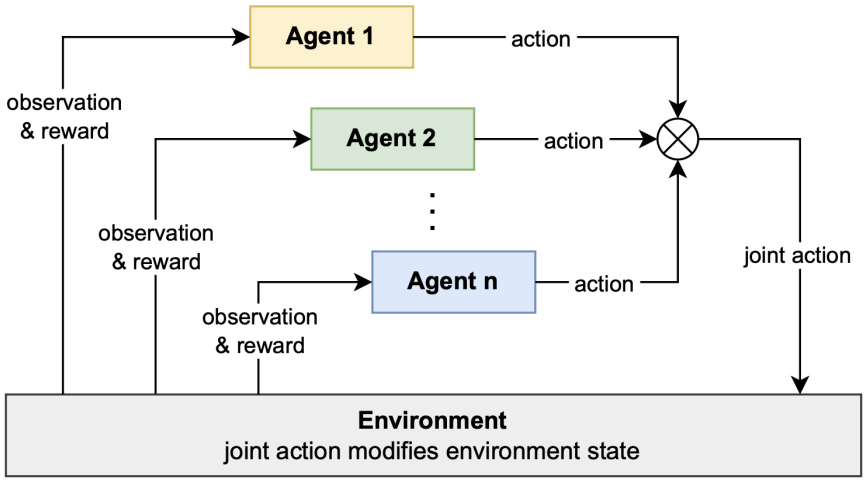


Figure 3.1: Joint Action scheme for Mutli-Agent MDP [1]

In an MMDP, each agent must consider not only its own actions but also the actions of other agents. This adds complexity to the problem, necessitating sophisticated methods for learning and decision-making.

In the context of MMDPs, the notation $\vec{a}$ is used to denote a tuple of actions taken by all the agents in the system. This can be expressed as $\vec{a} = (a_1, a_2, \ldots, a_n)$, where a_i is the action taken by agent i.

3.1.2 Partial Observability: The Dec-POMDP Model

In many multi-agent settings, agents do not have a full view of the global state of the environment. Instead, they only have access to their local observations. This scenario is best modeled using the Decentralized Partially Observable Markov Decision Process (Dec-POMDP).

A Dec-POMDP extends the MMDP model by incorporating local observations for each agent. Formally, a Dec-POMDP is defined by the tuple:

$$\textbf{Dec-POMDP} = \langle S, \{A_i\}_{i=1,\ldots,N}, O, P_T, R, N \rangle$$

Where:

- S is the global state space.
- $\{A_i\}_{i=1,\ldots,N}$ denotes the action set for each agent $i \in N$.
- O is the observation space, and $o_i \in O$ represents the local observation for agent i at global state s.
- P_T is the transition probability function.

- R is the reward function.
- N represents the set of agents.

In a Dec-POMDP, each agent chooses its action based on its local observation and policy. However, the global state transitions and rewards are influenced by the joint actions of all agents. This model captures the challenges of decentralized decision-making under uncertainty and partial observability.

An agent's observation, o_i, can be based on its immediate surroundings in the environment. Intuitively, this means that an agent perceives only a subset of the global state space that is within its vicinity or 'reach'. Mathematically, this can be represented using the concept of distance in the Euclidean space.

For an agent located at position p_i in a 2-dimensional space, the distance to any point p_j in that space can be computed using the Euclidean norm:

$$\text{distance}(p_i, p_j) = \|p_i - p_j\|_2 = \sqrt{(p_{i1} - p_{j1})^2 + (p_{i2} - p_{j2})^2}$$

Given a predefined observation radius, r, for agent i, the set of all states s that fall within this radius form the partial observation o_i. Formally:

$$o_i = \{s \in S \mid dist(i, s) \leq r\}$$

This formulation captures the idea of partial observability, where an agent perceives only a local subset of the global state space. However, in cases where an agent has full observability, the observation o_i coincides with the entire state space, i.e., $o_i = S$ for all i.

3.2 Asynchronous Independent Approach for Multi-Agent Planning

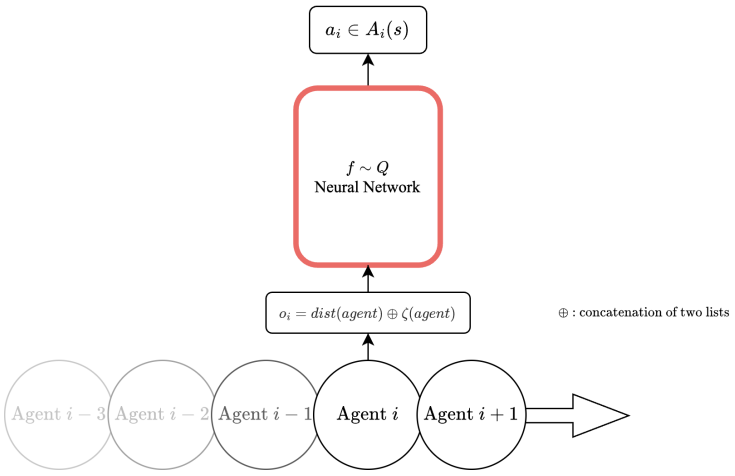


Figure 3.2: Asynchronous Approach

In multi-agent systems, diverse approaches can be taken to coordinate agents and execute tasks. One such method is the asynchronous independent approach, where agents operate based on individual observations but contribute collectively to a shared policy.

3.2.1 Individualized Actions with Shared Policy

A unique feature of this approach is the use of a common policy, typically implemented as a neural network, which guides the actions of every agent in the system. However, rather than a uniform application of this policy, each agent takes actions sequentially, guided by its localized observations. The essential idea can be summarized as:

> While every agent shares the same policy, their actions are determined individually, rooted in their unique observations and perspectives of the environment.

3.2.2 Local Observations with Global Contribution

Each agent's observation, in this scenario, is inherently linked to its vantage point within the environment. These observations are crucial as they provide the agent-specific data needed to decide the subsequent steps based on the shared policy. Though actions are taken independently, the collective experiences of all agents are harnessed to improve and refine the shared policy. This is fundamentally an *experience aggregation* mechanism where:

> Each agent, while operating independently, contributes to the shared policy's improvement by pooling their individual experiences.

3.2.3 Benefits of the Asynchronous Independent Approach

This strategy offers a balance between individual agent autonomy and collective learning. Agents are not restrained by waiting for synchronized actions from other agents, thereby enhancing system responsiveness. Additionally, by consolidating experiences from multiple agents, the shared policy continually evolves, benefiting from diverse experiences and promoting system-wide improvements.

In conclusion, the asynchronous independent approach for multi-agent planning provides a robust framework where individual agents can operate based on localized observations, yet collectively enhance a shared strategy through combined experiences.

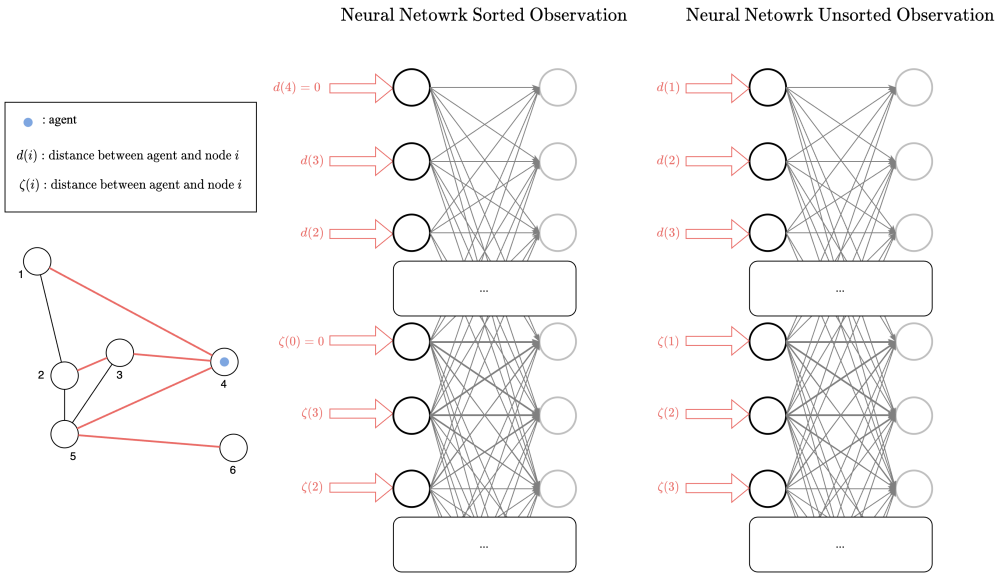
A critical component in this approach is to accurately engineer the agents' observation scheme. This is fundamental as each agent's observations inform their decisions and actions within the environment.

For an agent indexed by i, let's define its observation o_i as:

$$o_i = dist(i) \oplus \zeta(i)$$

where $\oplus$ represents the concatenation of two lists.

Now, considering A as the set of agents, $dist(i)$ denotes a list of distances from node i to every nodes in the environment. Let's denote the set of nodes as V, with v_j representing the j^{th} node. The list $dist(i)$ is constructed by calculating the distance from agent i to every



node and then sorting this list in ascending order. Thus, the ordered list $dist(i)$ can be written as:

$$dist(i) = \langle d(i, v_1), d(i, v_2), \ldots, d(i, v_n) \mid d(i, v_i) \leq d(i, v_{i+1}) \rangle$$

where $n = |V|$ is the total number of node.

The idle time list $\zeta(i)$ captures the idle times for each node ordered from the closest to the farest of i correspondingly to $dist(i)$. If $\zeta(v_j)$ denotes the idle time for agent v_j, the list $\zeta(i)$ can be expressed as:

$$\zeta(i) = \langle t(v_1), t(v_2), \ldots, t(v_n) \mid d(i, v_i) \leq d(i, v_{i+1}) \rangle$$

However, since $dist(i)$ is ordered based on proximity, we need to rearrange $\zeta(i)$ to ensure its order matches that of $dist(i)$. Essentially, the idle time of the agent closest to node i (i.e., the first entry in $dist(i)$) will be the first entry in $\zeta(i)$, and so on.

By integrating both the distance and idle time information in a structured manner, the agent receives a comprehensive view of its environment, allowing for more informed decision-making. This ordering ensures that the lists begin with the closest (or most recently visited) nodes, potentially aiding the policy in understanding the spatial relationships more intuitively.

3.3 Centralized Global Approach for Multi-Agent Planning

3.3.1 Overview

Unlike the asynchronous approach, where each agent observes and acts independently and asynchronously, the synchronous or centralized model operates under the assumption that

a single timestep updates all agents. In this model, every agent shares the same global observation, thereby achieving full observability, represented mathematically as $o_t = s_t$.

$$\pi : \mathcal{S} \rightarrow \mathcal{A}^N$$

Here, π denotes the global policy, a function that maps from the state space $\mathcal{S}$ to a super action in $\mathcal{A}^N$, where N represents the number of agents.

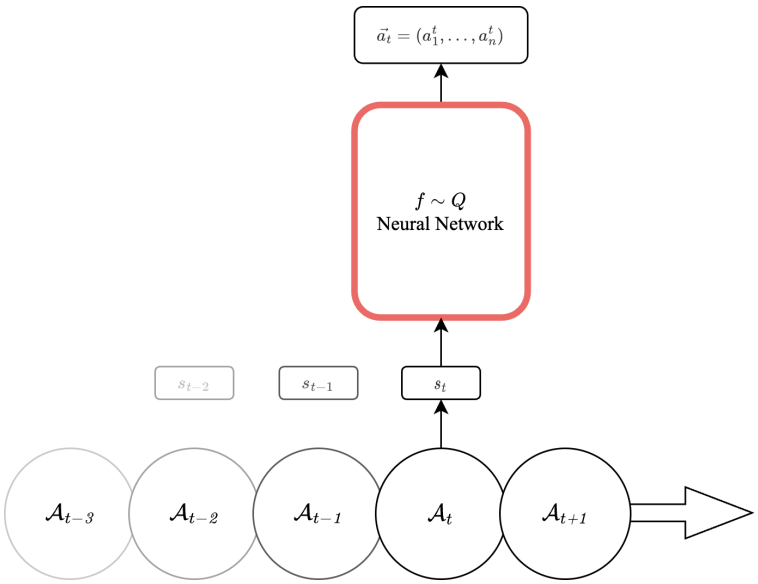


Figure 3.3: Centralized Approach

3.3.2 Constructing the Multi-Action Policy

The main challenge in a centralized approach is the construction of a multi-action policy that can output a super action, incorporating all the individual actions of the agents. This super action is necessary to have a global policy that can effectively coordinate the actions of multiple agents.

Neural Network Output Layer

The first step in this process is to design the neural network architecture appropriately. The output layer of the neural network should be designed to accommodate the actions of all agents. Specifically, the output layer should have a shape of (number of agents) $\times$ (number of nodes), which can be denoted mathematically as:

$$\text{size of output layer} = N \times |V|$$

Splitting Super Action into Individual Actions

Once the neural network processes the input state and generates the super action, it becomes necessary to split this super action into individual actions for each agent. This process

involves segmenting the output of the neural network into groups of the same size, where each group corresponds to an individual agent. Specifically, for agent a_i:

$\mathbf{y}_{(i-1)|V|+1:i|V|}$ denotes the segment of the output vector $\mathbf{y}$ corresponding to agent a_i.

Action Probability Distribution

After segmenting the super action into individual actions, the next step is to convert these segmented values into probabilities. This is typically achieved by passing the segmented values through a softmax layer:

$$p(a_i|s) = \frac{e^{\mathbf{y}_{(i-1)|V|+1:i|V|}}}{\sum_{j=1}^{|V|} e^{\mathbf{y}_{(i-1)|V|+j}}}$$

This equation represents the probability distribution of the actions for agent a_i. Given this probability distribution, an action for each agent can be sampled.

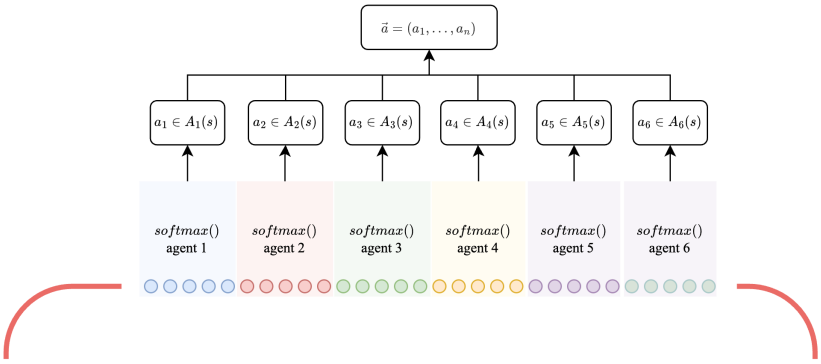


Figure 3.4: Global Policy Neural Network output

Conclusion

The centralized global approach for multi-agent planning provides a coherent and coordinated mechanism for planning the actions of multiple agents. By adopting a global policy that outputs a super action, and then subsequently splitting this super action into individual actions for each agent, a centralized control mechanism is established. However,

this approach also introduces increased complexity in terms of action extraction and back-propagation. It necessitates careful engineering of the neural network architecture and a thoughtful mechanism for extracting individual agent actions from the super action. Despite these challenges, a centralized approach offers the potential for more coordinated and efficient multi-agent planning.

Chapter 4

Challenges that arise in a Multi-Agent Patrolling Problem

4.1 Reward Sparsity

4.1.1 Agent Flip-Flop

The problem of patrolling agents in a graph presents interesting challenges when the agents' policies are not yet trained. When an agent is asked to move randomly to a node while it is on an edge, it exhibits behavior analogous to a *random walk*.

To delve deeper, consider an agent on the edge between two nodes. Its decision to move in one direction or the other is probabilistic. If the agent's policy is not properly trained, it will tend to flip-flop between the two nodes without making significant progress. This behavior is similar to a classic random walk on the number line.

4.1.2 Random Walk Approach

The *random walk* can be thought of as a stochastic process where a walker starts at a position, say 0, and at each step, moves either $+1$ or -1 with equal probability. Formally, if we denote X_n as the position of the walker after n steps, then:

$$X_n = X_{n-1} + Z_n$$

Where Z_n is a random variable taking values $+1$ or -1 with $P(Z_n = 1) = P(Z_n = -1) = 0.5$.

As the walker takes more steps, the expected value of its position remains 0:

$$E[X_n] = E[X_{n-1} + Z_n] = E[X_{n-1}] + E[Z_n] = 0$$

However, the variance increases with time:

$$\text{Var}(X_n) = n$$

Given a random walk defined by $X_n = X_{n-1} + Z_n$, where Z_n is a random variable with values $+1$ or -1 each with probability 0.5, we can compute the expectation and variance of

Z_n as:

$$E[Z_n] = 0.5(1) + 0.5(-1) = 0,$$
$$\text{Var}(Z_n) = E[Z_n^2] - (E[Z_n])^2 = 1.$$

For independent random variables, the variance of their sum is the sum of their variances:

$$\text{Var}(X_n + Y_n) = \text{Var}(X_n) + \text{Var}(Y_n).$$

Thus, for our random walk:

$$\text{Var}(X_n) = \text{Var}(X_{n-1} + Z_n) = \text{Var}(X_{n-1}) + \text{Var}(Z_n) = \text{Var}(X_{n-1}) + 1.$$

Starting with the base case for $n = 1$, where $\text{Var}(X_0) = 0$ (as the starting point is deterministic):

$$\text{Var}(X_1) = \text{Var}(X_0) + 1 = 1,$$
$$\text{Var}(X_2) = \text{Var}(X_1) + 1 = 2,$$
$$\text{Var}(X_3) = \text{Var}(X_2) + 1 = 3.$$

Following this pattern iteratively, it becomes evident that:

$$\text{Var}(X_n) = n.$$

□

Thus, the walker's position will diverge as time progresses, even though its expected position remains zero.

Connection with the Patrolling Problem

Consider a patrolling agent navigating a graph. When placed on an edge connecting two nodes and allowed to move without a predetermined strategy, its actions can be modeled as a random walk. Let the agent's position at time t be denoted by P_t, with the edge endpoints being at positions 0 and n. Initially, $P_0 = \frac{n}{2}$, representing the middle of the edge.

The agent's transitions can be described as:

$$P_{t+1} = P_t + Z_t,$$

where Z_t is a random variable taking values ± 1 with equal probability.

A primary concern in reinforcement learning is how the agent learns from rewards. In the patrolling scenario, rewards might be infrequent and only be received when the agent reaches one of the nodes. This infrequency leads to challenges in learning. The expected cumulative reward at time t is:

$$E[G_t] = E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right],$$

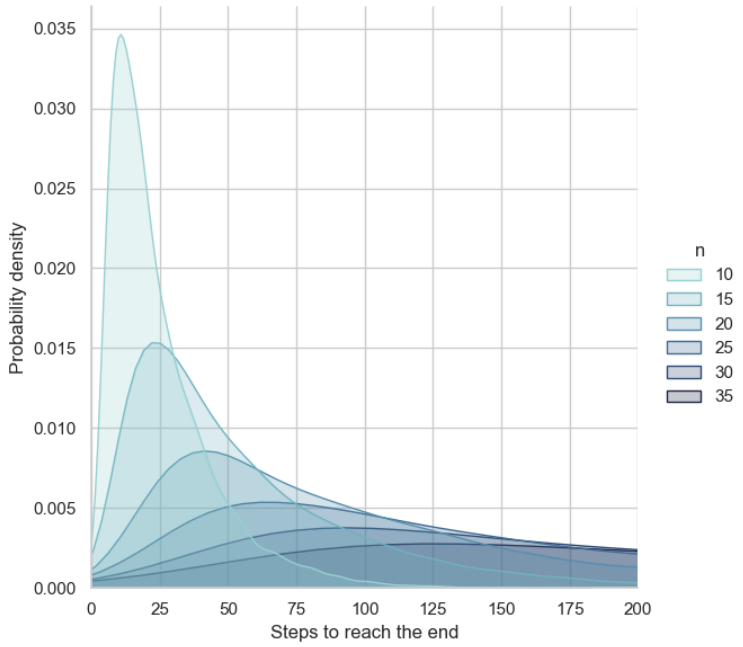


Figure 4.1: Random walk simulation on various edge lengths.

where $0 \leq \gamma < 1$ is the discount factor and R_{t+k} is the reward received at time $t + k$.

Due to the sparse nature of rewards in this problem:

$$E[R_{t+k}] \approx 0 \text{ for many values of } k.$$

Thus, for a large number of steps, the agent essentially receives no feedback, making $E[G_t]$ close to zero. This sparsity complicates the agent's learning since it doesn't have consistent feedback to adjust its behavior. To mitigate this, one could explore techniques like reward shaping, curriculum learning, or advanced exploration mechanisms to guide the agent towards more frequent reward situations and accelerate learning.

In our simulation involving a sample size of 10,000, we observed interesting behavior of the patrolling agent based on the edge lengths. When the agent is situated on an edge of size $n = 10$, equidistant from the two nodes, and moves at a speed of 1 unit per step, it takes on average about 15 steps to reach one of the vertices and subsequently receive a reward. However, the scenario changes notably for an edge of size $n = 25$. The possibility of acquiring a quick reward becomes less predictable. This is evident from the broader and less left-skewed distribution in the simulation, indicating a more extended range of steps the agent might take before reaching a vertex.

Another intuitive solution to improve the agent's receipt of rewards is to increase its speed. However, while this adjustment might expedite learning, it could render the simulation less realistic, potentially sacrificing the practical applicability of our model. We observed that upon augmenting the agent's speed, the system did not adapt as intended. Instead, the increased velocity resulted in agents frequently obtaining rewards irrespective of their chosen actions, thereby obfuscating the efficacy of their policies. This phenomenon underscores the

importance of maintaining a balance between facilitating learning and ensuring realistic simulation. To address this, we modified the state-action generation scheme. Instead of allowing an agent to decide its next move halfway through an edge, we now enforce that the agent travels entirely to the subsequent node as dictated by its policy π. Only upon reaching this node does the agent make an observation and determine its ensuing action. We term this mechanism the "Restrained Action Swap".

4.2 Restrained Action Swap (RAS)

4.2.1 Mitigating Reward Sparsity with the Restrained Action Swap Scheme

In reinforcement learning contexts like our patrolling problem, the sparsity of rewards can significantly hinder the learning efficiency of agents. To mitigate this challenge, an innovative approach we called the Restrained Action Swap (RAS) scheme can be adopted.

Traditionally, in environments with multiple agents, the action-selection loop operates iteratively for each agent. If there are three agents in the environment, for instance, the classic action-selection loop would proceed as: (s_1, a_1, r_1), (s_2, a_2, r_2), (s_3, a_3, r_3), (s_1, a_1, r_1), and so forth. Here, s_i, a_i, and r_i represent the state, action, and reward of the i^{th} agent respectively.

The Restrained Action Swap scheme offers a structured departure from this approach. The core idea is the introduction of an agent-action buffer, denoted as Φ. This buffer is defined as:

$$\Phi = \{(i, a) : i \in \{1, ..., N\}, a \in A_i\}$$

Where N is the total number of agents, and A_i represents the action space for the i^{th} agent.

The primary function of Φ is to maintain a record of the designated action for each agent until that agent reaches another node. At the start of an episode or a particular time frame, Φ is populated with actions for each agent. As the environment progresses, it continually references Φ for agent actions, rather than frequently querying the policy. Only when an agent arrives at a node does it get removed from Φ, signaling the policy to provide a fresh action for that agent. This ensures that at any given time, for every agent i in the set N, there exists a pair $(i, \cdot)$ in Φ.

By utilizing the RAS scheme, the frequency of policy queries is reduced, resulting in more stable action sequences and potentially reducing the sparsity of rewards. This offers a balanced approach, ensuring agents follow a consistent path to nodes while still allowing for dynamic action adjustments as the environment evolves.

4.2.2 Action Reward pair mismatch

A foundational change introduced by the RAS scheme involve making modification on the multi-agent looping mechanism. In classical multi-agent systems, the environment would typically loop through agents, consistently prompting them for new actions. This iterative

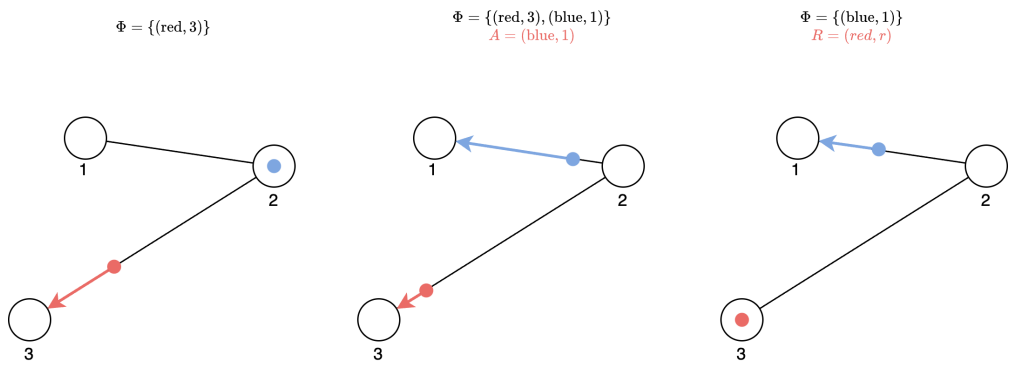


Figure 4.2: Action, Reward pair mismatch

querying could lead to an array of challenges, especially in dynamic environments with complex agent interactions.

Under the RAS scheme, instead of looping through agents, the environment iterates through the Φ buffer. Let t be the iteration index representing time or step. For each agent i that isn't already in the Φ buffer, the action a_i at time t is generated as:

$$a_t^i = \pi(o_t^i)$$

where o_t^i is the observation of agent i at time t. This action is not immediately associated with a reward. Instead, it is stored in the Φ buffer as:

$$\Phi(t) = \Phi(t-1) \cup \{i, a_i(t)\}$$

The agent i will not receive its reward r_i immediately upon the generation of action a_i. Instead, the reward is assigned only once the agent reaches the intended node based on its action. At that juncture, the tuple (i, a_i) is removed from the Φ buffer, and the corresponding reward r_i is determined. This reward, along with the agent and its action, is then added to a replay buffer:

$$\mathcal{D}(t) = \mathcal{D}(t-1) \cup \{o^i, a_i, r_i\}$$

This procedure ensures a robust correlation between the agent's action and the resultant reward, eliminating potential mismatches and fostering an accurate and effective learning process.

This is crucial for a number of reasons. Imagine a scenario where agent 'blue' is instructed to move towards node 1. While 'blue' is en route to its destination, another agent, say 'red', successfully reaches node 3, triggering the receipt of a reward based on its action. In the conventional system, because the environment loops through agents and not actions, the most recent action provided (to 'blue' in this case) might be mistakenly associated with the reward triggered by 'red'. This represents a risk of a mismatch between actions and rewards, which can significantly impede accurate learning.

By looping through Φ, the environment ensures that the action-reward pairs are correctly associated. As the system queries Φ for agent actions and updates it when an agent reaches

a node, the action associated with a specific agent remains consistent until the agent reaches its next node. This avoid the risk of action-reward mismatches and ensures a more robust and reliable learning process.

4.3 Reward Strategies

In the domain of reinforcement learning, one of the pivotal challenges is crafting a reward scheme that accurately reflects the desired objectives of the task at hand. The patrolling problem is no exception. It necessitates an astutely designed reward strategy, ensuring agents patrol in a manner optimizing the metrics of interest. In our scenario, we aim to minimize idle times across the nodes — these can be categorized broadly into two metrics: average idle time and maximum idle time.

Definition: Idle Time Metrics

For a set of vertices V and the idle time denoted by $\zeta(v)$ for a vertex v, the average idle time, $\bar{\zeta}$, is given by:

$$\bar{\zeta} = \frac{1}{M} \sum_{v \in V} \zeta(v)$$

where M represents the number of vertices, i.e., $M = |V|$.
The maximum idle time, $\zeta_{\max}$, is the longest idle time across all vertices and is represented as:

$$\zeta_{\max} = \max_{v \in V} \zeta(v)$$

A fundamental distinction between the patrolling problem and many conventional reinforcement learning environments is the nature of the reward. Classic environments, like the Frozen Lake or the Lunar Lander, provide rewards that are directly aligned with the end objective — for instance, the agent receives a reward upon reaching the goal in Frozen Lake or when the spacecraft successfully lands in Lunar Lander. In contrast, the patrolling problem operates under an infinite horizon T. There isn't a singular 'end goal.' Instead, agents must perpetually make decisions, continuously optimizing to either minimize $\bar{\zeta}$ or $\zeta_{\max}$, thus requiring a more nuanced reward design.

4.3.1 Challenges with Direct Idle Time-based Rewards

While it may appear intuitive to utilize idle times directly in reward computation, such an approach presents significant challenges. Consider an agent reaching a vertex v; a straightforward reward could be the idle time at that vertex.

$$r_t = \zeta(v_t)$$

where r_t is the reward at time t and v_t is the vertex visited at that time.
However, adopting this approach brings forth complications:

1. **Unbounded Reward Interval**: The range for r becomes unbounded, i.e., $r \in \mathbb{R}$. In the context of reinforcement learning, this poses significant issues, particularly for on-policy algorithms that need to estimate expected rewards or the advantage function. Algorithms might struggle to converge due to the unpredictability and vastness of the reward space.

2. **Temporal Reward Dependency**: Utilizing ζ as the basis for reward computation induces a dependency on the episode's temporal progression. That is, the same node with a high idle time ζ could yield different rewards based on when it's visited during an episode. For instance, a vertex identified as having the maximum idle time will not yield consistent rewards if visited early in an episode versus much later, even though the urgency (or significance) of visiting that node remains consistent. This discrepancy can misguide the agent's learning process, as the perceived value of actions becomes inconsistent.

In light of these challenges, it becomes imperative to devise a reward scheme that captures the urgency and significance of visiting nodes, without introducing vast inconsistencies or unbounded reward spaces. It's a delicate balance between ensuring agents recognize priority nodes and fostering stable, consistent learning.

4.3.2 Ranking Approach

A promising strategy to address the aforementioned challenges involves ranking nodes based on their idle times, and using these ranks as rewards. The procedure can be described as follows:

Let V be the set of all vertices in the graph, and $\zeta(v)$ be the idle time for a vertex $v \in V$. The rank of a vertex v, denoted by $\text{rank}(v)$, is defined by its position in the list of vertices sorted in descending order of idle times. Formally, for two distinct vertices $v_1, v_2 \in V$:

$$\forall (v_1, v_2) \in V^2, v_1 \prec v_2 \iff \zeta(v_1) < \zeta(v_2)$$

Here, $\prec$ is the ranking order, and $v_1 < v_2$ is some deterministic order on vertices, employed in case of ties in idle times.

The reward, r, upon reaching vertex v is then:

$$r(v) = \text{rank}(\zeta(v))$$

The merit of this scheme lies in its discretization of the reward space. Instead of allowing rewards to span the entire real line, rewards are now constrained to a finite subset of natural numbers:

$$r \in \{1, 2, \ldots, M\} \quad \text{where} \quad r \in \mathbb{N}$$

This ensures a more predictable and bounded reward space, facilitating more stable learning for agents.

4.3.3 Normalization Approach

Another viable strategy for shaping rewards is the normalization of idle times. This approach scales and shifts the original idle times, thus confining them to a predictable interval.

Given the set of all vertices V and the idle time function $\zeta : V \to \mathbb{R}$, for each vertex $v \in V$, we can compute a normalized reward $r(v)$ as:

$$r(v) = \frac{\zeta(v) - \zeta_{\min}}{\zeta_{\max} - \zeta_{\min}}$$

Where:

$$\zeta_{\min} = \min_{v' \in V} \zeta(v')$$
$$\zeta_{\max} = \max_{v' \in V} \zeta(v')$$

From the definition above, we can observe that the range of r lies within the interval $[0, 1]$:

$$r \in [0, 1] \quad \forall v \in V$$

Normalizing rewards in this manner holds dual benefits. First, it yields a more predictable reward structure that is easier for agents to generalize across different scenarios. Second, by confining rewards to the [0,1] range, we produce gradients that are often better suited for optimization in the context of deep learning architectures, thereby accelerating convergence.

4.3.4 Centered Idle Time Approach

Yet another approach to shaping rewards involves centering the idle times around their mean. This essentially rewards agents for visiting nodes with above-average idle times, while imposing a penalty for nodes with below-average idle times.

Given the set of all vertices V and the idle time function $\zeta : V \to \mathbb{R}$, the reward for a vertex $v \in V$ can be formulated as:

$$r(v) = \zeta(v) - \bar{\zeta}$$

Where:

$$\bar{\zeta} = \frac{1}{|V|} \sum_{v' \in V} \zeta(v')$$

In this scheme, $\bar{\zeta}$ represents the average idle time across all nodes. Consequently:

- If $\zeta(v) > \bar{\zeta}$, then $r(v) > 0$, indicating that agent receives a positive reward for visiting a node with an idle time exceeding the average.
- Conversely, if $\zeta(v) < \bar{\zeta}$, then $r(v) < 0$, meaning that there's a penalty for attending to a node with an idle time below the average.

The advantage of this approach lies in its direct connection with the overall system's average performance. By centering rewards around the mean, we encourage agents to prioritize nodes that deviate significantly from the average, pushing the system towards a balanced patrolling behavior.

4.3.5 Average Idle Time Approach

Another method to devise the reward structure is by using the average idle time, $\bar{\zeta}$, as a global reward.

$$r(v) = \bar{\zeta}$$

This approach, however, has a significant limitation: the reward $r(v)$ is not node-specific. By providing a uniform reward across all vertices, the agent might struggle to discern the effects of its specific actions on individual nodes. Such an approach could severely impede the agent's ability to learn from its mistakes and adjust its strategies on a node-by-node basis.

4.3.6 N-Worst Idle Time Approach

A more selective approach involves focusing on the n-worst idle times. For a given node v:

$$r(v) = \begin{cases} 1 & \text{if } \zeta(v) \text{ is among the } n\text{-worst idle times} \\ 0 & \text{otherwise} \end{cases}$$

The main concern with this approach is the potential for reward sparsity. If an agent only receives rewards for visiting the n-worst idle time nodes, it might rarely encounter rewarding situations, especially in larger environments. This can hamper the learning process and lead to slow or stagnant policy development.

4.3.7 Final Average Idle Time Approach

A last approach is to consider the overall system's performance at the end of a given episode or timeframe. Here, the agent would receive the average idle time, but only at the final step:

$$r_t = \begin{cases} \bar{\zeta} & \text{if } t = T \text{ (last step)} \\ 0 & \text{otherwise} \end{cases}$$

This method, though offering a broader view of the agent's overall performance, suffers from extreme reward sparsity. The agent has to wait until the very end to receive feedback, which can be detrimental for learning, especially in scenarios with long episodes or horizons.

4.4 Observation Strategies

In reinforcement learning, the choice of observation is crucial, as it provides the agent with the information needed to make informed decisions. Through our experiments, we discovered that directly using processed idle times—such as those manipulated for reward calculations—might not always be optimal for observations, especially when those manipulations are the basis for the reward scheme.

4.4.1 Ranking as Observations

Consider the ranking approach, where idle times are translated into rankings for reward calculations. When the agent's observation is also this ranked list:

$$O(v) = \text{rank}(\zeta(v))$$

It might seem redundant or even counterproductive. If the rewards and observations are both derived from the same ranking mechanism, the gradients generated during the backpropagation in the neural network might converge more slowly. The feedback loop could become tautological, where the agent is essentially being told the same information twice, but in slightly different forms. This could lead to less robust learning.

4.4.2 Normalization as Observations

Similarly, if we use the normalization approach for rewards:

$$r(v) = \frac{\zeta(v) - \min_{v' \in V} \zeta(v')}{\max_{v' \in V} \zeta(v') - \min_{v' \in V} \zeta(v')}$$

and also use the normalized idle times as observations:

$$O(v) = r(v)$$

The agent might face a similar challenge. With both the rewards and observations being based on the same normalized values, the gradient updates during training might not provide varied or rich enough information for effective learning.

4.4.3 Centered Idle Times as Observations

When idle times are centered around their average for reward computation:

$$r(v) = \zeta(v) - \bar{\zeta}$$

Using the centered values directly as observations:

$$O(v) = r(v)$$

could again hinder the learning process. The reason is akin to the earlier discussed strategies: redundancy in information could lead to less effective gradient propagation.

Conclusion: While it might be tempting to use processed idle times (like rankings, normalized, or centered values) directly as observations, our experiments suggest that doing so might slow down learning. It's crucial to ensure a diversity of information in both rewards and observations for more efficient policy updates.

4.4.4 Utilizing a Matrix Representation for Agent-Node Affiliation

Another insightful approach for constructing observations is to utilize a matrix representation, capturing the relationship between agents and nodes. This allows an agent to have a clear understanding of the positioning of its counterparts in relation to the graph nodes.

Let's define a matrix M such that:

$$M \in \mathbb{R}^{a \times v}, \quad m_{i,j} \in [0, 1]$$

where the dimensions of M are $n \times p$ with $n = N$ representing the number of agents and $p = |V|$ being the number of nodes. The elements m_{ij} of matrix M would be defined as:

$$m_{ij} = \begin{cases} 1 & \text{if agent } i \text{ is at node } j \\ 0.5 & \text{if agent } i \text{ is equidistant from node } j \text{ and another node } k \\ 0 & \text{otherwise} \end{cases}$$

This representation ensures that if an agent is positioned between two nodes, its affiliation to both nodes is captured as 0.5. Therefore, for an agent a between nodes i and j, we will have:

$$m_{ai} + m_{aj} = 1$$

For illustration, consider a scenario with 3 agents and 4 nodes. If agent 1 is at node 2, agent 2 is between nodes 1 and 3, and agent 3 is at node 4, the matrix M would be:

$$\text{Agents} \left\{ \begin{pmatrix} \overbrace{\begin{matrix} 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}}^{\text{Vertices}} \end{pmatrix} \right.$$

Such a matrix representation offers a compact and informative way to encapsulate the relative positioning of agents with respect to the graph nodes, proving beneficial for learning tasks.

Note that the matrix is flatten as input of the neural netowrk policy π.

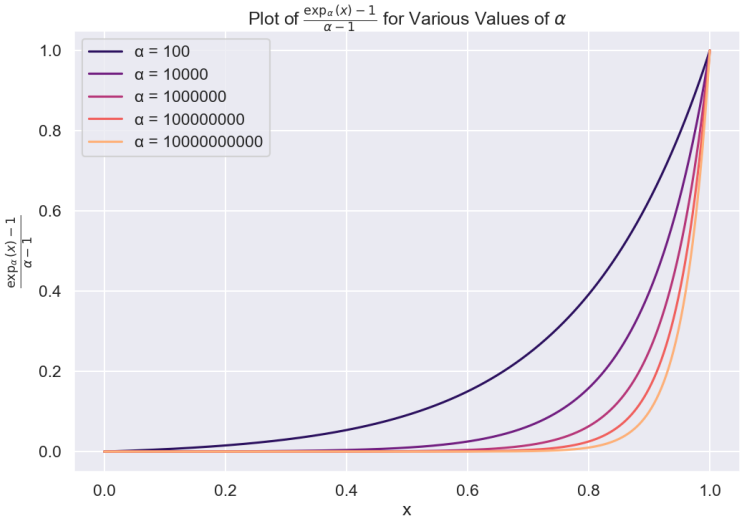
Chapter 5

Numerical Results

5.1 Synchronous Global Approach with PPO

In the first section, we focus on the results obtained using the global synchronous approach. We present two promising results obtained using the normalization method and the ranking method with PPO. It is important to note that in both cases, we transformed the numerical values of the path observations into the range $[0, 1]$ using the function

$$f(x) = \frac{\exp_\alpha x - 1}{\alpha - 1}$$



This transformation ensures that values with high idle times are given higher priority. We tested various values of α, but it seems that there is not much of a difference between $\alpha = 10^5$ and $\alpha = 10^8$. Additionally, as shown in Figure 5.1, there are oscillations in the average idle time. These oscillations occur because the idle time was screened at each time step. The sharp decrease in average idle time is due to the fact that once the environment is reset (this happens at a constant frequency during training) and a new episode starts, the idle time of each node is reset to 0. Therefore, the interesting aspect to observe is the peak,

which represents the average idle time at the end of each episode, just before the environment is reset.

In Figure 5.1, it is observed that both the ranking and normalization methods yield very similar results. Although the ranking method seems to be slightly better, the difference is too minimal to draw any concrete conclusions. This marginal improvement may be attributed to the discrete nature of the ranking method, which could potentially aid the training process.

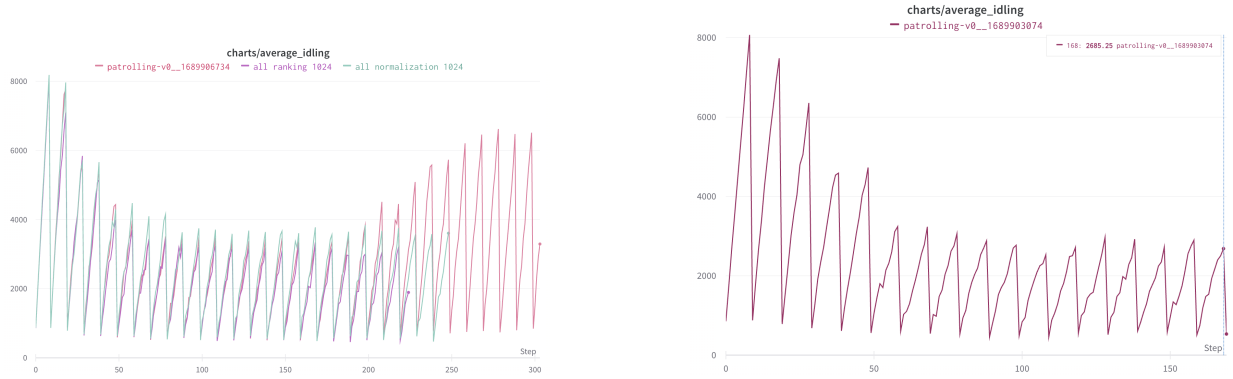


Figure 5.1: Average idle time during training using the synchronous approach on Cumberland (10 agents)

Overall, the training appears to reduce the final idle time by a factor of 2 or 3 on the Cumberland dataset. This is promising, as depicted in Figure 5.2, where the reward is steadily increasing. However, there seems to be a divergence between the average idle time (Figure 5.1) and the reward (Figure 5.2). While the increasing reward indicates that the PPO algorithm is learning from the environment and functioning correctly, it does not align perfectly with our expectation of reducing the idle time. Additionally, on Figure 5.1, we observed that when we tried to run one of the models for more episodes, there was a strong divergence where the average idle time actually got worse over episodes while the reward continued to increase. Unfortunately, the idle time cannot be directly incorporated into the reward scheme due to its unbounded nature in $\mathbb{R}^+$.

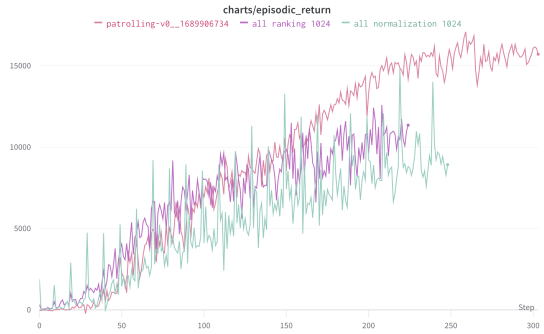


Figure 5.2: Reward during training using the synchronous approach

Additionally, upon observing the agents trained on the Cumberland environment, it was noted that the agents typically ended up patrolling a very limited area in the center of the

graph, as shown in Figure 5.4. This observation is problematic because, even when increasing the value of α, the agents do not seem curious enough to explore the sides of the environment. As a result, the idle time remains high because, although the idle times of the nodes in the center remain very low, the nodes on the extremities are not visited and their idle times increase over time. This observation explains the idle time pattern observed in Figure 5.1.

Upon testing the model on a smaller graph with only 4 nodes, we observed very promising results. The two agents behaved logically, as depicted in Figure 5.3. This observation suggests that the model has a scalability issue when applied to larger, more complex problems.

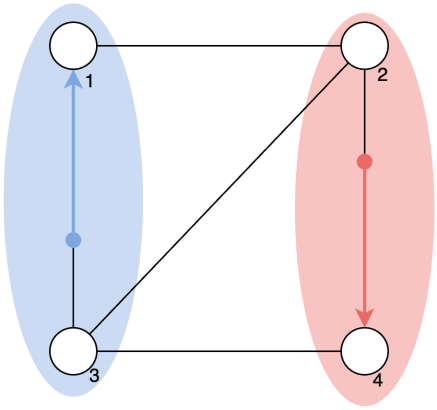


Figure 5.3: Agent behavior on a simple 4-node graph

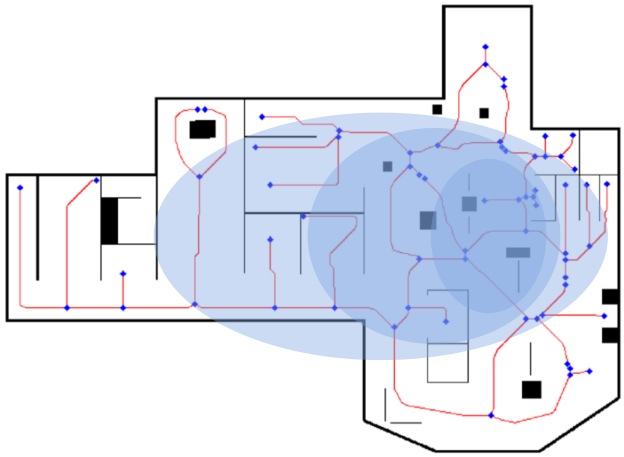


Figure 5.4: Agent traffic on the Cumberland environment

5.2 Asynchronous Individual Approach with DQL

Next, the asynchronous method was applied, but only to the Cumberland environment, as the simple 4-node graph did not pose a significant challenge. The results were quite encouraging, as it was observed that the idle time was reduced by a factor of 4.5, as illustrated in Figure 5.5.

It is important to note that the definition of a 'step' in this context is different from the synchronous case. In the synchronous approach, a 'step' involved all agents acting simultaneously, whereas in the asynchronous approach, a 'step' corresponds to a single action by one agent. Consequently, the duration of each step is actually shorter in the asynchronous case, and a full cycle is completed when all agents have taken a turn. Note that the oscillation is no longer present, in contrast to the synchronous approach, because we only evaluated idle time once at the conclusion of each episode.

Incorporating the restrained action swap, which essentially prevents flip-flopping by requiring the agent to reach the next node before requesting a new action, and addressing the action-reward pair mismatch problem, led to a more rapid decline in the average idle time at the end of each episode. Specifically, the implementation of these adjustments resulted in a nearly fivefold reduction in average idle time compared to random actions.

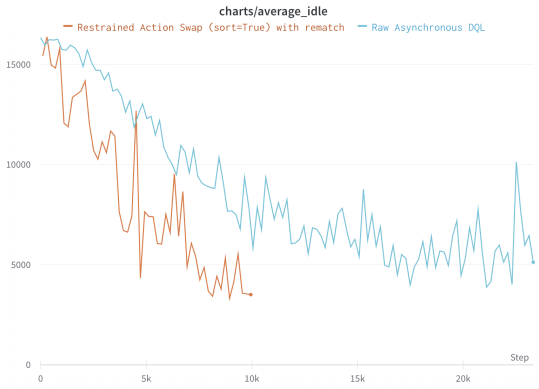


Figure 5.5: Average idle time

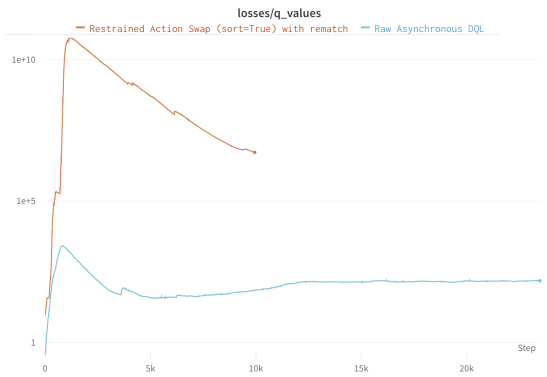


Figure 5.6: Loss

The restrained action swap is a critical modification as it ensures that the agent is committed to reaching the next node once an action has been chosen, thereby preventing frequent changes in direction, known as flip-flopping. This adjustment is essential for creating a more efficient and predictable path for the agents. Additionally, addressing the action-reward pair mismatch problem ensures that the rewards received by the agent accurately reflect the actions taken, which is crucial for the effective training of the agents.

These enhancements collectively contribute to a more efficient and effective patrolling strategy, as evidenced by the significant reduction in average idle time.

5.3 Conclusion

In conclusion, the application of deep reinforcement learning, as implemented in this study, proved to be highly effective in addressing simple problems, such as the 4-node graph. However, scaling up the problem introduced complexities that hindered the achievement of equally promising results. Although the observed reduction in idle time is encouraging, it did not translate to a high-performance level. Several factors could have contributed to this outcome.

Upon observing the agents' post-training performance, it was apparent that challenges, which seemed straightforward from a human perspective, such as flip-flopping, were not as obvious to the agents, leading them to struggle with basic movement and node-reaching tasks. The multi-agent patrolling problem is inherently complex, given the varying distances between nodes and the distinct states of each agent, depending on their location on the graph.

It is unclear whether modifying the observation or reward scheme would yield better results. The limited differences observed when altering the reward scheme (normalization or ranking) suggest that a fundamentally different approach may be necessary to achieve human-like common sense performance more easily. Additionally, simplifications, such as equalizing the distances between nodes, could potentially address some of the challenges, eliminating the need for distance measurement for each neighbor.

Another significant challenge lies in the dependency of the idle time on time and policy performance. Since the ultimate goal is to optimize idle time, it is essential to include it in

the reward scheme. However, this creates instability during training, sometimes resulting in an increase in rewards but not a corresponding decrease in idle time.

In summary, while deep reinforcement learning shows promise in addressing multi-agent patrolling problems, the complexities encountered when scaling up the problem indicate that there is room for improvement and innovation in the approach. Further research is needed to explore alternative methods and simplifications that could lead to more stable and efficient solutions.

One other approach may be to implement the observation as tree based where it would represent all the possible path from the agent. However, inputting a tree into a Neural Network appears more complex because of its variable-shape nature. Another alternative might be employing a Recurrent Neural Network bearing in mind that this would break the MDP principle.

Bibliography

[1] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2023.

[2] Anthony Goeckner, Xinliang Li, Ermin Wei, and Qi Zhu. Attrition-aware adaptation for multi-agent patrolling, 2023.

[3] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

[4] Laura Graesser and Wah Loon Keng. *Foundations of Deep Reinforcement Learning*. Addison-Wesley, 2020.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[6] David Portugal and Rui P. Rocha. Cooperative multi-robot patrol with bayesian learning. *Auton. Robots*, 40(5):929–953, jun 2016.

[7] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, dec 2020.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[9] Rose E. Wang, J. Chase Kew, Dennis Lee, Tsang-Wei Edward Lee, Tingnan Zhang, Brian Ichter, Jie Tan, and Aleksandra Faust. Model-based reinforcement learning for decentralized multiagent rendezvous, 2020.

[10] Hiroshi Yoshitake and Pieter Abbeel. The impact of overall optimization on warehouse automation, 2023.