POLITECNICO DI TORINO

Master's Degree in Mechatronics Engineering
Control Technologies for Industry 4.0

A.A. 2024-2025

April 2025

Creation of an RL Environment to Monitor Ocean Features with Autonomous Vehicles

Supervisors
Prof. Alessandro RIZZO
Dr. Ivan MASMITJA
Dr. Giacomo PICARDI

Candidate
Natalia LOIZZO

Abstract

The ability to detect and monitor dynamic ocean features is crucial for understanding and mitigating environmental impacts. These features, which can include variations in temperature, salinity, biological phenomena, and chemical spills, play a significant role in marine ecosystems and can have substantial ecological and economic consequences. This thesis explores the application of Reinforcement Learning (RL) for monitoring ocean features through the simulation of an autonomous agent operating in a dynamic and uncertain environment. The work builds on a preexisting project, modifying the environment to integrate various ocean features and implementing advanced detection techniques supported by sensors and an optimized reward function to guide the agent's behavior. Two different simulations have been developed and compared, each characterized by a distinct reward function. The first simulation relies on sensors capable of measuring the distance between the agent and the ocean feature, while the second utilizes a sensor that detects only the presence or absence of ocean features in the surrounding area. The obtained results highlight the potential of the RL approach in providing an effective method for detecting and monitoring ocean features, paving the way for future real-world implementations. This study can be applied to various environmental monitoring tasks, such as detecting oil spills, Posidonia meadow, and algal blooms, contributing to advancements in autonomous environmental surveillance systems.

Acknowledgements

I would first like to express my deepest gratitude to my internal supervisor, Professor Alessandro Rizzo, for believing in me and my project, and for giving me the freedom to pursue this work.

A heartfelt thank you to my external supervisors, Ivan Masmitja and Giacomo Picardi, who have been my constant support throughout this journey. Your guidance and encouragement have been invaluable, providing me with the strength to continue, even when I felt discouraged. You placed immense trust in me, and I am grateful for the opportunity to be part of the CSIC and your wonderful, talented, and welcoming team, who made me feel included in every initiative. There has not been a single day when I didn't wake up with a smile, excited to go to the ICM and continue working on my project.

I am deeply grateful to my family for giving me the greatest gift one can receive: total support for my education, always offered with love and without hesitation. Above all, your constant emotional support has been invaluable to me, and I cannot thank you enough for being my rock throughout this journey.

A special thank you to my friends, who, even though they saw me grow distant, have always cheered me on and supported me from afar.

Finally, to Tony, the best man I could ever ask to have by my side. You've always been there to celebrate my successes, encourage me to reach new goals, and give me your strength during my moments of weakness.

I can never thank you all enough for everything.

Alle donne, e in particolare a quelle nelle STEM, con la speranza che un giorno, quando diranno di aver scelto ingegneria, non ci sia più stupore o meraviglia nelle persone.
Un pensiero speciale alle donne che hanno già raggiunto traguardi in questo campo, che con il loro esempio mi hanno ispirata e mostrato che la passione e la determinazione sono la vera forza per superare ogni barriera.
Che ogni passo che compiamo porti più donne a sentirsi libere di seguire le loro passioni, senza limiti o pregiudizi.

Con stima e amore, Nataly

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview of Machine Learning

1.1.1 What is Machine Learning?

Machine Learning (ML) is a field that allows computers to learn from data without explicit programming. Instead of being manually programmed for each task, ML algorithms develop the ability to solve problems through experience, refining their "knowledge" over time. ML algorithms implicitly extract information from data. The goal is to approximate a statistical model using complex functions, such as neural networks. ML is closely linked to pattern recognition, using techniques to recognize elements and structures in data. A fundamental approach is supervised learning, in which an algorithm learns to classify or predict outcomes based on the positive and negative examples provided. The quality and quantity of data are crucial for the success of any ML application [1].

1.1.2 Connection Between Machine Learning and AI

Machine learning is just one component required for a system to qualify as artificial intelligence (AI). The machine learning aspect allows AI to perform the following functions:

- Adjust to unforeseen situations that were not explicitly programmed by the developer.
- Detect patterns in various types of data sources.
- Generate new behaviors based on identified patterns.
- Make choices depending on the effectiveness or failure of certain actions.

The application of algorithms for processing data is central to machine learning. For a machine learning process to be effective, it must use an algorithm suited to achieving the intended goal. Additionally, data must be prepared appropriately for analysis, either by the algorithm itself or by data analysts.

Other AI Disciplines Beyond Machine Learning
To accurately replicate human thought processes, AI incorporates various disciplines. Besides machine learning, AI also typically includes:

- **Natural language processing (NLP)**: conversion of text-based input to a machine-readable format.
- **Natural language understanding**: interpreting human language to perform tasks accordingly.
- **Knowledge representation**: organizing and storing information in a way that allows fast retrieval.
- **Planning (goal-driven problem-solving)**: leveraging stored data to make decisions almost instantly.

As technological advancements continue, the practical implementation of theories (technique) becomes just as crucial as the development of theories (science).

Specialists establish formalized guidelines known as specifications, which serve as a framework for defining AI and ML processes.

Mathematics plays a crucial role in ML as it dictates how large datasets are interpreted. Algorithms process input data to construct models that predict outcomes. However, if no identifiable patterns exist within the data, it becomes impossible to make reliable forecasts [2].

A key aspect of ML is its ability to learn from data without being explicitly programmed. Unlike traditional rule-based systems, ML algorithms improve over time by identifying patterns and adjusting their predictions based on experience. However, conventional ML still relies heavily on manual design and expert intervention. Addressing these limitations requires new approaches that improve the autonomy of learning systems, allowing them to make independent decisions about what and how to learn.

1.1.3 Categories of Machine Learning

One of the key challenges in **AI** is enabling systems to learn from data without requiring explicit programming. Unlike traditional rule-based systems, which rely on manually defined logic, modern **ML** techniques leverage data-driven learning to improve performance over time.

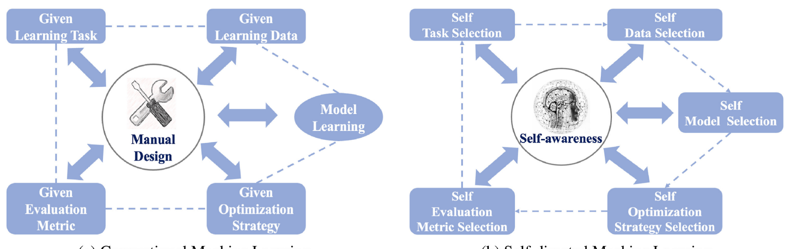
ML paradigms, depending on the type of experience they are allowed to have during the learning process, can be broadly categorized into:

- **Supervised Learning**: models train on data sets where each example is associated with a label or target. This allows the algorithm to learn an explicit mapping between input features and outputs. The term supervised originates from the idea that an instructor provides explicit guidance by showing the system the correct answers. Thus, supervised learning excels in tasks such as classification and regression, where clearly labeled data are available [3].

- **Unsupervised Learning**: deals with data sets that lack predefined labels. Instead of learning a direct mapping from inputs to outputs, the algorithm identifies patterns and structures within the data. Unlike supervised learning, unsupervised approaches do not rely on an instructor, making them more adaptable in scenarios where labeled data is scarce or unavailable [3].

Limitations of Traditional Machine Learning
A major limitation of supervised and unsupervised learning is their dependency on predefined learning setups, where human experts design datasets, models, and evaluation criteria. To **increase machine autonomy**, researchers have explored methods that allow models to self-direct their learning process.

One such paradigm is **Self-Directed Machine Learning (SDML)**, which enables systems to make decisions about what to learn, which data to use, and how to evaluate performance.
SDML draws inspiration from self-directed human learning, where learners set their goals and strategies autonomously based on their experiences. By incorporating self-awareness mechanisms, SDML allows models to dynamically adapt their learning process, moving closer to autonomous decision-making without full human supervision [4].



(a) Conventional Machine Learning (b) Self-directed Machine Learning

Figure 1.1: Comparison of conventional machine learning (a) and self-directed machine learning (b)[4].

Beyond these paradigms, some machine learning models do not rely on a fixed data set, but rather learn through interaction with an environment. This is the case of **Reinforcement Learning** (**RL**), where an agent takes actions within an environment and receives feedback in the form of rewards or penalties. Unlike supervised learning, which requires a predefined dataset with explicit labels, and unsupervised learning, which is limited to passive pattern recognition, **RL** enables a system to **learn dynamically through trial and error**. This ability **to adapt based on experience** rather than predefined instructions makes RL a powerful tool for **decision-making tasks in complex, changing environments**.

1.2 Fundamentals of Reinforcement Learning

One of the most fundamental aspects of AI is the ability to learn from experience. **(RL)** is a machine learning paradigm that relies on this principle, allowing an agent to improve its behavior through **direct interaction with a dynamic environment**. Unlike other ML techniques, where a model learns from labeled data provided by a supervisor, RL does not rely on explicit instructions about which actions are correct or incorrect. Instead, the agent must **autonomously explore, receive feedback from the environment, and adjust its strategy** to maximize a long-term numerical reward [5].

The fundamental idea behind RL is deeply rooted in natural learning processes. Humans and animals learn through trial and error, interacting with their surroundings and adapting their behavior based on the consequences of their actions. For instance, a child does not need explicit instructions to grasp an object; they attempt multiple times, receiving feedback from their body and the environment, until they succeed. Similarly, an RL agent **interacts with the environment without direct supervision**, gradually discovering which decisions yield the most favorable outcomes.

1.2.1 Key Characteristics of Reinforcement Learning

Reinforcement learning is distinguished from other ML approaches by three essential characteristics. First, it is a **closed-loop problem**, meaning that the agent's actions directly influence the future data it will learn from. Unlike supervised learning, where models operate on fixed datasets, RL involves continuous interaction with the environment, making the learning process dynamic and evolving over time.

Another defining characteristic is that the agent does not receive explicit instructions about which actions to take. In supervised learning, each training example includes a correct answer provided by a supervisor, whereas in reinforcement learning, the **agent must independently discover** which decisions are beneficial by experimenting and observing their consequences.

Lastly, a crucial feature of RL is that the agent's actions affect not only the immediate reward but also future rewards. This introduces significant complexity, as the agent must learn to optimize for long-term benefits rather than just short-term gains. A strategy that seems optimal in the short run might lead to suboptimal results in the long run, requiring the agent to carefully balance its decision-making process.

1.2.2 The Concepts of Reward and Value

In reinforcement learning, the **reward** serves as the primary signal guiding the agent's learning process. Each time the agent takes an action, it receives **numerical feedback** indicating the quality of its decision. This reward signal is crucial because it incentivizes behaviors that maximize long-term returns.

However, relying solely on immediate rewards can lead to **myopic decision-making**, where the agent chooses actions that offer short-term gains without considering long-term consequences. To address this, RL introduces the concept of **value**, which represents the expected sum of future rewards that an agent can obtain starting from a given state. In other words, while the **reward** indicates **what is good in the short term**, the **value** expresses **what is advantageous in the long run**. A well-trained agent does not just maximize immediate rewards but also evaluates **how a decision influences the achievement of more complex objectives over time**.

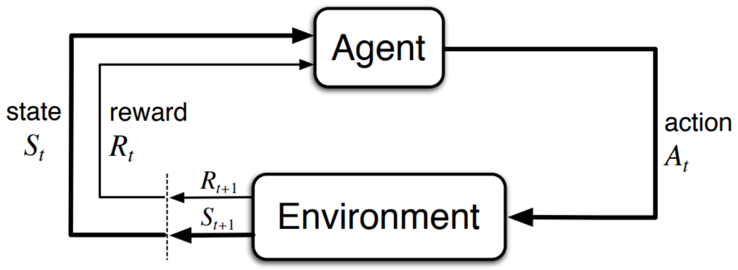


Figure 1.2: The agent–environment interaction in RL [5].

1.2.3 The Exploration-Exploitation Dilemma

A fundamental challenge in RL is the trade-off between exploration and exploitation. To achieve high rewards, the agent should prefer actions that have previously yielded favorable outcomes. However, to discover potentially better strategies, it must also explore untested options.

This creates a delicate balance:

- **Exploitation** refers to selecting actions that, based on past experience, are known to yield the highest reward.
- **Exploration** involves trying new actions to acquire information that may lead to better strategies in the future.

An agent that relies entirely on exploitation risks becoming stuck in suboptimal solutions, never discovering superior alternatives. Conversely, excessive exploration may lead to inefficient learning, as the agent continuously gathers information without applying it effectively to maximize rewards. In stochastic environments, where action outcomes vary, an agent must test a given action multiple times to estimate its expected reward accurately. This trade-off is unique to RL and does not arise in supervised or unsupervised learning paradigms.

1.2.4 A Holistic Approach to Learning

RL is not just a method for solving isolated problems but provides a **comprehensive framework for decision-making** in uncertain and dynamic environments. Unlike many ML techniques that focus on specific subproblems, RL considers an **interactive agent** with explicit goals, capable of perceiving the environment, making decisions, and **continuously adapting to evolving situations**.

This approach is **particularly effective in real-world applications** where explicit supervision is impractical or impossible. For example, in autonomous robotics, complex system optimization, and strategic game playing, RL enables the development of agents that can learn independently without requiring labeled data sets. Additionally, RL explicitly addresses uncertainty, allowing agents to operate even in environments where the consequences of actions are not entirely predictable.

RL represents an advanced learning paradigm that allows agents to autonomously learn from interaction with their environment. By leveraging key concepts such as reward, value, and the balance between exploration and exploitation, RL provides a flexible and powerful approach to solving complex, dynamic problems. Its ability to adapt to unstructured scenarios makes Reinforcement Learning one of the most effective techniques for developing intelligent, autonomous systems.

1.3 Markov Decision Processes (MDPs)

RL problems are commonly modeled using **Markov Decision Processes (MDPs)**, which provide a mathematical framework for decision-making under uncertainty. An MDP defines the interaction between an **agent** and an **environment**, where the agent learns to take actions that maximize cumulative rewards.

1.3.1 Definition and Components

An MDP is defined by the tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma) \tag{1.1}$$

where:

- $\mathcal{S}$: Set of possible **states** of the environment.
- $\mathcal{A}$: Set of **actions** the agent can take.
- $P(s'|s,a)$: **Transition function**, defining the probability of moving to state s' from state s after action a.
- $R(s,a)$: **Reward function**, mapping state-action pairs to numerical rewards.
- $\gamma \in$ [0,1]: **Discount factor**, determining the importance of future rewards.

The **Markov property** ensures that the next state depends only on the current state and action, not on previous states.

1.3.2 Value Functions and Policy Optimization

In RL, the agent follows a **policy** π, which defines the probability of selecting an action in a given state. The **state-value function** and **action-value function** estimate the expected future rewards when following a specific policy.

The **state-value function**, denoted as $v_\pi(s)$, represents the **expected return** when the agent starts in state s and follows policy π (Equation 1.2).

$$v^\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \tag{1.2}$$

Similarly, the **action-value function**, denoted as $q_\pi(s,a)$, represents the **expected return** starting from state s, taking action a, and then following policy π (Equation 1.3).

$$q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \tag{1.3}$$

These functions guide the optimization of an **optimal policy** π^* that maximizes expected rewards.

1.3.3 Bellman Equations and Optimal Policies

The Bellman equations establish a recursive relationship for value functions, providing the foundation for many RL algorithms. Specifically, the **Bellman equation for the state-value function** expresses the expected return in terms of the current reward and the discounted value of future states (Equation 1.4):

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s'} P(s'|s,a)\left[R(s,a) + \gamma v^{\pi}(s')\right] \tag{1.4}$$

The objective of RL is to determine the **optimal policy** π^* that maximizes cumulative rewards over time. The corresponding **optimal state-value function** is defined by the Bellman optimality equation (Equation 1.5):

$$v^*(s) = \max_a \sum_{s'} P(s'|s,a)\left[R(s,a) + \gamma v^*(s')\right] \tag{1.5}$$

These equations form the theoretical backbone for various RL algorithms, including **Q-learning**, **Deep Q-Networks (DQN)**, and **policy gradient methods**, enabling agents to learn optimal strategies through interaction with the environment.

1.3.4 Relevance to This Study

In this work, an MDP framework is used to model the interaction between an autonomous agent and the ocean environment. The agent's goal is to learn a policy that allows it to remain within a dynamically evolving ocean feature, optimizing its trajectory based on reward signals.

This framework enables the development of a RL-based system that can effectively monitor and adapt to changing marine conditions.

1.4 Contributions and Objectives of the Thesis

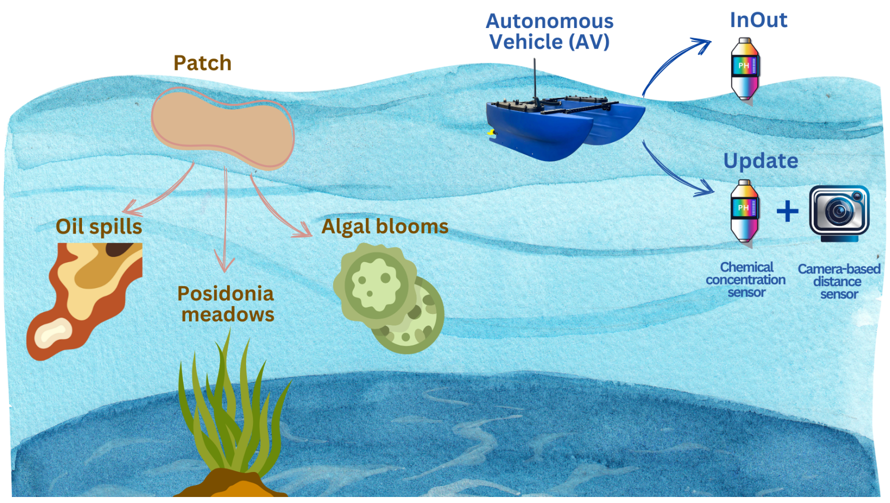


Figure 1.3: Representation of the simulated RL environment for monitoring Ocean Features using an Autonomous Surface Vehicle.

The main objective of this thesis is the development of a RL environment for monitoring oceanic features using autonomous vehicles. This environment provides a structured framework to test different navigation strategies and sensor configurations in dynamic scenarios, where patches of varying shapes and sizes are generated in each simulation, as illustrated in Figure 1.3.

Unlike other works that focus primarily on training an agent to maximize a specific reward function, this research emphasizes the design and validation of a suitable RL environment, identifying key parameters that influence effective agent learning.

The contributions of this work can be summarized as follows:

1. **Development of an RL environment** – Creation of a simulated framework for monitoring oceanic phenomena, incorporating a patch model with varying shape and size across simulations.

2. **Analysis of agent observations** – Evaluation of different observation space configurations to determine which information is most relevant for the agent's learning process.

3. **Experimentation with reward functions** – Implementation and testing of various reward functions to assess their effectiveness in guiding the agent's behavior.

4. **Comparative analysis of sensor configurations** – Investigation of different sensor setups to evaluate their impact on the agent's ability to detect and monitor oceanic features.

5. **Definition of future research directions** – Identification of key areas for improvement, including the potential integration of additional environmental factors.

The insights gained from the preliminary simulations provide a strong foundation for further development, paving the way for more advanced RL-based strategies in ocean feature monitoring.

Chapter 2

Background and Related Works

2.1 Environmental Robotics and Monitoring

Environmental monitoring is an essential field encompassing diverse applications, including marine exploration, wildlife conservation, ecosystem assessment, and air quality monitoring. The ability to collect accurate and timely data from remote or hazardous locations is crucial for understanding and addressing environmental challenges. Traditional methods often rely on manual sampling, satellite imaging, or static sensor networks, which may be limited in resolution, coverage, or real-time adaptability [6, 7, 8].

In recent years, **environmental robotics** has emerged as a transformative solution to these challenges. By integrating autonomy and artificial intelligence, robotic platforms can be deployed for extended periods in dynamic and inaccessible environments, significantly enhancing data collection capabilities. These systems leverage advanced decision-making algorithms to optimize their sensing strategies, enabling efficient and adaptive monitoring of environmental phenomena [9].

The increasing need for autonomous sensing and environmental monitoring has driven significant research efforts toward developing robotic platforms capable of tracking, analyzing, and responding to dynamic environmental features. A crucial area of study within this field is the development of autonomous agents capable of detecting and tracking dispersed substances in complex and time-varying conditions.

In this context, three key works have laid the foundation for this study, each contributing fundamental insights that shaped the design and methodology of this research:

- The first study investigates how flying insects track odour plumes by relying on olfactory and mechanosensory cues, demonstrating how artificial agents can leverage biologically inspired strategies for plume localization [10]. The behavioural patterns observed in this work provide valuable inspiration for the design of autonomous robots that must operate in turbulent flow environments. (See Figure 2.1).

- The second study focuses on chemical plume tracking (CPT) in marine robotics, specifically exploring how autonomous underwater vehicles (AUVs) can detect hydrothermal vent emissions using a combination of probabilistic models and deep RL [11]. This work is highly relevant to the challenges of tracking chemical spills in oceanic environments, which share key similarities with hydrothermal plume dispersion. (See Figure 2.2).
- The third work serves as the baseline system for this research, introducing a range-only underwater target tracking framework that was later adapted and modified for ocean features monitoring [12]. While originally designed for tracking a mobile target, the methodology was extended in this study to accommodate static ocean features, requiring a shift in sensing modalities and agent behavior. (See Figure 3.1).

Each of these works contributes distinct yet complementary elements to the problem of autonomous environmental feature localization, reinforcing the need for robust sensing strategies that allow robotic agents to adapt to uncertain and dynamic conditions.

2.2 Plume Tracking in Environmental Robotics

Among the various applications of environmental robotics, **plume tracking** represents a key challenge, particularly in scenarios where an agent must locate the source of a dispersed substance in a dynamic medium, such as air or water. Plume tracking has been extensively studied in both aerial and underwater environments, leading to different approaches tailored to each domain.

2.2.1 Odour Plume Tracking in Environmental Robotics

Plume tracking is a crucial capability for autonomous agents operating in environments where substances disperse in a dynamic medium, such as air or water. One of the most studied cases of plume tracking is related to airborne odour plumes, which are widely explored in the context of insect flight behavior and bio-inspired robotics [13, 14, 15].

Many flying insects rely on odor plumes to locate sources of food, mates, or oviposition sites. The strategies they employ depend on multiple factors, including the **spatial scale** and **visibility** of the odor source. At greater distances, or when the source is obscured, the search process is primarily guided by **olfactory and mechanosensory cues**, which allow the insect to estimate wind velocity and direction. In this regime, stereotyped behavioral sequences play a crucial role: insects tend to surge **upwind** when they detect the odor and perform **crosswind or U-turn maneuvers** when they lose contact with the plume, helping them to relocate the odor trail [16, 17].

An example of this behavior is shown in Figure 2.1, where a robotic agent tracks an odor source using a bio-inspired approach. The robot follows a zigzagging trajectory influenced by the wind direction, mimicking the search patterns observed in insects [18]. The probability distribution of the source location is continuously updated as the robot collects new sensor measurements.

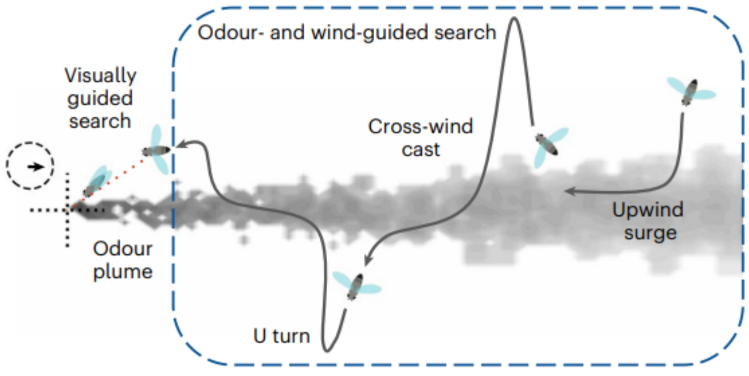


Figure 2.1: Example of an odor plume tracking scenario. The robot starts from a distant position and follows the odor plume to locate the source, adapting its trajectory based on concentration measurements and wind conditions [15].

Recent studies suggest that **agents track plumes based on their local shape rather than relying purely on wind direction**. While some models propose explicit upwind surges near the plume centerline [13], other analyses do not fully support this hypothesis [19]. This discrepancy highlights the importance of investigating how local plume dynamics influence tracking behavior in non-stationary wind conditions.

Deep reinforcement learning (RL) and recurrent neural networks (RNNs) have been employed to train artificial agents for autonomous plume tracking. Unlike traditional approaches that rely solely on instantaneous sensor readings, these networks enable agents to **process and retain past sensory information**, allowing for more adaptive and effective decision-making [14, 20].

Specifically, the neural networks learn to encode and utilize key task-related variables that extend beyond immediate egocentric observations. These representations include:

- **Agent's heading direction:** The agent continuously tracks its orientation to determine how to navigate relative to the plume source [15].
- **Elapsed time since last plume encounter:** If the agent has not detected the plume for a certain duration, it must adjust its search strategy to relocate it [19].
- **Exponentially weighted moving average of detected odor concentrations:** Rather than relying on a single sensor reading, the agent maintains a weighted average of past odor detections to assess the overall trend of the plume [19].
- **Binary signal tracking odor encounters over time:** The agent records whether it has detected the plume in recent time steps, helping it decide whether to persist in its current direction or adopt a new search strategy [19].

By integrating these learned representations, the agent does not merely react to instantaneous sensory inputs but instead **leverages memory of past encounters** to optimize its plume-tracking behavior, similar to how biological organisms follow odor cues in dynamic environments.

Notably, the activity of these networks is **low-dimensional**, meaning that the complex neural responses can be described using only a few key variables, rather than requiring an extremely large number of parameters [21, 22].

The way the network's activity evolves over time follows **distinct patterns associated with different behaviors**:

- When the agent **loses track of the plume** and starts searching randomly, its neural activity follows **quasiperiodic limit cycles**, meaning that the agent's behavior repeats in a structured, cyclical manner.
- When the agent **successfully tracks the plume**, the network activity forms **attractor-like structures** that guide the agent toward the source, effectively shaping its trajectory in a predictable way [23, 24].

These insights reinforce the idea that memory is critical for plume tracking, particularly in non-stationary wind environments, where recent sensory history informs decision-making [25]. However, in steady-wind conditions, short-term reflexive mechanisms (lasting <0.5 s) may suffice for successful tracking [19].

This research has direct implications for robotic applications, as bio-inspired algorithms can enhance plume-tracking efficiency in complex environments. By leveraging memory-driven strategies and encoding sensory history, artificial agents can improve their robustness in real-world conditions, such as gas leak detection, environmental monitoring, and search-and-rescue operations [26, 27].

In the next section, the application of similar principles to the tracking of chemical plumes in underwater environments is examined, focusing on how an autonomous agent identifies hydrothermal vent sources using chemical concentration measurements.

2.2.2 Chemical Plume Tracking in Marine Robotics

In underwater environments, **chemical plume tracking (CPT)** is essential for detecting hydrothermal vents, monitoring oil spills, and studying underwater chemical emissions. Autonomous underwater vehicles (AUVs) equipped with chemical sensors have been widely employed for this task, using a combination of bio-inspired behaviors and probabilistic models to navigate toward the source. Traditional CPT methods can be broadly categorized into two main approaches. Bio-inspired strategies mimic the olfactory search behaviors observed in animals, such as the *surge-and-cast* technique seen in moths, where the agent moves up-current when detecting a chemical and performs lateral sweeps when the signal is lost [16, 18].

On the other hand, probabilistic models estimate the likely position of the source based on a mathematical representation of plume dispersion, often employing Bayesian inference to refine estimations over time [28, 26].

While these techniques have been successfully applied in controlled conditions, they exhibit **notable limitations in real-world oceanic environments**. Bio-inspired methods struggle in turbulent flow conditions, where plumes become fragmented and intermittent, making simple chemotaxis-based approaches unreliable. Probabilistic models, though more robust, require significant computational resources and can be challenging to implement in real-time tracking scenarios. These constraints have motivated the development of reinforcement learning-based strategies, which dynamically integrate multiple tracking approaches to improve adaptability and efficiency.

A recent study [11] introduced a novel CPT framework that leverages deep RL to fuse bio-inspired and probabilistic tracking strategies. In this approach, two deep RL algorithms—Double Deep Q-Network (DDQN) and Deep Deterministic Policy Gradient (DDPG), were trained to determine the optimal tracking behavior in response to changing environmental conditions. The system dynamically switched between a moth-inspired *surge-and-cast* behavior and a Bayesian inference-based strategy, allowing for adaptive plume tracking in both laminar and turbulent flow environments. The training process was conducted in a high-fidelity simulated ocean setting, as illustrated in Figure 2.2, where an AUV successfully tracks a simulated hydrothermal plume.

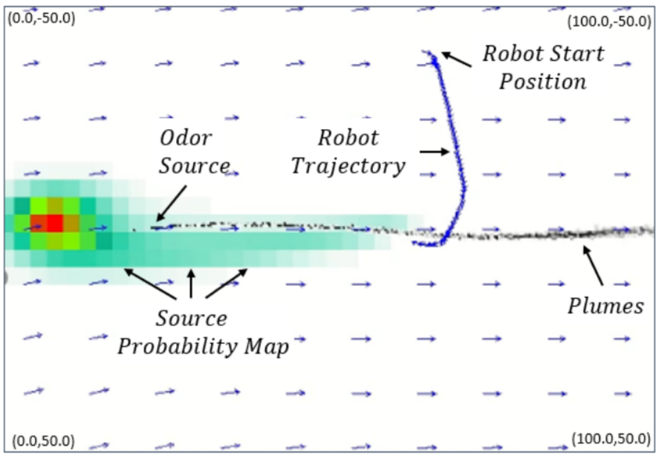


Figure 2.2: Simulation of an AUV tracking a hydrothermal plume using reinforcement learning-based strategies. Adapted from [11].

The findings from this study are particularly relevant to the work presented in this thesis, as both investigations employ RL to enable autonomous agents to monitor oceanic features. While the primary focus of the referenced study was hydrothermal vent detection, the underlying methodology of using deep RL to integrate multiple tracking strategies closely aligns with the approach adopted in this work. However, a key distinction lies in the environmental assumptions: while the CPT study considers a dynamic ocean setting with turbulent flows, the present work simplifies the problem by assuming a static environment, neglecting ocean currents. Moreover, while the referenced study specifically focuses on tracking hydrothermal plumes, the framework developed in this thesis is designed for general **ocean feature monitoring**, which could include oil spills or other oceanographic phenomena. Despite these differences, both approaches rely on RL to process real-time environmental data and adapt the agent's behavior accordingly.

Moreover, the use of deep RL for decision-making enhances the agent's ability to operate in complex environments where traditional methods fail. The integration of DDQN and DDPG allows the system to transition smoothly between different search strategies based on environmental feedback, significantly improving tracking efficiency in turbulent conditions.

The insights gained from this research highlight the potential of deep RL-based tracking systems for environmental monitoring. By enabling autonomous agents to adapt to complex and dynamic conditions, these methods pave the way for more effective and scalable solutions in oceanic surveillance, disaster response, and ecological research.

2.3 Baseline System for Target Tracking in Marine Robotics

One of the key contributions that shaped this study is the RL framework developed by Masmitja et al. [12], which serves as the foundation for the modifications implemented in this work. Originally designed for *underwater target tracking*, this system enables an autonomous surface vehicle (ASV) to estimate the position of a submerged target using *range-only localization* techniques.

Unlike the studies previously discussed, which focused on biologically inspired odor plume tracking [10] and deep RL-based hydrothermal vent localization [11], this framework was specifically developed for RL applications in marine robotics. The environment integrates several real-world constraints such as *ocean currents, measurement errors, and acoustic communication failures*, providing a robust platform for training autonomous agents in underwater navigation.

Underwater target tracking presents significant challenges compared to other RL-based tracking problems, as it involves optimizing vehicle navigation in highly dynamic and uncertain marine conditions. Oceanographic monitoring is often hindered by the unreliability and limited bandwidth of underwater communication systems, where GPS technology is not operational [29]. As a result, robotic platforms and autonomous underwater vehicles (AUVs) have emerged as essential tools for gathering oceanographic data while reducing costs and improving mission efficiency [30, 31, 32, 33]. However, as mission complexity increases, so do the challenges related to system reliability and endurance. Since energy resources are limited, optimizing control strategies becomes essential to ensure efficient and prolonged operations in underwater environments.

The future development of autonomous vehicles capable of dynamically adapting to environmental changes could enable large-scale studies, reducing the uncertainty that often characterizes oceanographic research. ML-driven control systems are emerging as a promising solution to enhance autonomy and adaptability in these challenging marine settings [34, 35].

To align with the objectives of this study, the original framework was significantly modified, shifting its focus from *target tracking* to *ocean feature monitoring.* This transition required adjustments in various aspects of the environment, including: reward function, observation space, policy optimization process.

2.4 Conclusion

This chapter has explored the role of *robotics* in environmental monitoring applications such as plume tracking across different domains, emphasizing its significance in both *aerial* and *marine* environments. Two fundamental studies were reviewed:

one investigating RL-based *odor plume tracking* in airborne scenarios [10], and another demonstrating *deep RL techniques for hydrothermal vent detection* in complex underwater conditions [11].

However, the most significant contribution shaping this work is the *baseline RL framework* developed by Masmitja et al. [12]. This system provides a robust environment for underwater navigation and RL-based target tracking, forming the foundation for the modifications introduced in this study to enable ocean feature monitoring.

In the next chapter, the key modifications introduced to transition from a target-tracking RL system to a framework for autonomous ocean feature monitoring will be presented in detail.

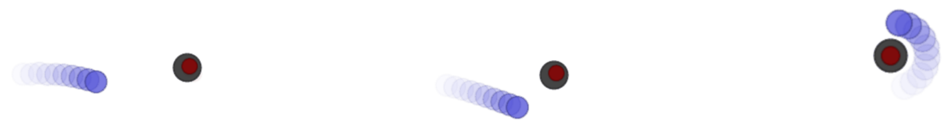
Chapter 3

Methodology

In this work, a new RL-based environment was developed to track ocean features, adapting an existing RL framework initially designed for underwater target tracking. The baseline environment, developed by Masmitja et al. [12], focused on training an autonomous surface vehicle (ASV) to track a moving submerged target using deep RL techniques such as **SAC** [36], **DDPG**, and **TD3**. This system was built on top of the **OpenAI Particle Environment** [37, 38], integrating realistic constraints such as **ocean currents, measurement errors, and communication failures**.

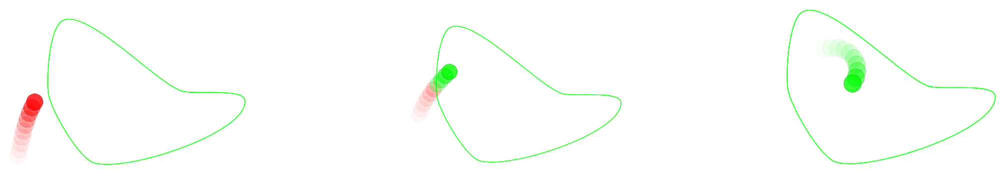
To address the specific problem of **monitoring ocean features**, the original environment was modified by introducing a **patch**, which the agent needs to detect and remain within. This adaptation required **changes to the rendering system, the agent's reward function, and the simulation parameters**. Furthermore, a **chemical concentration sensor** was implemented to determine whether the agent was inside or outside the polluted area. This sensor was used in conjunction with a **distance-based detection system**, providing supplementary information to enhance the agent's monitoring performance.

Before diving into the technical details of the methodology, it is useful to visualize the transition from the original project to the modified system developed in this study. The figures below illustrate the graphical representation of the environment before and after adaptation.



(a) Initial trajectory **(b)** Midway trajectory **(c)** Final trajectory

Figure 3.1: Agent trajectory in the original target-tracking environment. The ASV, represented by the blue dot, follows a submerged target (black dot) using a range-only tracking approach. The red dot represents the estimated target position computed using a Least Squares (LS) method.



(a) Initial trajectory **(b)** Approaching the patch **(c)** First entry into the patch

Figure 3.2: Agent movement within the adapted environment. The trajectory illustrates the agent's approach and initial interaction with the feature patch (in green). Further details and a comprehensive analysis of the results are provided in Chapter 4.

The **original environment** (Figure 3.1) was designed for **underwater target tracking**, where an ASV was trained to follow a submerged target using a **range-only tracking method**. This technique estimates the target's position using **acoustic distance measurements** and estimation algorithms such as **Least Squares (LS)** and **Particle Filter (PF)**. The ASV, trained via deep RL, optimizes its trajectory to maintain a **stable acoustic link** with the target.

In contrast, the **modified environment** (Figure 3.2) shifts the focus towards **ocean features monitoring**, where the agent must learn to navigate and accurately localize and follow the boundary of a **dynamically generated patch** instead of tracking a single moving target. The modifications introduced required a **redesign of the rendering system, the localizaton sensors and techniques, the agent's reward function, and the simulation parameters**.

This chapter details the methodology used to build and train this new environment, explaining the modifications applied to the original system, the design of the agent, and the learning algorithms employed.

3.1 Algorithms

The control of ASVs in complex marine environments requires robust decision-making strategies that can handle uncertainty, sensor noise, and dynamically changing conditions. RL has emerged as a powerful tool for optimizing control policies in such scenarios. The underlying idea is to train an agent that interacts with the environment, receives feedback through rewards, and learns to maximize long-term performance.

3.1.1 Baseline Algorithms Used in the Original Framework

The original framework developed by Masmitja et al. [12] employs **deep RL** techniques, particularly focusing on **off-policy actor-critic methods**. The main challenge in using **deep Q-learning** in continuous state and action spaces is ensuring **stability and convergence** [39]. To address this issue, **Deep Deterministic Policy Gradient (DDPG)** [40] was implemented, allowing the policy to learn a deterministic mapping from states to actions while using an actor-critic structure. However, DDPG suffers from hyperparameter sensitivity, making it difficult to train reliably.

To improve stability and sample efficiency, **Soft Actor-Critic (SAC)** [36] was introduced in the original framework (Figure 3.3). SAC is based on the **maximum entropy RL framework**, where the objective is not only to maximize the expected reward but also to **maximize the entropy** of the policy, encouraging more exploratory behavior. This property is particularly useful in robotic applications where sensor readings contain noise or environmental conditions change dynamically. By exploring multiple strategies to complete a task, SAC allows for more **robust and adaptable decision-making**. Both **DDPG** and **SAC** were implemented and tested as part of the **range-only target tracking and path-planning system**.

Additionally, **Long Short-Term Memory (LSTM) networks** were incorporated into the SAC architecture to study their impact on performance (Figure 3.4). Two different recurrent structures were developed:

- **LSTM-SAC:** A recurrent neural network (RNN) that takes as input the last n states, inspired by [41, 42].
- **H-LSTM-SAC:** A simplified implementation where one of the hidden layers

is replaced by an RNN with a history length of 1. The internal hidden state is used as input at the next step as a memory unit.

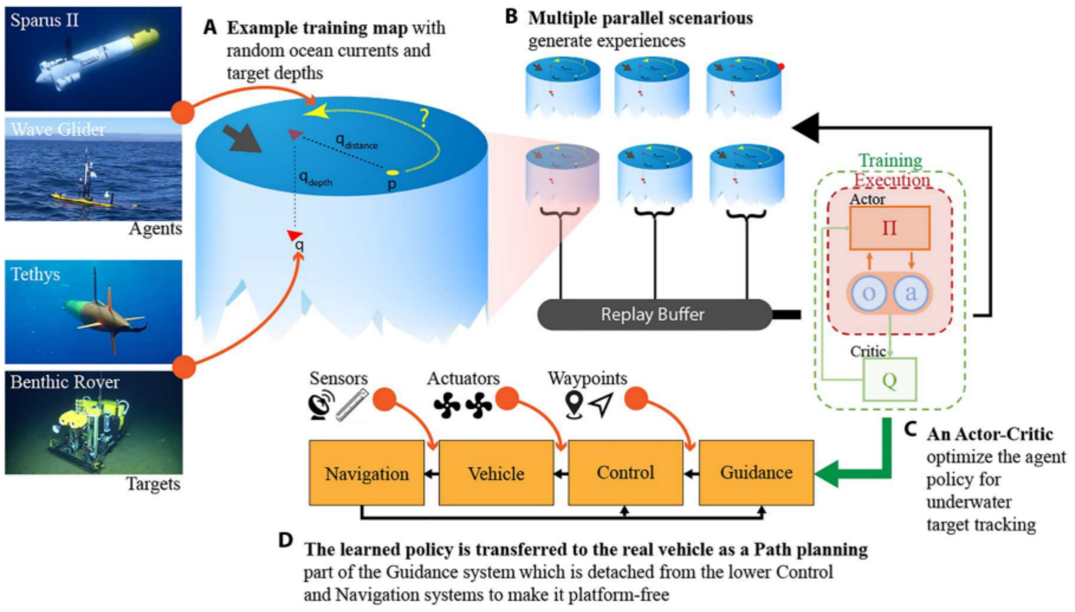

Sparus II
Wave Glider
Agents
Tethys
Benthic Rover
Targets
A Example training map with random ocean currents and target depths
B Multiple parallel scenarious generate experiences
Replay Buffer
Training
Execution
Actor
Critic
C An Actor-Critic optimize the agent policy for underwater target tracking
Sensors
Actuators
Waypoints
Navigation
Vehicle
Control
Guidance
D The learned policy is transferred to the real vehicle as a Path planning part of the Guidance system which is detached from the lower Control and Navigation systems to make it platform-free

Figure 3.3: Deep RL concept as range-only path planning. An agent was trained in a virtual environment that uses real conditions, such as ocean currents and distance measurement noise (A). During the training, multiple parallel scenarios were used to boost the process (B), and different actor-critic algorithms were studied (C). Last, the policy learned was transferred to the real vehicle as a path planning method as part of its guidance system (D) [12].

3.1.2 Adaptation to the Ocean Features Monitoring Problem

The transition from underwater target tracking to ocean feature monitoring required a fundamental shift in the agent's objective. Instead of pursuing a moving target using acoustic range measurements, the agent needed to learn how to identify and interact with oceanic phenomena, such as oil spills, Posidonia meadows, or algal blooms. This shift necessitated modifications in both the training framework and the environment dynamics to better reflect the new monitoring task.

One of the most significant modifications involved the observation space, which redefined how the agent perceives its surroundings and directly influenced the structure of the neural network's input layer. As described in Equation 3.3, the

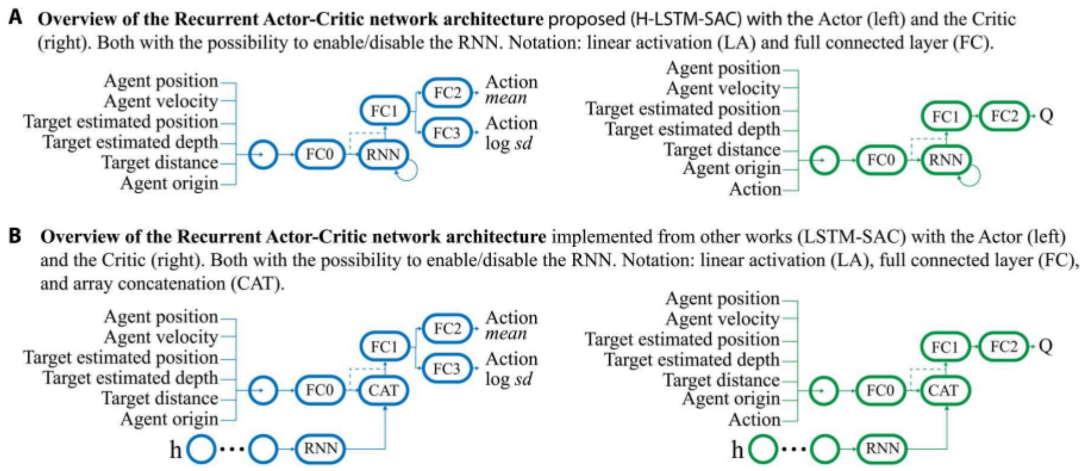

A Overview of the Recurrent Actor-Critic network architecture proposed (H-LSTM-SAC) with the Actor (left) and the Critic (right). Both with the possibility to enable/disable the RNN. Notation: linear activation (LA) and full connected layer (FC).
Agent position
Agent velocity
Target estimated position
Target estimated depth
Target distance
Agent origin
Action
FC0
FC1
FC2
FC3
RNN
Action mean
Action log sd
Q
B Overview of the Recurrent Actor-Critic network architecture implemented from other works (LSTM-SAC) with the Actor (left) and the Critic (right). Both with the possibility to enable/disable the RNN. Notation: linear activation (LA), full connected layer (FC), and array concatenation (CAT).
CAT
h

Figure 3.4: A high-level representation of the implemented deep recurrent RL algorithms. The proposed H-LSTM-SAC algorithms in which a single-cell RNN was used and the hidden state was passed to the nextstep (A) and the version implemented from previous works, where an RNN with a history h of the last n observations was used (B). Both architectures can enable or disable the RNN part (dotted line). In addition, besides the SAC architecture, a DDPG and a TD3 were also implemented.[12].

observation space in the original framework included agent's position and velocity, the relative position of surrounding entities, the relative position of other agents (when in a multi-agent setting), the range measurement to the target, the target's depth and the agent's origin point .

These modifications expanded the observation vector, requiring adjustments to the first fully connected layer of the neural network in **Multi-Agent Hybrid Recurrent SAC (MAHRSAC) algorithm** to accommodate the increased input dimensionality. A detailed breakdown of the observation states is provided in Section 3.5.2.

In order to minimize changes to the original structure, the training was conducted using the **MAHRSAC**. This algorithm is an adaptation of **SAC** [36], integrating RNN to enhance temporal dependencies in decision-making. The implementation is based on **Hybrid Recurrent SAC (HRSAC)** and incorporates modifications inspired by existing **PyTorch SAC frameworks** [43, 44, 45].

Although **MAHRSAC** is generally designed for **multi-agent** learning, it was adapted for **single-agent training** by configuring the **agent count** in the configuration file:

```
# Number of agents per environment
num_agents = 1
```

By setting this parameter to **1**, the existing MAHRSAC implementation was effectively converted into a single-agent **Hybrid Recurrent SAC** model. This approach allowed the training process to remain consistent with the original deep RL framework, avoiding the need for extensive modifications to the network architecture or environment interface, and allowing future applications in MARL configurations.

The **MAHRSAC algorithm** extends SAC by integrating **recurrent memory components**, making it particularly well-suited for monitoring **dynamic ocean features**. Even though the original framework leveraged LSTM-enhanced SAC architectures (LSTM-SAC and H-LSTM-SAC), the implementation of MAHRSAC inherently maintains memory over past states, enabling better adaptability to complex spatial patterns, such as evolving oceanic formations.

The reinforcement learning agent optimizes its behavior by maximizing the total expected return:

$$R = \sum_{t=0}^{T} \gamma^t r_t \tag{3.1}$$

where:

- R is the total expected reward,
- γ is the discount factor controlling the importance of future rewards,
- r_t is the reward at time step t,
- T is the episode horizon.

3.2 Environment Design

3.2.1 Original Environment

The simulation environment used in this study is based on the **OpenAI Particle Environment** [37], originally designed for multi-agent interactions in continuous observation and action spaces. The baseline framework developed by Masmitja et al. [12] adapted this environment for **underwater target tracking** by integrating a **range-only estimation algorithm** and visualization tools based on **(LS)** and **(PF)** methods.

The **target estimation module** plays a crucial role in this environment. The LS method, which provides fast but less adaptable estimations, was used during training due to its lower computational cost compared to PF, which is more effective for tracking moving targets. The **distance measurements** between the agent and the target were modeled using a Gaussian noise distribution.

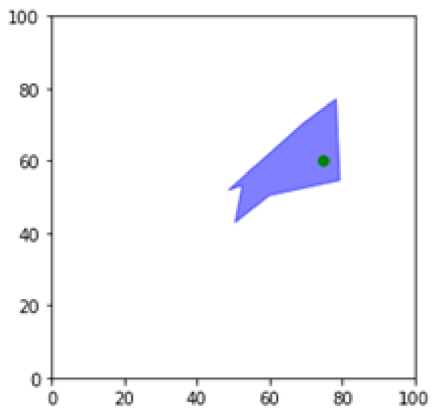
The environment also incorporated **ocean current effects**, simulating their impact by modifying the agent's position at each time step. To introduce additional realism, a **dropping factor** was implemented, simulating potential **communication failures** between the agent and the target, preventing distance measurements in certain conditions. During training, both the target's **position, velocity, and movement direction** were randomized to simulate real-world tracking challenges.

3.2.2 Modified Environment: Ocean Feature Monitoring

To accommodate this new objective, the original target-tracking environment was redesigned into a system where the agent interacts with a dynamically generated patch, representing an ocean feature. Unlike the previous setup, where the agent followed a submerged target, the new design required it to recognize and adapt its behavior based on the spatial properties of the patch.

This adaptation involved the removal of the target-tracking components, including the target itself and the associated estimation modules. These modifications provided the agent with a new challenge: rather than chasing a single entity, it had to develop a strategy to remain engaged with an extended environmental feature.

The first step in implementing the new environment was the **patch generation system**. Initially, this was tested separately using an external script, where a **random polygon** was generated in a defined space. A reference implementation from [46] was used as a starting point, but significant modifications were made to meet the study's requirements. The original implementation created **four small patches** in a limited area, which was not suitable for this application. To address this, the code was modified to generate **a single patch per execution** in a random position within a broader environment space. The **scaling of the patch** was adjusted to match the simulation's spatial parameters. The initial patch generation method produced irregular shapes (Figure 3.5a), which were later refined using Bézier curves to obtain smoother contours better suited for simulation (Figure 3.5b) [46, 47, 48, 49, 50].


(a) Initial patch generation: irregular shape

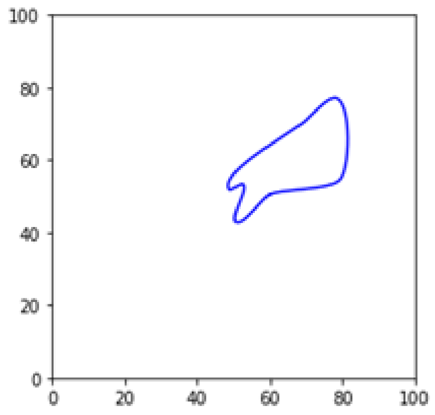

(b) Refined patch using Bézier curves

Figure 3.5: Comparison of patch generation methods. (a) The initial method produced irregularly shaped patches, which were not well-suited for simulation. (b) The refined approach used Bézier curves to smooth the contours, resulting in a more realistic and usable patch.

3.2.3 Patch Generation Using Bézier Curves

Initially, the randomly generated patches had highly irregular shapes, which could lead to unrealistic scenarios. To smooth the patch boundaries, a **Bézier curve approach** was adopted [50]. The patch was generated using a set of random control points, which were then smoothed using a Bézier interpolation method.

The Bézier interpolation was implemented using the function shown in Listing 3.1.

Listing 3.1: Bézier Curve Generation Function

```
import numpy as np
from scipy.special import binom
...
def bezier(points, num=200):
    """
    Generates a Bézier curve from a set of control points.

    Parameters:
    points : array-like
        List of control points defining the curve.
    num : int, optional
        Number of points in the generated curve.

```

```python
    Returns:
    curve : ndarray
        Array of (x, y) coordinates representing the Bézier curve.
    """
    bernstein = lambda n, k, t: binom(n, k) * t**k * (1. - t)**(n - k)

    N = len(points)
    t = np.linspace(0, 1, num=num)
    curve = np.zeros((num, 2))

    for i in range(N):
        curve += np.outer(bernstein(N - 1, i, t), points[i])

    return curve
```

This function generates a Bézier curve from a set of random control points, ensuring a smooth and continuous boundary for the patch.

3.2.4 Integration of the Patch System into the Original Framework

To enable visualization and interaction with the dynamically generated patch within the original environment, modifications were required in multiple components of the codebase.

The first step in this integration involved duplicating the existing *tracking* script. The original `tracking.py` file was copied and renamed as `patch.py`, which was then modified to support the new simulation logic. Since ocean currents and landmark velocities were no longer relevant to the ocean feature monitoring problem, these components were removed to simplify the environment.

To facilitate the generation and rendering of the polygonal patch, a new script `polygon_generator.py` was introduced. This file imported all necessary libraries and functions from the initial standalone patch generation script, integrating them into the simulation framework. Unlike the original system, which relied exclusively on circular objects using:

Listing 3.2: Function to Generate a Circular Shape

```python

def make_circle(radius=10, res=30, filled=True):
    points = []
    for i in range(res):
        ang = 2 * math.pi * i / res
        points.append((math.cos(ang) * radius, math.sin(ang)
    * radius))

    if filled:
        return FilledPolygon(points)
    else:
        return PolyLine(points, True)
```

the updated framework introduced a new function:

Listing 3.3: Function to Generate a Polygon

```

def make_polygon(v, filled=True):
    if filled:
        return FilledPolygon(v)
    else:
        return PolyLine(v, True)
```

which allowed the creation of arbitrary polygonal shapes based on a set of input vertices.

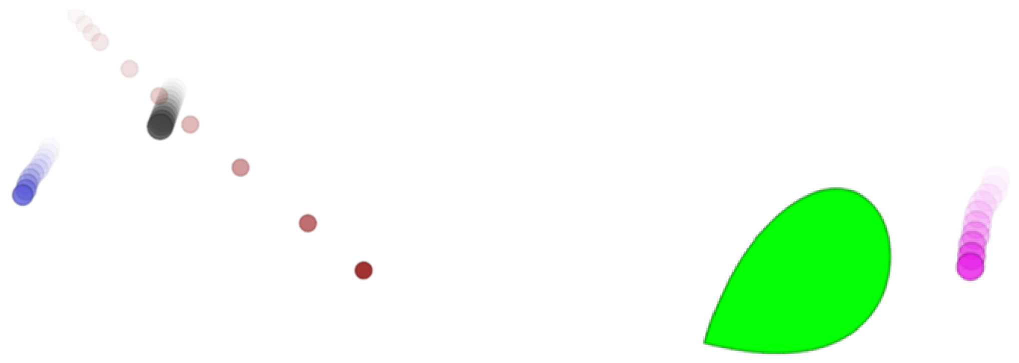
Finally, modifications were made to `environment.py` to correctly render the patch within the visualization system. The following code snippet (Listing 3.4) illustrates how the generated polygon was integrated into the rendering pipeline:

Listing 3.4: Integration of the dynamically generated patch into the rendering system.

```
# Create geometry from polygon vertices
geom = rendering.make_polygon(v=self.poly, filled=True)

# Set green color with transparency
geom.set_color(0, 1, 0, alpha=0.8)

# Append the new polygon geometry to the rendering list
self.render_geoms.append(geom)

# Add transformation attributes
self.render_geoms_xform.append(xform)
```

This modification ensured that the patch was visually represented within the simulation, replacing the original circular target visualization.

Figure 3.6 illustrates the original environment at the start of training, where the agent was tasked with tracking a moving target. After the integration of the patch system, the modified rendering (Figure 3.6) now displays the dynamically generated ocean feature, enabling a new mode of interaction for the agent.



Original environment: agent tracking a moving target.
Legend: Blue dot = agent, Black dot = target, and Red dot = predicted target position.

Modified environment: agent interacting with a patch.
Legend: Green patch = ocean feature, Pink circle = agent.

Figure 3.6: Comparison between the original and modified environment visualization. The agent now interacts with a dynamically generated patch instead of a moving target.

3.3 Sensors Implementation

In RL-based tracking systems, sensors play a fundamental role in providing the agent with essential environmental information. This study required the adaptation of the original tracking system to monitor ocean features, replacing the acoustic range-based detection system with a combination of **chemical concentration sensing** and **distance-based detection**.

3.3.1 Chemical Concentration Sensor

The primary sensing mechanism in this study was a **chemical concentration sensor**, responsible for determining whether the agent was inside or outside a feature patch. This was achieved through a **point-in-polygon (PIP) detection algorithm** [51], which evaluates the agent's position relative to the patch boundaries in real time.

The implementation leveraged the **Shapely** library, specifically its built-in functions for geometric operations:

- `within()`: Determines whether a given `Point` is inside a `Polygon`.

- `contains()`: Checks whether a `Polygon` fully encloses a `Point`.

The `contains()` function was used to check whether the agent's position was within the dynamically generated ocean feature patch. This function is based on standard computational geometry techniques for PIP detection, such as:

- **Ray-Casting Algorithm**: A horizontal ray is cast from the point in question, and the number of intersections with the polygon's edges is counted. An odd number of intersections indicates that the point is inside, while an even number means it is outside.

- **Winding Number Algorithm**: Computes how many times the polygon winds around the point. A nonzero winding number indicates the point is inside the polygon.

Shapely's implementation is optimized for computational efficiency, likely integrating one of these standard techniques to enable real-time detection in the RL environment.

Listing 3.5: Implementation of the Chemical Concentration Sensor

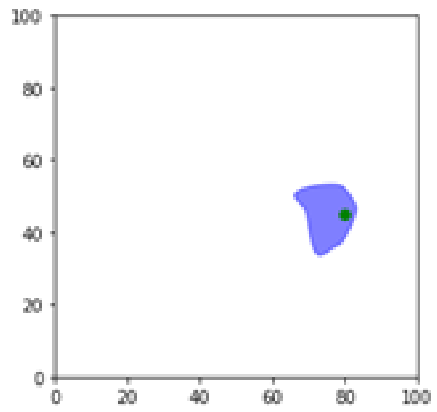
```python
from shapely.geometry import Point, Polygon

agent_position = Point(agent.state.p_pos)  # Current agent position
self.world.inside = patch_polygon.contains(agent_position)

if self.world.inside:
    agent.color = [0, 1, 0]  # Green when inside the patch
else:
    agent.color = [1, 0, 0]  # Red when outside the patch
```

Agent Interaction with the Patch

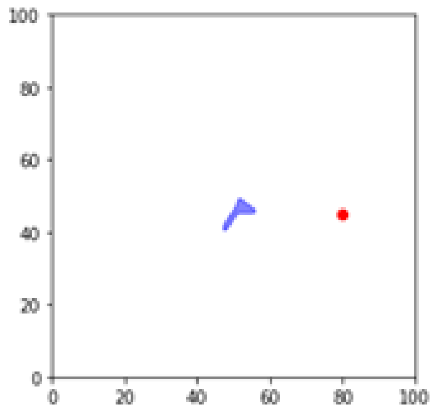
To facilitate debugging and validation, a visual representation of the agent's state was introduced:

- The agent is displayed in **green** when inside the patch.

- The agent is displayed in **red** when outside the patch.



(a) Agent inside the patch



(b) Agent outside the patch

Figure 3.7: Visual representation of the agent's state based on its position relative to the patch. (a) The agent turns **green** when inside the patch and (b) **red** when outside, providing a clear indication of its detection status.

This visualization played a crucial role in validating the effectiveness of the implemented detection mechanism.

Real-World Implications

In real-world applications, chemical concentration sensors are widely used to measure environmental properties such as salinity, pollutants, or pH levels [52]. Among these, **pH sensors** are particularly relevant for detecting environmental changes in aquatic ecosystems.

The virtual chemical sensor implemented in this study serves as a conceptual approximation of such real-world sensors. In a future deployment, an autonomous system could integrate an actual chemical sensor, such as a pH detector, to dynamically identify and track oceanic features, mirroring the functionality simulated in this study.

3.3.2 Distance-Based Sensor

In addition to the chemical concentration sensor, a **distance-based sensor** was implemented to estimate the agent's proximity to the monitored ocean feature. This sensor provides spatial awareness, allowing the agent to refine its trajectory by detecting proximity to the patch boundaries.

The distance estimation was implemented using the `distance()` function from the Shapely library [53], which computes the **minimum Euclidean distance** between the agent's position and the patch perimeter.

Listing 3.6: Computation of the Agent's Distance from the Patch Perimeter

```
# Compute the agent's distance from the patch perimeter
self.distance_to_perimeter = polygon.exterior.distance(agent_position
    )
```

The key characteristics of this approach include:

- If the agent is **inside the patch**, the distance is automatically set to **zero**.
- If the agent is **outside the patch**, the function computes the shortest Euclidean distance to the patch perimeter.

While this method provides an effective approximation in simulation, real-world applications would require an actual **camera-based distance sensor**. Such sensors typically rely on vision-based techniques, including object detection and geometric depth estimation [54], to determine an agent's distance from a detected feature.

By integrating this distance information into the reinforcement learning process, the agent was encouraged to refine its trajectory, enhancing tracking accuracy and optimizing navigation strategies. A more detailed discussion of its impact on learning is provided in Section 3.4 (Reward Function Design).

3.4 Reward Function Design

The design of the **reward function** is a crucial aspect of RL, as it directly influences the agent's learning process and final behavior. A well-defined reward function ensures that the agent learns meaningful strategies aligned with the task's objectives, whereas a poorly designed reward function can lead to suboptimal behavior or convergence to undesired policies.

To align with the objective of **ocean feature monitoring**, the reward function was redesigned to guide the agent in detecting and interacting with a dynamically generated patch, rather than tracking a moving target. Unlike the original framework by Masmitja et al. [12], which relied on range-only measurements, the adapted approach encourages the agent to learn an exploration strategy that allows it to effectively delineate the patch boundaries and adjust its trajectory accordingly.

3.4.1 Reward Function in the Original Framework

The original implementation used a combination of **dense and sparse rewards**, where the agent was incentivized to reduce its distance to the target while improving the accuracy of its estimated position. Specifically, three main reward components were defined:

- **Distance-Based Reward**: Encouraged the agent to minimize the distance to the estimated target position.
- **Estimation Error Reward**: Penalized discrepancies between the predicted and actual target positions.
- **Terminal Reward**: Penalized excessive deviations from the target and rewarded successful tracking.

The final reward function was computed as:

$$r = r_d + r_e + r_{\text{terminal}} \tag{3.2}$$

where each term contributes to optimizing the tracking trajectory.

3.4.2 Redesigning the Reward Function for Ocean Feature Monitoring

Unlike target tracking, where the agent follows a moving object, the objective in this work is to monitor a specific ocean feature by learning to detect and respond to its presence.

The detection of the ocean feature is facilitated by a binary variable called the `inside` flag. This variable stores the output of the chemical concentration sensor, assigning a value of 1 when the agent is inside the patch and 0 when it is outside. Effectively, it represents the agent's perception of the feature based on simulated chemical concentration measurements.

Two different reward function designs were implemented:

1. **Chemical Concentration-Based Reward**: The agent receives a reward based on its position relative to the patch.
2. **Distance-Based Reward**: The agent is rewarded based on its proximity to the patch boundary.

Both approaches were tested separately to evaluate their effects on learning dynamics. In the second approach, even though the reward function did not explicitly rely on the chemical concentration sensor, the agent still received the `inside` flag as input, indirectly incorporating feature presence into the policy.

Reward Function Based on Chemical Concentration

The first reward function directly utilizes the **chemical concentration sensor**, reinforcing the agent for remaining inside the patch.

The reward function was designed to incentivize the agent to detect and interact with the ocean feature effectively. The implementation is shown in Listing 3.7.

Listing 3.7: Updated reward function implementation. The agent is incentivized to detect and interact with the ocean feature while ensuring effective navigation.

```python
def reward(self, agent, world):

    # Reset at each function call to ensure reward calculation is independent of previous states
    global done_state
    done_state = False  # When done_state = True, the episode is terminated

    rew = 0.0  # Initialize the reward

    # STEP 1: Basic inside detection (commented out alternative)
    # if world.inside == True:
    #     rew += 1
    # else:
    #     rew -= 0.1

    # STEP 2: Reward based on inside history buffer
    if len(world.inside_history) > 0:
        aux = sum(world.inside_history)  # Sum of the last recorded inside values

        if aux > 1:  # Agent has been inside recently
            rew += 1  # Encourage staying near the patch
            # Save current position as the new reference point
            agent.state.p_pos_origin = agent.state.p_pos.copy()
        else:
            rew -= 0  # No penalty for remaining outside

    # Termination condition: End the episode if the agent moves too far
    dist_from_origin = np.sqrt(np.sum(np.square(agent.state.p_pos - agent.state.p_pos_origin)))
    if dist_from_origin > self.set_max_range * 2.:  # Agent is outside the valid world boundary
        rew -= 100  # Strong penalty for leaving the environment
        done_state = True
        self.agent_outofworld += 1
    else:
        rew -= 0.1  # Small penalty for excessive wandering

    return rew  # Return the computed reward
```

The variable `inside_history` is a **memory buffer** that stores the last eight values of the `inside` flag. This allows the agent to track whether it has been inside or outside the patch over the previous eight time steps, providing a short-term memory of its interaction with the feature.

The variable `done_state` is used to indicate whether the training episode should terminate. Specifically:

- **Initialization:** At the beginning of the reward function, `done_state` is set to `False`, indicating that the episode is still active.
- **Termination condition:** If the agent's distance from its original position exceeds a predefined threshold (`self.set_max_range * 2.`), `done_state` is set to `True`, signaling that the episode should end.
- **Effect:** When `done_state = True`, the reinforcement learning framework can terminate the current episode and start a new one, preventing the agent from straying too far from the patch without obtaining useful information.

In summary, `done_state` acts as a **safety mechanism** to prevent ineffective exploration and ensure that the agent remains focused on interacting with the ocean feature.

Key Features of this Reward Function:

- Encourages the agent to detect and interact with the patch, learning to navigate around it effectively.
- Discourages prolonged absence from the patch.
- Implements a termination condition to prevent excessive wandering far from the relevant area.

Reward Function Based on Distance to Patch

The second reward function leverages the **distance to the patch perimeter** as its primary metric. This approach incentivizes it to navigate around the boundary, ensuring efficient interaction with the ocean feature.

The implementation is given by the following code snippet (Listing 3.8):

Listing 3.8: Reward function based on agent's distance to the patch boundary. The agent is rewarded for staying near the perimeter of the patch while penalized for being too far.

```python
def reward(self, agent, world):

    global done_state
    done_state = False  # Reset at each reward calculation

    rew = 0.0  # Initialize the reward

    polygon_points = world.polygon_points  # Retrieve polygon points

    # Create the polygon
    try:
        polygon = Polygon(polygon_points)
    except Exception as e:
        pass  # Handle potential errors in polygon creation

    agent_position = Point(agent.state.p_pos)  # Convert agent position to a Point object

    # Compute the distance from the agent to the patch perimeter
    self.distance_to_perimeter = polygon.exterior.distance(agent_position)

    # Reward logic: The closer the agent is to the perimeter, the higher the reward
    if self.distance_to_perimeter < 0.05 or world.inside == True:  # Threshold for being "on the perimeter"
        rew += 1.0  # Reward for being near the perimeter
        agent.state.p_pos_origin = agent.state.p_pos.copy()  # Update reference position
    else:
        rew -= self.distance_to_perimeter  # Penalize the agent based on distance

    # Termination condition: If the agent moves too far from the original reference position
    dist_from_origin = np.sqrt(np.sum(np.square(agent.state.p_pos - agent.state.p_pos_origin)))
    if dist_from_origin > self.set_max_range * 2.:  # Agent outside valid area
        rew -= 100  # Strong penalty for leaving the environment
        done_state = True
        self.agent_outofworld += 1

    return rew  # Return the computed reward
```

Key Features of this Reward Function:

- Encourages the agent to stay near the boundary of the patch.
- Provides continuous feedback, improving training stability.
- Allows for better adaptation in scenarios where precise boundary detection is required.

3.5 Observation Space

In a RL framework, the **observation space** represents the information available to the agent at each time step. The design of this spaces significantly impacts the learning process, as it determines how the agent perceives its surroundings and interacts with them.

3.5.1 Observation Space in the Original Framework

In the original target-tracking framework developed by Masmitja et al. [12], the observation vector consisted of various environmental and agent-related parameters. Specifically, at each time step, the agent received the following information:

- The agent's velocity $\mathbf{v}_t$
- The agent's position $\mathbf{p}_t$
- The relative position of surrounding entities (e.g., landmarks) $\mathbf{e}_t$
- The relative position of other agents $\mathbf{o}_t$ (when in a multi-agent setting)
- The range measurement to the target d_{range}
- The target's depth d_{depth}
- The agent's origin point $\mathbf{p_o}$

Formally, the observation at time step t was defined as:

$$o_t = [\mathbf{v}_t, \mathbf{p}_t, \mathbf{e}_t, \mathbf{o}_t, d_{\text{range}}, d_{\text{depth}}, \mathbf{p_o}] \tag{3.3}$$

This observation structure was specifically designed for a range-only target tracking problem, where the agent relied on acoustic distance measurements to estimate the target's position. The inclusion of range information (d_{range}) and depth measurement (d_{depth}) was essential for optimizing the agent's trajectory.

However, in the newly developed environment, the objective shifted from tracking a moving target to monitoring ocean features, requiring a revision of the observation space to better accommodate the new task.

3.5.2 Modified Observation Space for Ocean Feature Monitoring

In this study, two different observation structures were implemented, corresponding to the two different reward functions discussed in the previous section:

Observation Structure Based on Historical Data

In the first formulation, the agent received **historical data** on whether it had been inside the patch in recent time steps, providing a more nuanced understanding of its trajectory relative to the feature:

$$o_t = [\mathbf{v}_t, \mathbf{p}_t, \text{inside\_history}, \mathbf{p_o}] \tag{3.4}$$

This formulation complemented the first reward function, which used chemical concentration data (Listing 3.7).

Observation Structure Incorporating Distance Information

In the second formulation, the agent received information from both the chemical concentration sensor and the distance sensor. The observation vector included the agent's velocity and position, the distance to the patch boundary, the agent's origin point and a binary flag indicating whether the agent was inside or outside the patch:

$$o_t = [\mathbf{v}_t, \mathbf{p}_t, d_{\text{perimeter}}, \mathbf{p_o}, \text{inside}] \tag{3.5}$$

where:

- $d_{\text{perimeter}}$ represents the agent's distance to the patch boundary.

This observation structure aligned with the second reward function (Listing 3.8), which utilized both chemical concentration feedback and distance measurements to guide the agent's navigation.

The flexibility of these observation spaces allowed for a comparative analysis of different sensor-based learning approaches, helping determine the most effective strategy for autonomous ocean feature monitoring.

With the methodological framework established, the next step is to evaluate the performance of the RL agent. The following chapter presents the results obtained from the training process, analyzing how different reward functions and observation spaces influenced the agent's ability to monitor ocean features effectively.

Chapter 4

Results

The performance of the reinforcement learning (RL) agent in the newly designed environment is crucial for evaluating the effectiveness of the proposed methodology. This chapter presents a comprehensive analysis of the experimental results obtained from training and testing the agent under various configurations.

In line with the methodological framework outlined in Chapter 3, the results focus on assessing the agent's ability to navigate and monitor ocean features effectively, utilizing two distinct observation structures and reward functions, as introduced in Sections 3.4.2 and 3.5.2. The evaluation emphasizes key performance metrics, including the agent's ability to remain within the feature patch and the distance between the agent and the patch.

Before diving into the results of these experiments, the first section of this chapter provides an overview of the computational environment used for the agent's training and testing. Specifically, the role of the remote server **Drago** is discussed, highlighting how it facilitated the efficient execution and monitoring of the RL experiments. This section serves to set the context for the subsequent analysis of the agent's performance, as the computational resources available were crucial to obtaining meaningful results in a timely manner.

4.1 Use of Drago for RL Agent Training

In this work, the training of the reinforcement learning (RL) agent was carried out on a High-Performance Cluster (HPC), named **Drago**, from ICM-CSIC in Spain.

The use of the HPC Drago was essential for several reasons, including:

- **High Computational Capacity:** The training of RL agents, particularly with deep reinforcement learning algorithms, demands significant computational resources. These algorithms rely heavily on neural networks, requiring intense operations such as *backpropagation* and the updating of network weights.

Additionally, executing numerous simulations in parallel further exacerbates the computational demand.

- **GPU Utilization:** Drago is equipped with high-performance GPUs, which dramatically accelerate the training process. Training without GPU support would have resulted in prohibitively long timescales, possibly taking several days or weeks to reach a satisfactory level of learning.
- **Remote Execution and Monitoring with TensorBoard:** Drago, being a remote server, allows experiments to be run without overloading the local PC. The use of monitoring tools, such as **TensorBoard**, enabled real-time observation of critical metrics, such as the evolution of the reward function, loss values, and other key parameters, directly from the workstation.
- **Parallel Simulation Execution:** To collect statistically significant data across various experiments (e.g., Update and InOut setup tests), hundreds of simulations were required. On a regular computer, this would have taken an unacceptably long amount of time. Drago, however, provided the capability to run multiple simulations in parallel, significantly reducing the time required for data collection and analysis.

In summary, Drago proved to be indispensable in enabling efficient and timely development, testing, and analysis of the RL environment. Its high computational resources, GPU support, remote execution capabilities, and parallel simulation processing allowed for a comprehensive exploration of the RL algorithms and sensor configurations, contributing to the successful completion of this project.

Additionally, an appendix has been included with the main instructions and configurations for using Drago (see Appendix A).

4.2 Test with InOut Configuration

To evaluate the agent's performance under the first reward function formulation, a series of experiments were conducted using the **InOut** configuration.

Four test cases were designed, all based on the same observation space defined in Section 3.5.2 as:

$$o_t = [\mathbf{v}_t, \mathbf{p}_t, \text{inside\_history}, \mathbf{p_o}] \tag{4.1}$$

This configuration provides a binary flag indicating whether the agent is currently inside the feature patch. Instead of using only the current value, the observation space includes the `inside_history`, a vector that stores the last eight values of this flag, allowing the agent to incorporate short-term temporal information when making decisions.

4.2.1 Test Configurations

The following test cases were executed under the same scenario, with the agent's initial position set to (0,0):

- **Test 5**: The reward function assigns a value of +1 when the agent is inside the patch (`inside = True`).

- **Test 56**: Identical to Test 5, but with a larger reward of +10 when inside the patch.

- **Test 57**: Same as Test 56, but the `done` condition was removed. For more details on the `done` condition, see Section 3.4.2.

- **Baseline Test**: A reference test in which the agent does not rely on a learned policy. The agent starts at the center of the environment and moves outward in a gradually expanding circular motion, forming a **spiral trajectory**. This continues until the agent encounters the feature patch, at which point it remains inside. The predefined motion is governed by an increasing radius variable, ensuring a systematic search pattern. This trajectory is illustrated in Figure 4.1, where the agent's spiral motion converges on the feature patch.

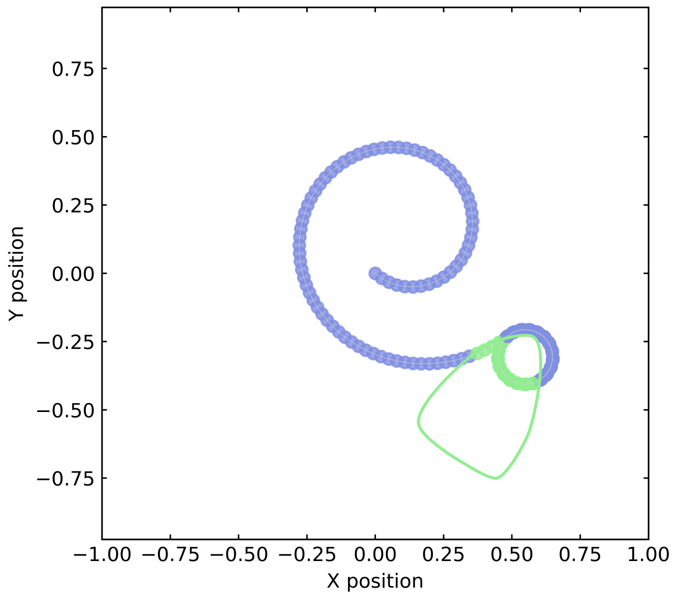


Figure 4.1: Trajectory followed in the Baseline Test. The agent (blue) starts at the center and follows an expanding spiral trajectory until it enters the feature patch (green), where it remains.

Each test was initially evaluated at its respective best-performing checkpoint, based on the evolution of the reward function of each test:

- Test 5: 900k steps
- Test 56: 1600k steps
- Test 57: 600k steps
- Baseline: no learning involved

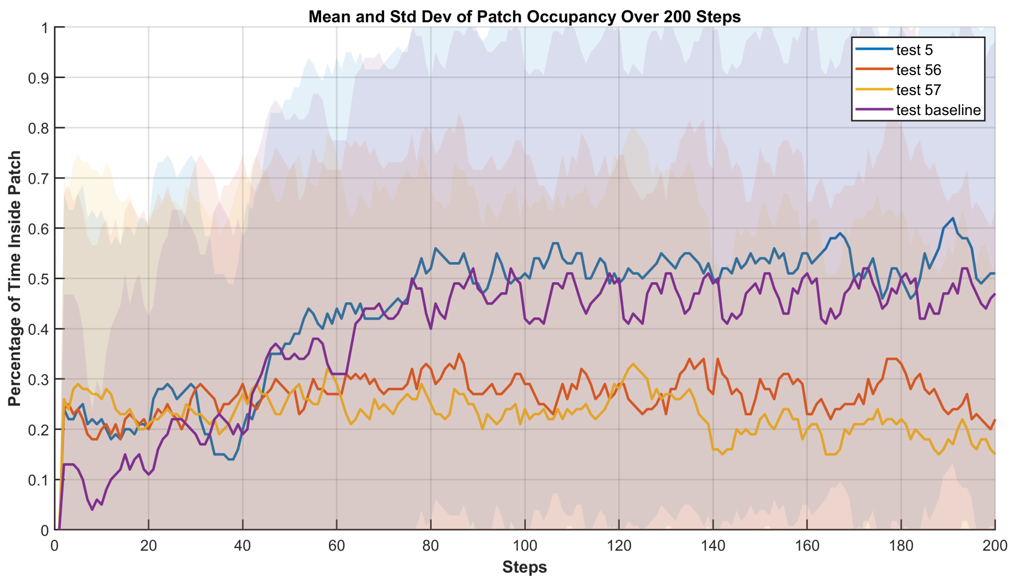


Figure 4.2: Comparison of average patch occupancy across the four InOut setup tests at their respective best-performing checkpoints.

As shown in Figure 4.2, **Test 5** outperforms the other configurations in terms of average time spent inside the patch. Despite the higher reward in Test 56, its learning outcome is less stable and effective. Test 57, where the `done` condition is disabled, performs worse, likely due to the absence of a termination mechanism that encourages the agent to stay within the patch. The baseline test, as expected, performs moderately but serves primarily as a reference trajectory rather than a competing policy.

4.2.2 Reward Function Analysis and Selection of Evaluation Points

To further analyze the learning dynamics of Test 5, the evolution of the reward function was studied in detail.

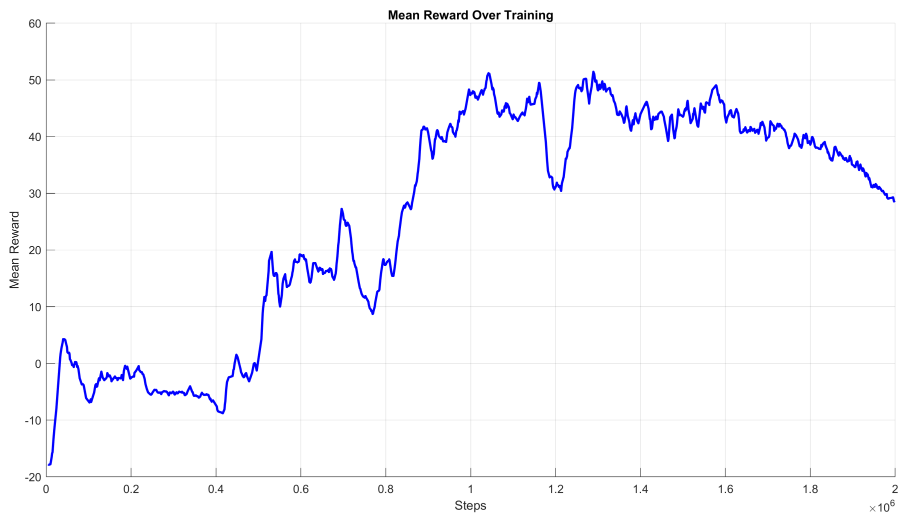


Figure 4.3: Mean reward over training steps for Test 5 (InOut configuration). The highest average reward is observed at 1300k steps.

From the reward curve shown in Figure 4.3, five training checkpoints were selected for deeper analysis: **400k**, **600k**, **900k**, **1300k**, and **1900k**. These points include the highest reward value and additional intermediate steps to observe how performance evolves.

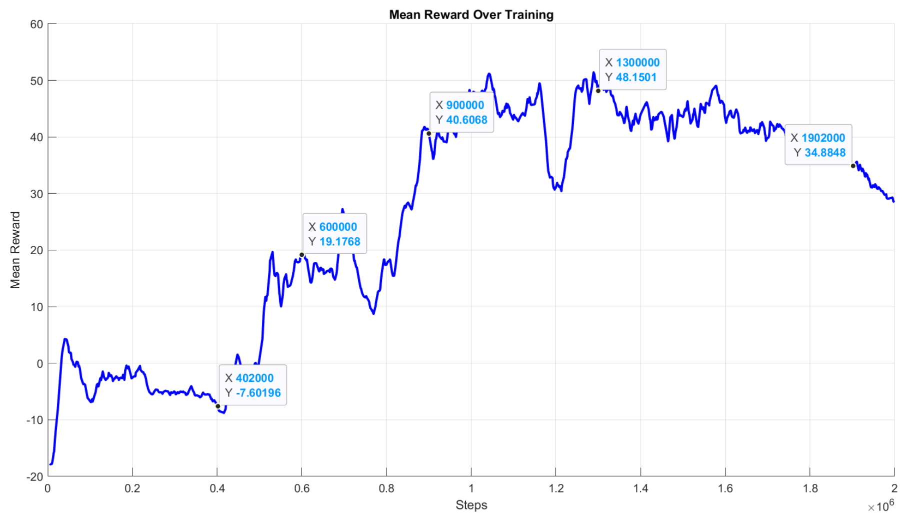


Figure 4.4: Mean reward over training steps for Test 5 (InOut configuration). Five specific checkpoints were selected for further evaluation: 400k, 600k, 900k, 1300k, and 1900k.

4.2.3 Performance Evaluation Across Five Checkpoints

Following the selection of the five main training checkpoints (400k, 600k, 900k, 1300k, and 1900k), each model was evaluated over 100 independent simulations. Although the 1300k checkpoint achieved the highest average reward (see Figure 4.4), further behavioral analysis revealed that the models at 600k and 900k achieved superior performance in terms of time spent inside the patch.

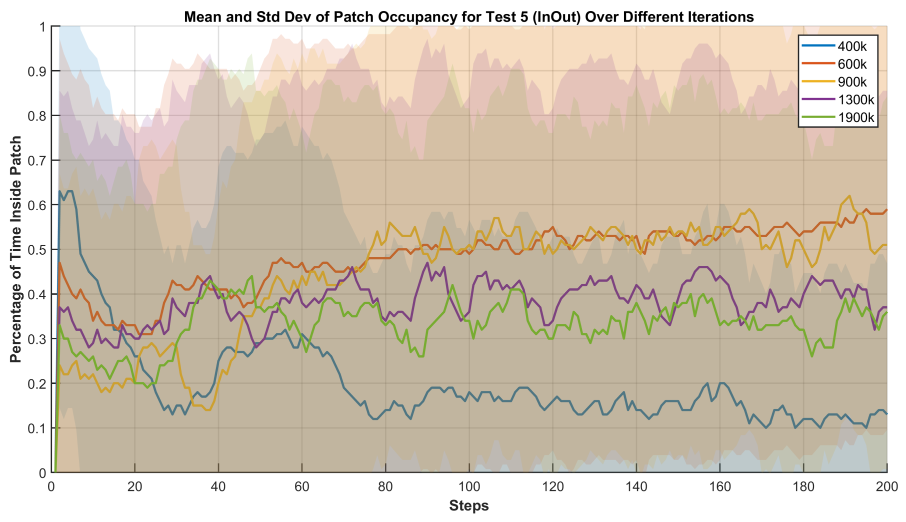
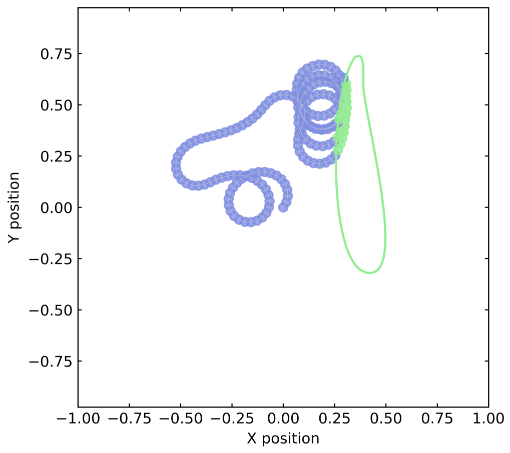


Figure 4.5: Mean and standard deviation of patch occupancy over 200 steps for five different training checkpoints in Test 5. The 600k and 900k models show more consistent and higher occupancy than the 1300k model, despite the latter having the highest mean reward during training.

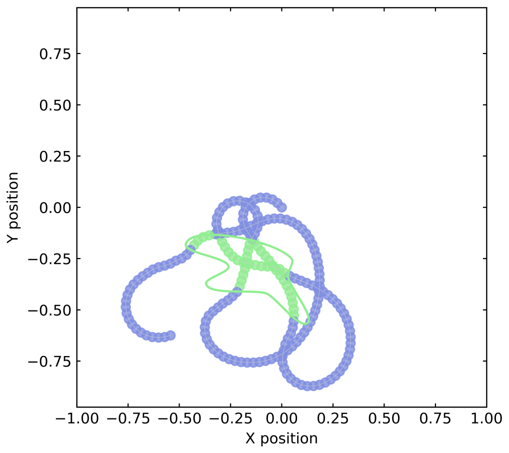
To further analyze the agent's behavioral differences across training stages, Figure 4.6 illustrates the search trajectories for the models trained at 600k, 900k, and 1300k iterations.

As seen in Figure 4.6, the models trained at 600k and 900k exhibit a more extensive exploration phase before consistently reaching and remaining in the patch. In contrast, the 1300k model demonstrates a more constrained movement pattern, indicating less exploration. This suggests that, although the 1300k agent achieved the highest mean reward during training, its learned policy may have become too conservative, limiting its adaptability in more dynamic scenarios.

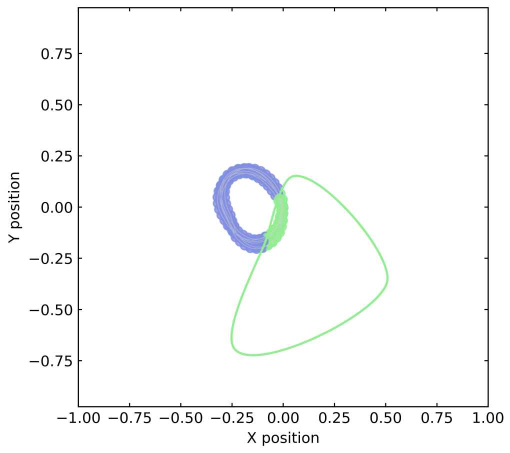
The broader search behavior observed at 900k enables the agent to efficiently locate and track the patch over extended time horizons, leading to superior long-term performance in patch occupancy (Figure 4.5). Based on these observations, the 900k checkpoint was selected for final evaluation.



(a) Agent trajectory at 600k



(b) Agent trajectory at 900k



(c) Agent trajectory at 1300k

Figure 4.6: Agent search trajectories at different training checkpoints. The 600k and 900k models exhibit broader exploration patterns before converging to the patch, while the 1300k model shows a more constrained search behavior.

Step-by-Step Patch Occupancy (900k)

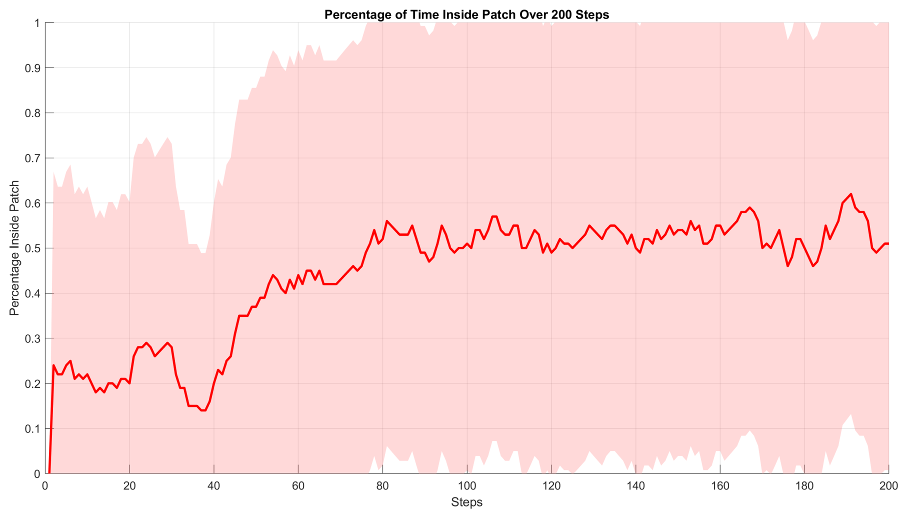


Figure 4.7: Percentage of time spent inside the patch over 200 simulation steps for the 900k checkpoint. The shaded area indicates the standard deviation across 100 simulations.

In Figure 4.7, the red curve represents the average percentage of time the agent remains inside the patch at each step, across 100 simulations. The occupancy increases progressively over time, stabilizing around 50% in the final portion of the simulation. The narrow standard deviation band confirms the stability and repeatability of the policy.

Distribution Across Time Windows

To evaluate temporal consistency, the simulation was divided into three time intervals: early (steps 1–50), middle (steps 75–125), and late (steps 150–200). The boxplot in Figure 4.8 shows the distribution of patch occupancy across these intervals.

The agent shows a clear improvement in patch occupancy from the early to the middle and late phases of the episode. The median and mean values both increase steadily, and the interquartile range narrows over time, confirming that the agent not only finds the patch but learns to remain inside it consistently.

Occupancy Distribution Across Simulations

Finally, the histogram in Figure 4.9 provides a global overview of performance across all simulations. A large number of simulations achieve patch occupancy

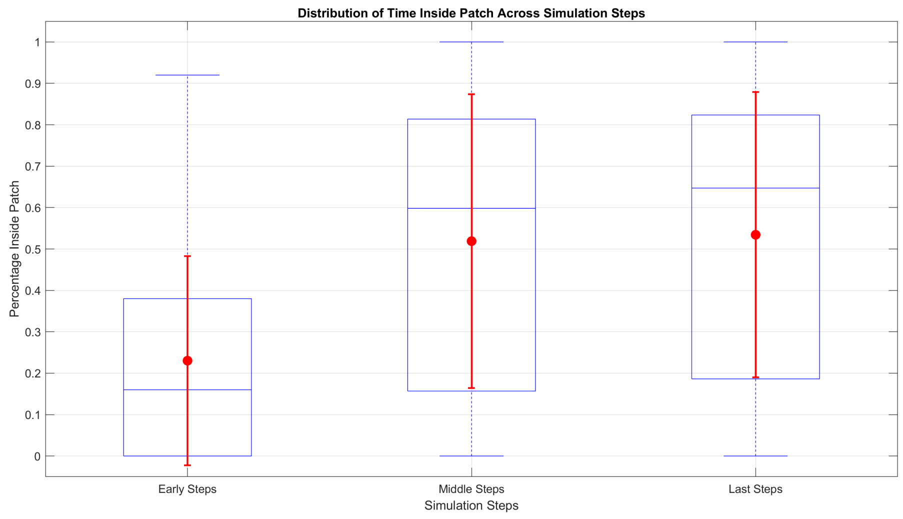


Figure 4.8: Boxplot of patch occupancy over early, middle, and late simulation steps for the 900k checkpoint. The red markers represent the mean values.

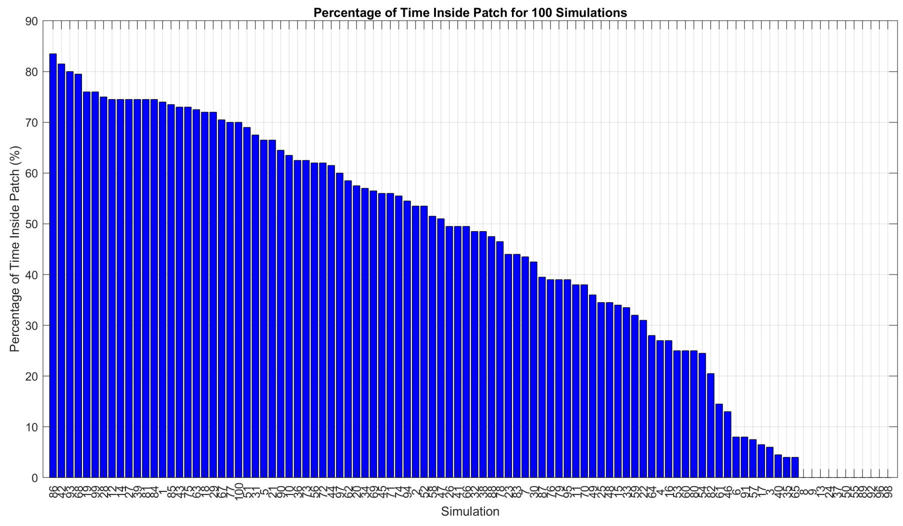


Figure 4.9: Percentage of time spent inside the patch for each of 100 simulations (900k checkpoint). Simulations are sorted by performance.

levels above 70%, with several exceeding 80%. Only a small portion of episodes fall below 40%, confirming that the learned policy is not only effective but also robust across different starting conditions.

These metrics collectively indicate that the agent trained at 900k steps offers the best balance between reward function performance and reliable behavior. The policy is stable, generalizable, and consistently leads the agent to enter and remain within the feature patch throughout the simulation.

4.3 Test with Update Configuration

To evaluate the agent's performance under the second reward function formulation, experiments were conducted using the **Update** configuration. This configuration provides the agent with both chemical concentration data and distance to the patch boundary. Five test cases were designed to assess different aspects of the training process.

4.3.1 Test Configurations

The following test cases were executed under the same scenario, with the agent's initial position set to (0,0):

- **Test 2**: The observation space in this test includes only the agent's velocity ($\mathbf{v}_t$), position ($\mathbf{p}_t$), and distance to the patch boundary ($d_{\text{perimeter}}$).

  $$o_t = [\mathbf{v}_t, \mathbf{p}_t, d_{\text{perimeter}}] \tag{4.2}$$

  The agent was evaluated at its best-performing checkpoint, reached at $800k$ training steps, where the highest cumulative reward was observed (see Figure 4.11).

- **Test 23**: This test follows the same configuration as Test 2 but includes two additional observation states: the `inside` flag, indicating whether the agent is within the patch, and the agent's origin point ($\mathbf{p_o}$).

  $$o_t = [\mathbf{v}_t, \mathbf{p}_t, d_{\text{perimeter}}, \mathbf{p_o}, \text{inside}] \tag{4.3}$$

  The test was conducted using the checkpoint corresponding to the highest average reward during training, which occurred at $1800k$ steps.

- **Test 26**: Identical to Test 23, except that the agent's initial position is set to (0,0). The highest average reward during training was achieved at $1300k$ steps.

- **Test 27**: This test follows the same configuration as Test 26 but removes the *done* condition (see explanation in Section 3.4.2). The agent's most effective performance was observed at the $1600k$ step checkpoint.

- **Baseline Test**: Unlike the previous tests, this scenario does not involve a learned policy. Instead, the agent follows a predefined trajectory, independent of environmental feedback. This trajectory is illustrated in Figure 4.1, where the agent's spiral motion converges on the feature patch.

4.3.2 Initial Selection of Best Checkpoint

For each test case, the model checkpoint corresponding to the highest average reward was selected and evaluated through 100 simulations. This allowed for a statistically meaningful comparison across all five configurations.

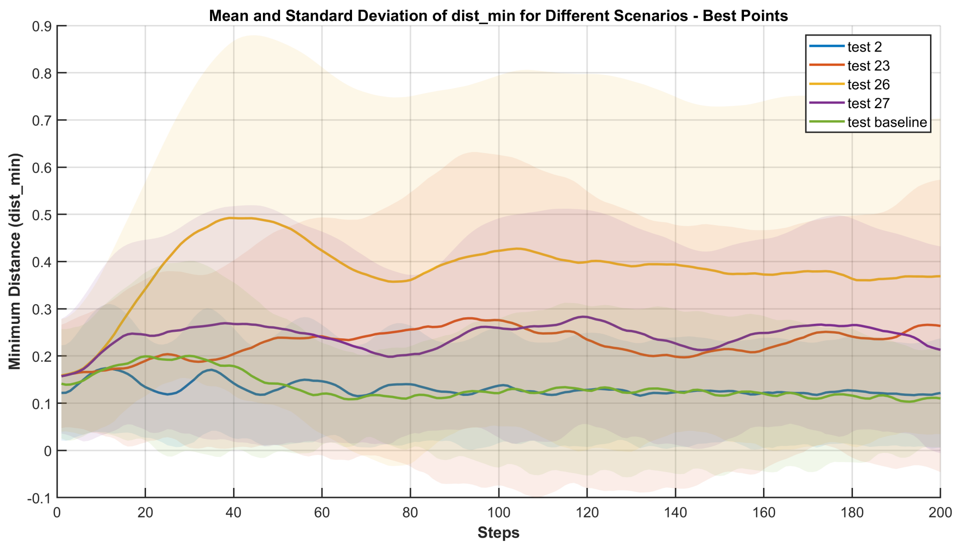


Figure 4.10: Comparison of minimum distance trends across the five Update test cases at their respective best-performing points.

From this comparison, **Test 2** emerged as the most effective configuration and was selected for further analysis. Despite having a reduced observation space compared to other configurations, Test 2 demonstrated superior performance. This suggests that increasing the number of observation states does not necessarily lead to better tracking efficiency. Instead, a more concise state representation may help the agent focus on the most relevant features of the environment, reducing potential distractions and facilitating more stable learning. The results indicate that Test 2 effectively balances environmental awareness and decision-making, leading to a more robust tracking behavior.

4.3.3 Global Comparison Across Training Steps

The evolution of the reward function for Test 2 is shown in Figure 4.11, where five checkpoints were selected: one at the highest average reward (800k), and four others chosen to observe performance trends over training.

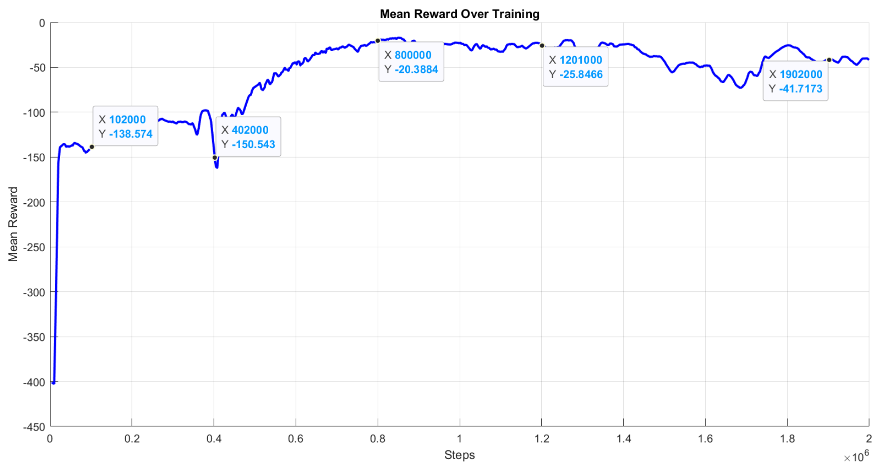


Figure 4.11: Mean reward over training steps. The best-performing checkpoint was reached at 800k iterations.

To evaluate performance behaviorally, each checkpoint was simulated over 100 episodes, and average distance to the patch was computed.

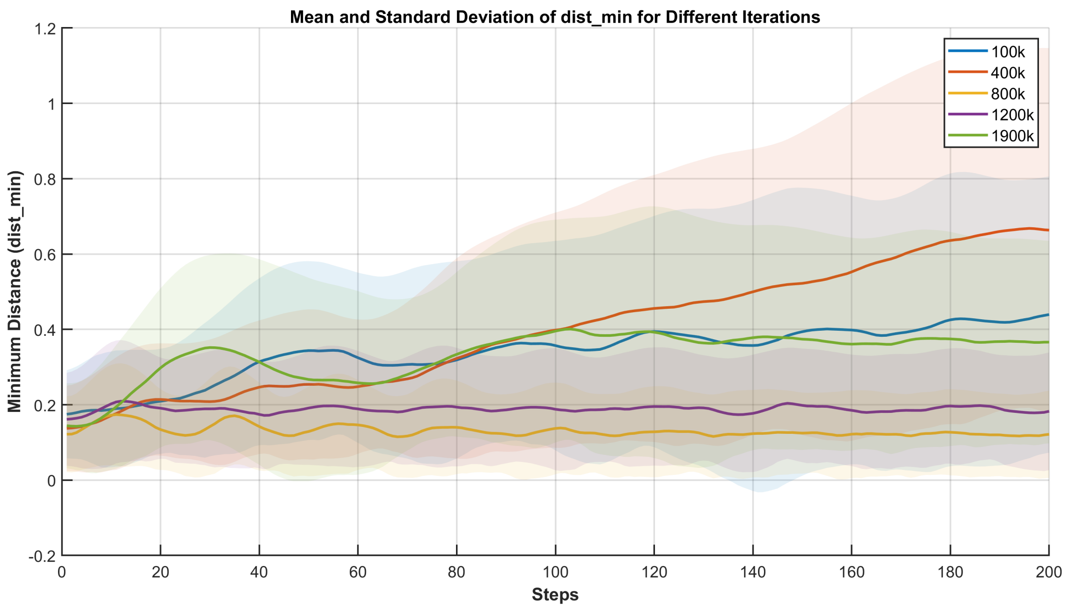


Figure 4.12: Comparison of the average minimum distance over 200 steps for five different training iterations in Test 2 (Update configuration).

The best performance was observed at 800k and 1200k steps, both demonstrating low mean distances and narrow standard deviations. In contrast, the 1900k model showed signs of overfitting and degraded consistency.

4.3.4 Performance Metrics at Best Checkpoint (800k)

Based on the trends shown in Figure 4.12, the 800k checkpoint was selected for detailed metric analysis. The following subsections report the agent's tracking performance in different forms.

Step-by-Step Distance to Patch

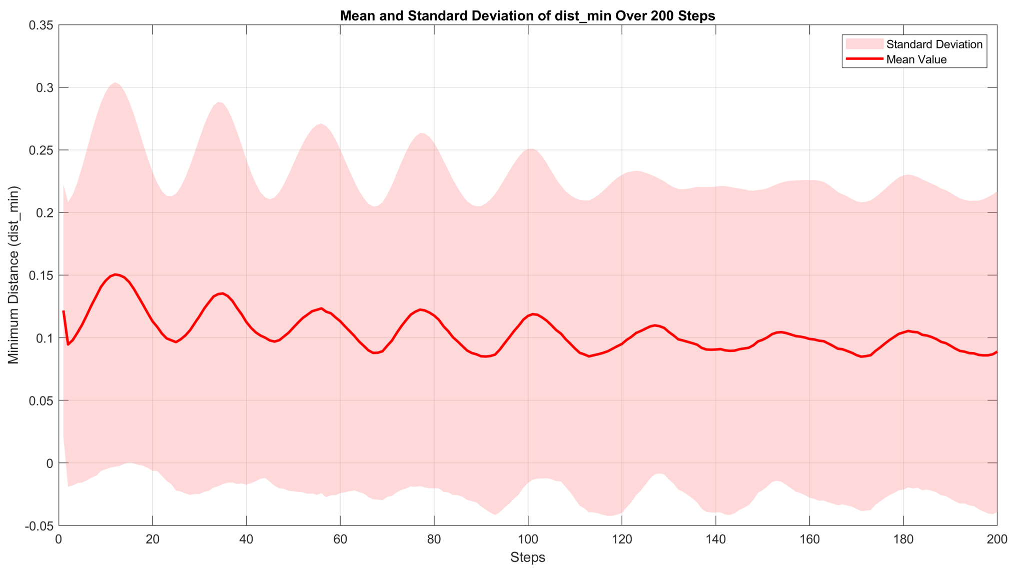
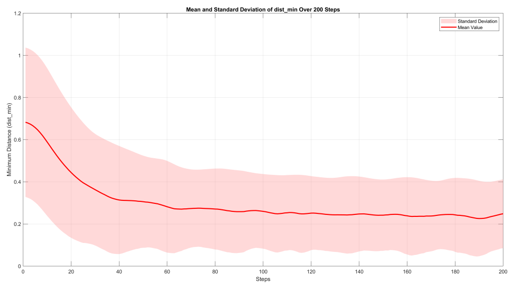


Figure 4.13: Mean and standard deviation of the minimum distance over 200 steps at the 800k checkpoint (Test 2, Update). The red curve indicates the mean distance, while the shaded area represents standard deviation across 100 simulations.

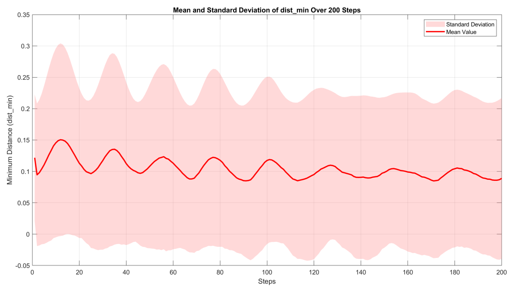
Figure 4.13 shows that the agent maintains a low average distance to the patch across the episode, with limited fluctuations and strong stability. This confirms the agent's ability to remain close to the target once detected.

Effect of Initial Agent Position

To understand how the initial position affects learning, the agent was evaluated both with random starting points and from a fixed origin (0,0). The results are shown in Figure 4.14.



(a) Old configuration: agent starts away from the patch



(b) New configuration: agent starts at the origin (0,0)

Figure 4.14: Comparison of mean and standard deviation of the minimum distance over time for the 800k checkpoint under two initial agent positions.

The fixed-origin configuration (Figure 4.14b) shows more stable results, as the agent starts closer to the patch. However, the random-start configuration better highlights the learning trajectory from exploration to convergence.

Distribution Across Time Windows

To assess temporal robustness, patch distance was analyzed in three windows: early (1–50), middle (75–125), and late (150–200) steps.

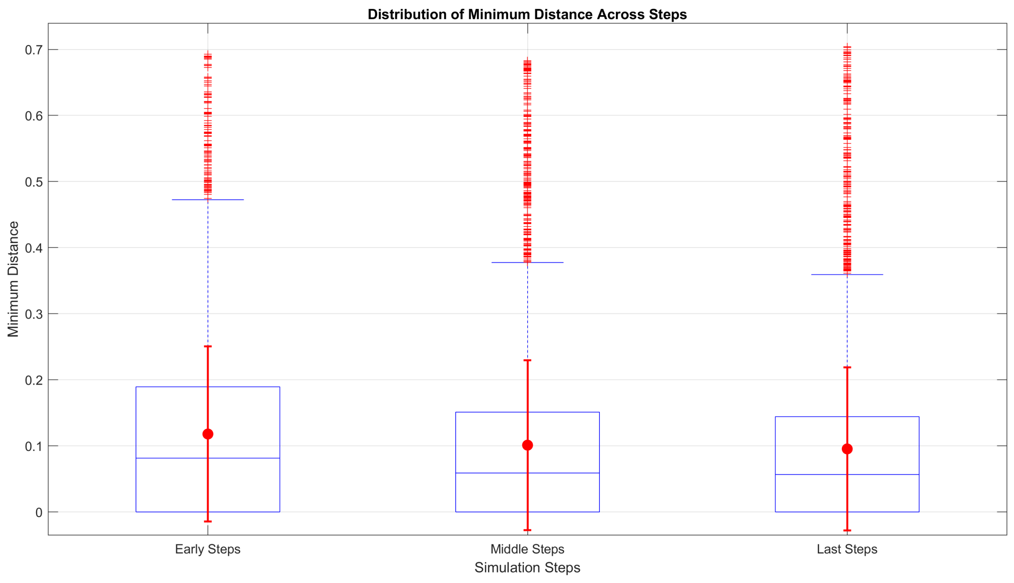


Figure 4.15: Boxplot of the minimum distance to the patch across three time intervals of the simulation.

As shown in Figure 4.15, the agent consistently maintains proximity to the patch across all intervals. The low dispersion confirms that the learned policy is both effective and stable throughout the episode.

4.4 Comparison Between InOut and Update Configurations

This section presents a comparative analysis between the two tested approaches: the **InOut** configuration and the **Update** configuration. Although both setups were designed to guide the agent towards staying inside the target patch, their operational principles and learning outcomes differ significantly.

4.4.1 Sensing Configuration and Observation Space

The **InOut configuration** provides a binary indicator of whether the agent is inside the patch. This information is encoded in the observation vector as a short-term history (`inside_history`) over the last 8 steps (Equation 3.4). Despite the simplicity of this approach, the agent was able to learn effective strategies for patch occupancy.

The **Update configuration**, in contrast, offers richer feedback: it combines both chemical concentration data and the Euclidean distance to the patch boundary. This leads to a more informative observation space, potentially allowing the agent to reason more precisely about its relative location with respect to the patch.

4.4.2 Behavioral Performance Comparison

From a behavioral standpoint, both configurations led to successful learning of patch-seeking behavior. However, differences emerged in terms of:

- **Learning efficiency**: The InOut-based agent reached stable performance faster (around 600k–900k), while the Update setup required longer training (800k–1200k).
- **Occupancy stability**: InOut (at 900k) showed strong consistency, with narrow variance and a smooth convergence in patch occupancy. The Update setup (800k) exhibited slightly higher fluctuations over time, especially in early steps.
- **Interpretability**: The InOut setup produced more easily interpretable metrics, as the binary nature of the signal aligns directly with the reward. The Update configuration, while more expressive, introduced complexity in correlating reward signals with specific behaviors.

4.4.3 Which Configuration Performs Better?

Based on the final analysis, the **InOut configuration** demonstrates superior practical performance. Despite its minimalistic formulation, it led to:

- higher average patch occupancy (over 50% in the final steps),
- consistent behavior across simulations (as shown in Figures 4.7 and 4.9),
- a smoother and earlier convergence during training.

The Update configuration still holds value for future extensions (e.g., navigating towards complex gradients or dynamically changing plumes), but under the static conditions of this study, the InOut setup provided more robust and efficient learning.

4.5 Configurations Comparison and Final Considerations

To evaluate the effectiveness of the two setups modalities explored in this thesis, a comparative analysis was conducted between their best-performing configurations: Test 5 with the InOut configuration at **900k training steps** and Test 2 with the Update configuration at **800k training steps**. Figure 4.16 illustrates the average patch occupancy over 200 simulation steps, along with the standard deviation.

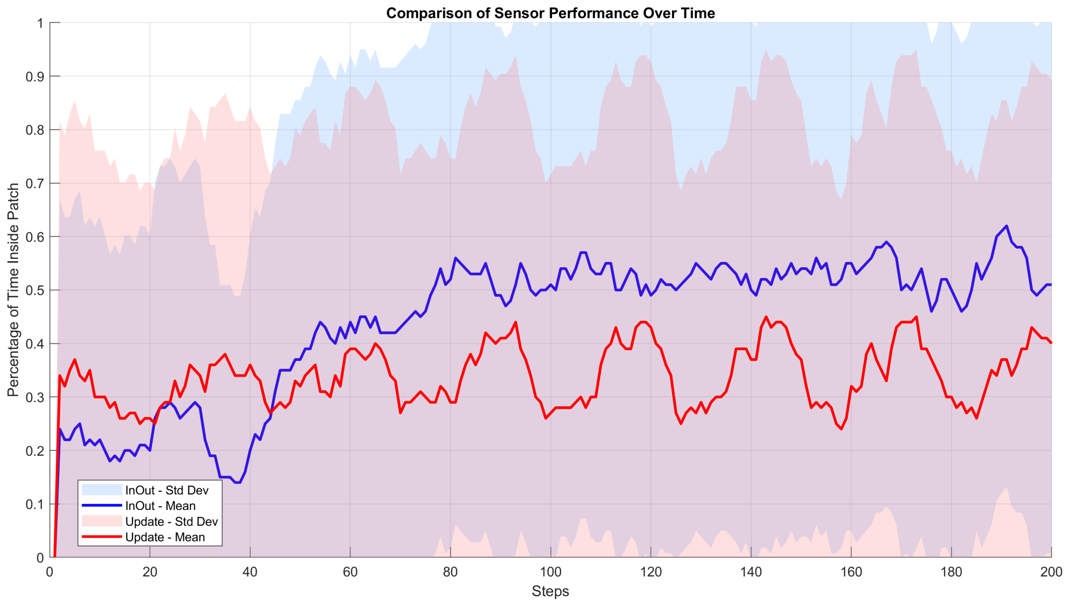


Figure 4.16: Patch occupancy comparison between InOut (900k) and Update (800k) setups over 200 simulation steps. Shaded areas represent one standard deviation.

The results clearly indicate the superior performance of the InOut setup. It achieves a consistently higher percentage of time inside the patch while exhibiting significantly lower variability across simulations. This suggests a more stable and reliable policy, an essential factor for real-world applications where robustness outweighs occasional peak performance.

Despite having access to a richer observation space, including distance to the patch perimeter and chemical concentration, the agent trained with the Update setup displayed lower and more fluctuating occupancy patterns. This instability may be attributed to the increased complexity of the Update configuration's reward landscape, which introduces additional noise during training.

These findings reinforce the principle that **clarity and simplicity in feedback mechanisms lead to more effective learning** [5]. The binary nature of the InOut setup provides unambiguous information (`inside` or `outside`), allowing the agent to develop a reliable strategy for maintaining patch occupancy. The observed performance confirms that minimal yet well-structured information is sufficient to drive the emergence of robust behaviors.

Chapter 5

Conclusions

5.1 Final Considerations

This thesis focused on the development of a RL environment for monitoring ocean features using autonomous vehicles. As outlined in Section 1.4, the primary objective was not to achieve an optimal agent capable of perfectly staying inside the patch but rather to construct a robust and flexible simulation environment. The proposed framework allows for the testing of different agent configurations, observation spaces, and reward functions, providing a foundation for future research in this domain.

The results obtained from the preliminary simulations highlighted key aspects:

- The *inside_history* buffer proved to be more effective than simply using the binary *inside* flag combined with the distance to the patch.
- The *InOut* setup showed superior performance in terms of patch occupancy and consistency compared to the Update setup.
- The analysis of reward functions provided valuable insights into how different feedback mechanisms influence the agent's learning process.

These findings suggest that well-structured, minimal information can significantly improve learning efficiency. The use of memory-based feedback, such as the *inside_history* buffer, enables the agent to make better decisions over time, leading to more stable and effective tracking behaviors, as also discussed in Section 2.2.

5.2 Future Directions

The results obtained in this work provide a strong foundation for further research in RL-based ocean monitoring. Several key areas for future investigation include:

- **Enhanced Observation Space:** Investigating the combination of the *inside_history* buffer with the agent-to-patch distance in the observation space to provide both historical and spatial awareness.
- **Improved Reward Functions:** Refining the reward structure to encourage more efficient exploration and tracking behaviors.
- **Integration of Environmental Dynamics:** Extending the environment by incorporating ocean currents, sensor noise, and other real-world constraints to make the simulation more representative of actual marine conditions.
- **Multi-Agent Exploration:** Expanding the framework to support multi-agent systems for cooperative monitoring and tracking.

In conclusion, the work presented in this thesis successfully establishes a structured RL environment for monitoring oceanic phenomena. The insights gained from the initial simulations provide a valuable starting point for refining agent behavior and optimizing RL strategies in marine monitoring applications. By continuing in this direction, future research can further enhance the realism, adaptability, and efficiency of autonomous ocean-monitoring systems.

Appendix A

How to work with HCP Drago

Documentation:

Listado de Software | Portal Documentación AIC SGAI

Basic instructions:

Modify files:

```
vim <nameofyourfile.extension>
```

After this command:

- To start inserting text at the cursor position, press `i` or `a`
- To save and quit forcefully, ignoring warnings, press ESC and then `:wq!`

Check modules installed:

```
module av
```

Instead of manually entering certain instructions each time, you can simply edit the .sh file by adding the instructions you want to be executed every time just before the last blank line.

Load modules

For example, if you don't want to load the modules every time, you can just put the following commands in the `runner.sh`, before the last blank line.

```
module unuse /dragofs/sw/campus/0.2/modules/all/Core
module unuse /dragofs/sw/restricted/0.2/modules/all/Core
module load fosscuda/2020b
module load Horovod/0.23.0-TensorFlow-2.5.0
module load rama0.3
module load CUDA/12.2.0
```

New stuff to run with GPU partition:

```
module load rama0.3
module load GCC/12.2.0
module load OpenMPI/4.1.4
module load Transformers/4.30.2
```

Install Python packages:

```
pip install gym==0.10.0
pip install imageio
pip install protobuf==3.20.3
pip install torch==1.13.1
```

Launch execution:

```
sbatch -p compile runner.sh    (10H)
sbatch -p gpu runner.sh         (7d)
```

Manage and monitor jobs:

To see jobs that have been scheduled but have not yet started running:

```
squeue     start
```

To see only the jobs that are currently running:

```
squeue −t RUNNING
```

To list your running or pending jobs:

```
squeue
```

To immediately remove a job from the job queue or also if is currently running:

```
scancel <jobID>
```

See .txt files:

```
tail <name.txt> −n <numlines>
```

When you execute the command `tail -n <numlines> <name.err>`, you'll see various parameters displayed:

- **Percentage:** Shows the percentage of the total training completed.
- **<num>/<numlines>:** The current step within the episode/the total number of planned steps for the training.
- **<time> < <a bigger time> :** the elapsed time since the start of the training < the estimated remaining time until the training is complete
- **<num>it/s :** This is the average iteration speed, indicating how many iterations are completed per second.
- **avg_rew = <num> :** This is the average reward obtained so far during the episode, indicating performance. Negative values suggest poor performance.
- **avg_err= <num> :** This is the average error metric for the current episode, reflecting the performance of the model. Lower values typically indicate better performance.

Copy files:

Commands for copying a directory from your computer to Drago, unzipping it, and then removing the ZIP file:

- Compress the Directory: On your local machine, compress the folder you want to transfer:

```
Compress-Archive -Path "<source_folder_path>" -
DestinationPath " destination_zip_path>"
```

  Example:

```
Compress-Archive -Path "C:\Users\frago\Desktop\CODICE_NUOVO\
RLforUTrackingUpdate" -DestinationPath "C:\Users\frago\Desktop
\RLforUTrackingUpdate.zip"
```

- Transfer the ZIP File: Use scp to copy the ZIP file to Drago:

```
scp "<local_zip_path>" <remote_user>@<remote_host>:<
remote_destination>
```

  Example:

```
scp "C:\Users\frago\Desktop\RLforUTrackingUpdate.zip" <
username@drago_address>:~/oilspill
```

- Unzip the File on Drago: Log in to Drago, navigate to the destination folder, and unzip the file:

```
ssh <remote_user>@<remote_host>
cd <remote_destination>
unzip <zip_file_name>
```

  Example:

```
ssh <username@drago_address>
cd ~/oilspill
unzip RLforUTrackingUpdate.zip
```

- Remove the ZIP File on Drago (Optional): Navigate to the folder containing the ZIP file and delete it:

```
 ssh <remote_user>@<remote_host>ù
 cd <remote_destination>
rm -r <zip_file_name>
```

Example:

```
 ssh <username@drago_address>
 cd ~/oilspill
rm -r RLforUTrackingUpdate.zip
```

TensorBoard:

How to run TensorBoard on a remote server like Drago and access it from your local computer? You need to perform SSH tunneling to redirect the TensorBoard port from the server to your computer. Here's how to do it:

- Log in to your Drago server via SSH writing this in the Prompt, Then write the password.
- Launch the job:

```
 cd oilspill
 source oilspill_env/bin/activate
 source oilspill_env9/bin/activate
 cd RLforUTracking
 cp test_configuration.txt test1inGPU.txt
 ls     make sure that there is the copy of the first
configuration file
```

- After copying it:

```
 vim test1inGPU.txt     I can remove the    #    from the
algorithm that I want to use
 After modifying the new configuration file:
 cd ..
 vim runner.sh     change the old configuration file with the
new one (in the last white row)     test1inGPU
```

- Launch the updated runner.sh:

```
sbatch -p compile runner.sh  OR sbatch -p gpu runner.sh
Make sure that the job was submitted properly:
squeue -t RUNNING
```

Now, you are in your environment and you can run TensorBoard specifying the port you want to run it on.

- First you have to load the tensorflow module:

```
module load rama0.2
module load GCC/10.2.0
module load OpenMPI/4.0.5
module load TensorFlow/2.4.1
```

- Then you can launch tensorboard. For example, you can use port 6006 (which is the default port for TensorBoard):

```
tensorboard --logdir=<path_to_your_log_directory> --host
=127.0.0.1 --port=<your_desired_port>
```

  Make sure to replace <path_to_your_log_directory> with the actual path where your event files are located and <your_desired_port> with the port number you want to use.

  In our case the logs are with the initial folder, not inside the oilspill:

```
tensorboard --logdir=logs/RLforUTracking/[CONFIG FILE NAME]/
log --host=127.0.0.1 --port=6006
```

```
tensorboard --logdir=logs/RLforUTracking/test_configuration/
log --host=127.0.0.1 --port=6008
```

```
tensorboard --logdir=logs/RLforUTracking/test2inGPU/log --
host=127.0.0.1 --port=6008
```

```
tensorboard --logdir=logs/RLforUTrackingUpdate/
test2_update_incompile/log --host=127.0.0.1 --port=6008
```

- A new round of tests on March 4th:

```
tensorboard --logdir=logs/RLforUTrackingUpdateIvan/
test22_update_inGPU/log --host=127.0.0.1 --port=6001
```

- Open another terminal on your local computer and create an SSH tunnel to redirect the port. Replace username with your username on Drago and drago_address with the server address (it could be an IP address or a hostname):

```
ssh -L 6006:127.0.0.1:6006 <username@drago_address>
```

- After establishing the tunnel, open your browser and go to the address:

```
http://127.0.0.1:6006
http://127.0.0.1:6008
```

You should see the TensorBoard interface and your logs.

Bibliography

[1] Ethem Alpaydin. *Introduction to Machine Learning.* 2nd. The MIT Press, 2010. ISBN: 026201243X.

[2] J.P. Mueller and L. Massaron. *Machine Learning For Dummies.* For dummies. Wiley, 2016. ISBN: 9781119245513.

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

[4] Wenwu Zhu, Xin Wang, and Pengtao Xie. «Self-directed machine learning». In: *AI Open* 3 (2022), pp. 58–70. ISSN: 2666-6510. DOI: `https://doi.org/10.1016/j.aiopen.2022.06.001`. URL: `https://www.sciencedirect.com/science/article/pii/S2666651022000109`.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[6] Sadiku Sani, Ibrahim Amina, Abuhuraira Musa, Muntaka Dahiru, and Muhammad Baballe. «Drawbacks of Traditional Environmental Monitoring Systems». In: *Computer and Information Science* 16 (Aug. 2023), pp. 30–30. DOI: `10.5539/cis.v16n3p30`.

[7] Naga Venkata Sudha Rani Nalakurthi, Ismaila Abimbola, Tasneem Ahmed, Iulia Anton, Khurram Riaz, Qusai Ibrahim, Arghadyuti Banerjee, Ananya Tiwari, and Salem Gharbia. «Challenges and Opportunities in Calibrating Low-Cost Environmental Sensors». In: *Sensors* 24.11 (2024). ISSN: 1424-8220. URL: `https://www.mdpi.com/1424-8220/24/11/3650`.

[8] Abdallah Yussuf Ali Abdelmajeed and Radosław Juszczak. «Challenges and Limitations of Remote Sensing Applications in Northern Peatlands: Present and Future Prospects». In: *Remote Sensing* 16.3 (2024). ISSN: 2072-4292. DOI: `10.3390/rs16030591`. URL: `https://www.mdpi.com/2072-4292/16/3/591`.

[9] Yoonchang Sung, Zhiang Chen, Jnaneshwar Das, Pratap Tokekar, et al. «A survey of decision-theoretic approaches for robotic environmental monitoring». In: *Foundations and Trends® in Robotics* 11.4 (2023), pp. 225–315.

[10] Satpreet H Singh, Floris van Breugel, Rajesh PN Rao, and Bingni W Brunton. «Emergent behaviour and neural dynamics in artificial agents tracking odour plumes». In: *Nature machine intelligence* 5.1 (2023), pp. 58–70.

[11] Lingxiao Wang and Shuo Pang. «Autonomous underwater vehicle based chemical plume tracing via deep reinforcement learning methods». In: *Journal of Marine Science and Engineering* 11.2 (2023), p. 366.

[12] Ivan Masmitja, Mario Martin, Tom O'Reilly, Brian Kieft, Narcıs Palomeras, Joan Navarro, and Kakani Katija. «Dynamic robotic tracking of underwater targets using reinforcement learning». In: *Science robotics* 8.80 (2023), eade7811.

[13] K. L. et al. Baker. «Algorithms for Olfactory Search Across Species». In: *Journal of Neuroscience* 38 (2018), pp. 9383–9389.

[14] I. J. et al. Park. «Neurally Encoding Time for Olfactory Navigation». In: *PLoS Computational Biology* 12 (2016), e1004742.

[15] F. Singh, F. van Breugel, S. Rao, and B. Brunton. «Emergent Behaviour and Neural Dynamics in Artificial Agents Tracking Odour Plumes». In: *Nature Machine Intelligence* 5 (2023), pp. 112–124.

[16] J. S. Kennedy and D. Marsh. «Pheromone-Regulated Anemotaxis in Flying Moths». In: *Science* 184 (1974), pp. 999–1001.

[17] J. S. Kennedy. «Zigzagging and Casting as a Programmed Response to Wind-Borne Odour: A Review». In: *Physiological Entomology* 8 (1983), pp. 109–120.

[18] T. C. Baker. «Upwind Flight and Casting Flight: Complementary Phasic and Tonic Systems Used for Location of Sex Pheromone Sources by Male Moths». In: *Proceedings of the 10th International Symposium on Olfaction and Taste* (1990), pp. 18–25.

[19] R. Pang, F. van Breugel, M. Dickinson, J. A. Riffell, and A. Fairhall. «History Dependence in Insect Flight Decisions During Odor Tracking». In: *PLoS Computational Biology* 14 (2018), e1005969.

[20] K. Rajan and L. F. Abbott. «Eigenvalue Spectra of Random Matrices for Neural Networks». In: *Physical Review Letters* 97 (2006), p. 188104.

[21] S. Vyas, M. D. Golub, D. Sussillo, and K. V. Shenoy. «Computation Through Neural Population Dynamics». In: *Annual Review of Neuroscience* 43 (2020), pp. 249–275.

[22] S. Saxena and J. P. Cunningham. «Towards the Neural Population Doctrine». In: *Current Opinion in Neurobiology* 55 (2019), pp. 103–111.

[23] J. D. Seelig and V. Jayaraman. «Neural Dynamics for Landmark Orientation and Angular Path Integration». In: *Nature* 521 (2015), pp. 186–191.

[24] J. et al. Green. «A Neural Circuit Architecture for Angular Integration in Drosophila». In: *Nature* 546 (2017), pp. 101–106.

[25] T. S. Okubo, P. Patella, I. D'Alessandro, and R. I. Wilson. «A Neural Network for Wind-Guided Compass Navigation». In: *Neuron* 107 (2020), 924–940.e18.

[26] D. Grünbaum and M. A. Willis. «Spatial Memory-Based Behaviors for Locating Sources of Odor Plumes». In: *Movement Ecology* 3 (2015), p. 11.

[27] M. Demir, N. Kadakia, H. D. Anderson, D. A. Clark, and T. Emonet. «Walking Drosophila Navigate Complex Plumes Using Stochastic Decisions Biased by the Timing of Odor Encounters». In: *eLife* 9 (2020), e57524.

[28] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis. «Conditional Variable Importance for Random Forests». In: *BMC Bioinformatics* 9 (2008), p. 307.

[29] John Heidemann, Milica Stojanovic, and Michele Zorzi. «Underwater sensor networks: applications, advances and challenges». In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 370.1958 (2012), pp. 158–175.

[30] KL Smith Jr, AD Sherman, PR McGill, RG Henthorn, J Ferreira, TP Connolly, and CL Huffard. «Abyssal Benthic Rover, an autonomous vehicle for long-term monitoring of deep-ocean processes». In: *Science Robotics* 6.60 (2021), eabl4925.

[31] Ivan Masmitja et al. «Mobile robotic platforms for the acoustic tracking of deep-sea demersal fishery resources». In: *Science Robotics* 5.48 (2020), eabc3701.

[32] Enrica Zereik, Marco Bibuli, Nikola Mišković, Pere Ridao, and António Pascoal. «Challenges and future trends in marine robotics». In: *Annual Reviews in Control* 46 (2018), pp. 350–368.

[33] S Revindran. «Underwater robot can follow marine organisms over record distances». In: *Nature* 10 (2010).

[34] Dana R Yoerger et al. «A hybrid underwater robot for multidisciplinary investigation of the ocean twilight zone». In: *Science Robotics* 6.55 (2021), eabe1901.

[35] Aya Saad et al. «Advancing ocean observation with an ai-driven mobile robotic explorer». In: *Oceanography* 33.3 (2020), pp. 50–59.

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

[37] Igor Mordatch and Pieter Abbeel. «Emergence of Grounded Compositional Language in Multi-Agent Populations». In: *arXiv preprint arXiv:1703.04908* (2017).

[38] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. «Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments». In: *Neural Information Processing Systems (NIPS)* (2017).

[39] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *nature* 518.7540 (2015), pp. 529–533.

[40] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. «Continuous control with deep reinforcement learning». In: *arXiv preprint arXiv:1509.02971* (2015).

[41] Bohao Li and Yunjie Wu. «Path planning for UAV ground target tracking via deep reinforcement learning». In: *IEEE access* 8 (2020), pp. 29064–29074.

[42] Lingheng Meng, Rob Gorbet, and Dana Kulić. «Memory-based deep reinforcement learning for pomdps». In: *2021 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2021, pp. 5619–5626.

[43] Denis Yarats and Ilya Kostrikov. *Soft Actor-Critic (SAC) implementation in PyTorch*. `https://github.com/denisyarats/pytorch_sac`. 2020.

[44] Phil Tabor. *SAC Implementation for Reinforcement Learning*. 2019. URL: `https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/PolicyGradient/SAC/sac_torch.py`.

[45] Pranav Shyam. *PyTorch Soft Actor-Critic (SAC) Repository*. 2018. URL: `https://github.com/pranz24/pytorch-soft-actor-critic/blob/master/sac.py`.

[46] Stack Overflow Community. *Create Random Shape Contour Using Matplotlib*. Online forum post. 2018. URL: `https://stackoverflow.com/questions/50731785/create-random-shape-contour-using-matplotlib`.

[47] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd. Springer, 2008. ISBN: 978-3-540-77973-5.

[48] Mike Kamermans. *A Primer on Bézier Curves*. Online article. 2013. URL: `https://pomax.github.io/bezierinfo/`.

[49] Shapely Developers. *Shapely Documentation*. Online documentation. 2025. URL: `https://shapely.readthedocs.io`.

[50] S. Siahposhha. «Approximation Methods for Quadratic Bézier Curve by Circular Arcs». In: *Plus Journal* (2021). URL: `https://www.plus.ac.at/wp-content/uploads/2021/02/Paper_Siahposhha.pdf`.

[51] MatecDev. *How to Check if a Point is Inside a Polygon in Python*. 2023. URL: `https://www.matecdev.com/posts/point-in-polygon.html#how-to-check-if-a-point-is-inside-a-polygon-in-python`.

[52] Abel Inobeme et al. «Chemical Sensor Technologies for Sustainable Development: Recent Advances, Classification, and Environmental Monitoring». In: *Advanced Sensor Research* 3.12 (2024), p. 2400066.

[53] Stack Overflow Community. *Distance Outside Shapely Polygon*. 2018. URL: `https://stackoverflow.com/questions/53882074/distance-outside-shapely-polygon`.

[54] S. Park et al. «Vision-Based Detection and Distance Estimation of Micro Unmanned Aerial Vehicles». In: *Sensors* 15.9 (2015), pp. 23805–23833. DOI: `10.3390/s150923805`.