MULTI-AGENT REINFORCEMENT LEARNING FOR COOPERATIVE AUTONOMOUS NAVIGATION AND PURSUIT IN DRONE SWARMS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2025

Author: Iva Jorgusheska
Project Supervisor: Dr. Wei Pan

Department of Computer Science

Contents

Word Count: 11895

List of Tables

List of Figures

Abstract

MULTI-AGENT REINFORCEMENT LEARNING FOR COOPERATIVE AUTONOMOUS NAVIGATION AND PURSUIT IN DRONE SWARMS

Author: Iva Jorgusheska

Project Supervisor: Dr. Wei Pan

A dissertation submitted to The University of Manchester
for the degree of Bachelor of Science, 2025

Autonomous drone swarms are an emerging and rapidly advancing technology with impactful applications in search and rescue, surveillance, terrain-inaccessible delivery, and military operations. Multi-Agent Reinforcement Learning (MARL), particularly actor-critic methods, has demonstrated strong potential in enabling decentralized coordination among agents. However, these methods often require significant computational resources and training time. This thesis investigates the use of Proximal Policy Optimization (PPO), a foundational actor-critic algorithm, as a lightweight alternative to more specialized multi-agent variants, evaluating its potential for cooperative drone navigation and pursuit.

We develop a complete MARL framework from scratch, including a custom simulation environment composed of four obstacle-rich layouts with increasing difficulty, a PPO training pipeline with modular actor-critic architecture, memory management, and visualization tools for real-time episode rendering. The trained policy is deployed both in simulation and on physical Crazyflie 2.1 drones for real-world testing. The task involves three pursuer drones learning to capture an evading target under three configurations: standard PPO with partial observability, PPO with full global observability, and a fine-tuned PPO that transitions from full to partial observability mid-training.

The fine-tuned PPO consistently outperforms the other configurations, achieving a 77.3% success rate in an unseen environment, compared to just 27.3% for the standard PPO. These results show that with smart adjustments during training, traditional PPO, kept lightweight and efficient, can remain competitive with more complex and resource-intensive multi-agent architectures.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Chapter 1

Introduction

1.1 Context and Motivation

Drones, or Unmanned Aerial Vehicles (UAVs), are among the most rapidly advancing autonomous technologies of the 21st century [18]. Their versatility, cost-effectiveness, and ability to operate in hazardous or hard-to-reach environments make them invaluable across a range of domains, from agriculture and logistics to healthcare, surveillance, and defense [3]. As their deployment expands, there is a growing demand for UAVs that can navigate autonomously and reliably in dynamic and uncertain settings.

A particularly powerful advancement in UAV technology is the use of **drone swarms**, collections of UAVs working together to achieve shared goals without relying on centralized control [1]. These systems offer improved scalability, fault-tolerance, and coverage efficiency compared to single-agent platforms. In domains such as search and rescue, disaster relief, and environmental monitoring, swarm-based coordination enables decentralized decision-making and large-scale adaptability [5]. Inspired by collective behaviors in biological systems, these swarms operate using local observations and limited communication to perform complex tasks cooperatively. [5]

One important and challenging use case in this domain is the **multi-agent pursuit-evasion problem**, where several pursuer drones must coordinate to intercept an evading drone. This task has practical applications in surveillance, airspace security, border monitoring, and defense. It is also widely studied in simulated environments and games, where AI agents are trained to learn competitive or cooperative strategies to outperform human or algorithmic adversaries. The difficulty of this problem increases when the evader drone is significantly faster, requiring the pursuers to leverage advanced planning, predictive modeling, and collaborative decision-making.

Traditional centralized approaches to multi-agent control face limitations in real-world settings due to latency, communication constraints, and single points of failure [27]. In contrast, **decentralized autonomy**, where each drone acts based only on local sensing and a learned policy offers higher scalability and robustness [21]. Achieving reliable cooperation in such decentralized systems, however, remains a challenging problem.

Reinforcement Learning (RL) has emerged as a leading approach for training autonomous agents in both single and multi-agent scenarios. In RL, agents learn behavior through interaction with an environment, improving their policies via trial and error. **Multi-Agent Reinforcement Learning (MARL)** extends this to cooperative or competitive multi-agent systems, but it introduces new challenges such as partial observability and environment non-stationarity caused by multiple learning agents.

The observation range of the agents defines the extent to which each drone is aware of its surroundings, including the positions of teammates and the target. **Partial observability implies** that agents have access only to local information within a limited sensing radius, lacking full knowledge of the global environment. In simulated environments, this parameter can be flexibly configured to explore various perception models. **Full observability** assumes that each agent continuously knows the precise locations of all other agents, both teammates and the evader, at all times. A **hybrid** approach involves initializing training under one observability regime and concluding under another, encouraging adaptability to changing sensory inputs.

Among RL algorithms, **Proximal Policy Optimization (PPO)** is widely regarded for its balance of stability, performance, and simplicity. Although originally designed for single-agent settings, studies such as Chao Yu et al. [29] have shown that PPO can perform competitively in multi-agent scenarios. More specialized variants like MAPPO adapt PPO specifically for multi-agent tasks by introducing centralized training with decentralized execution. However, MAPPO's complexity and computational requirements can be prohibitive in resource-constrained applications.

The **motivation** behind this thesis is to explore how smart, targeted modifications in the PPO training process can enhance its performance in multi-agent drone pursuit tasks, potentially narrowing the performance gap with MARL-specific methods like MAPPO, while maintaining PPO's relative simplicity and training efficiency. Unlike traditional approaches that rely heavily on centralized architectures or communication-intensive methods, this work seeks to identify minimal yet effective policy adjustments that promote decentralized cooperation under varying observability conditions.

An important aspect of this work is the ability to flexibly create and modify custom environments for training, testing, and inference. In practice, evaluating agents across diverse scenarios is essential for understanding their generalization capabilities. This project introduces a user-friendly script for generating customizable 2D environments tailored to pursuit-evasion tasks.

This tool supports experimentation by allowing researchers to test agents in environments different from those they were trained in, an essential step for evaluating robustness, transferability, and real-world applicability.

This thesis focuses on training a team of three pursuer drones to autonomously capture a fourth, faster evader drone in custom environments with different obstacles layouts. The task is inspired in part by the HOLA system [16], which investigates real-time zero-shot coordination for catching an evader drone. This thesis describes the development of a full MARL pipeline using PPO, incorporating environment design, agent architecture, training dynamics, simulation scripts and evaluation. It investigates how different levels of observability, ranging from localized perception to partial or global information sharing, affect the coordination and effectiveness of the swarm.

The implemented codebase is designed to be highly modular and user-friendly, allowing other researchers to easily adjust the perception range and visualize when agents detect the evader and relay that information to teammates. This supports reproducible experimentation and fine-grained control over the training conditions.

Furthermore, a goal of this project is to bridge simulation with real-world use. Unlike many models limited to virtual environments, the policies here are also tested on Crazyflie 2.1 drones. Using their optical flow sensors, the drones perform real-time inference and localization, allowing us to evaluate behavior in real settings.

1.2 Aims and Objectives

To investigate decentralized coordination in autonomous drone swarms using reinforcement learning, this thesis aims to build a complete experimental pipeline based on the Proximal Policy Optimization (PPO) algorithm. The following key objectives structure the development process:

- **Design and implement a custom multi-agent simulation environment from scratch**, including:

  - A fully original environment generation script to enable easy creation, modification, and testing of 2D pursuit-evasion scenarios.
  - Four environments with different obstacle layouts to test generalization capabilities.
  - Support for adjustable environment parameters (number of pursuers/evaders, perception radius, movement speed) to ensure scalability.

- **Build a PPO-based neural network architecture**, consisting of:
  - An **actor network** that outputs an action distribution for the agent.
  - A **critic network** that estimates the value function for stable learning.

- **Develop and train the PPO agents (`AgentPPO`)** with configurable hyperparameters, including:
  - Generalized Advantage Estimation (GAE) for reduced variance in advantage computation.
  - Clipped objective function to ensure stable policy updates.
  - Entropy regularization to maintain exploration throughout training.
  - Minibatch gradient descent using the Adam optimizer for efficiency.

- **Design and implement three training configurations** to simulate varying degrees of decentralization:
  - `classic`: Purely decentralized behavior with local perception only.
  - `full_obs`: Full observability of teammates and evader.
  - `fine_tune`: Starts with full observability and later switches to local perception.

- **Train the PPO agents**
  - Conduct hyperparameter tuning and fine-tuning to optimize performance.
  - Evaluate robustness and generalization across unseen environments.

- **Create visualization and analysis tools**:
  - Enable full-episode export/saving and visualization, including final pursuit and capture states.

- **Deploy the trained PPO policy on physical Crazyflie 2.1 drones**:
  - Integrate with a real-time testing framework.

1.3 Evaluation

The performance of the trained models will be assessed using the following core metrics:

- **Success Rate:** The proportion of episodes in which the evading drone is successfully captured.
- **Collision Rate:** The proportion of episodes in which drones collide with obstacles or with each other.
- **Average Steps:** The average number of steps per episode before reaching a terminal state (capture or failure).

Each trained model is tested on all four environment setups: the one it was trained on and three new, unseen ones. This helps us get a better idea of how well the models perform both in familiar situations and when faced with something completely different. These metrics were chosen to strike a balance between overall task success and behavioral quality. Success rate captures goal completion, but average steps help reveal if the drones are truly pursuing the evader or just stalling to avoid failure. Collision rate reflects how well the agents are coordinating in tight, dynamic spaces, an important factor in real-world deployment. Similar metrics are commonly used in related MARL tasks to evaluate both effectiveness and safety [7] [10].

1.4 Report Structure

The report is organised as follows:

- **Chapter 1:** Introduces the project by setting it within its broader context, outlining the main objectives, presenting the key achievements, and defining the evaluation metrics used throughout the study.
- **Chapter 2:** Presents the necessary technical background to support the project, reviewing foundational concepts and surveying recent advancements and relevant work in the field of multi-agent reinforcement learning.

- **Chapter 3:** Provides a comprehensive description of the entire development process, covering the design and implementation of the neural networks, agent policies, and simulation environments, as well as the overall system integration.
- **Chapter 4:** Describes the training procedures, including different environment variants, evaluation methodologies, and performance assessments conducted both in a custom-built simulation and in real-world-inspired scenarios.
- **Chapter 5:** Summarises the project outcomes, critically discusses observed limitations, and proposes potential future directions for extending and improving upon the current work.

Chapter 2

Background and Theory

In this chapter I outline the key theoretical concepts and technical foundations underlying the project. Starting from core ideas in reinforcement learning and multi-agent systems, it builds toward the specific challenges of decentralised drone coordination and pursuit-evasion. Relevant research is also reviewed to contextualize the project within the broader field.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a fundamental paradigm within machine learning where an agent learns to make sequential decisions through interaction with an environment. Unlike supervised learning, where the model learns from labeled data, or unsupervised learning, which finds hidden patterns in data, reinforcement learning is driven by feedback from the environment in the form of rewards. The central goal is to learn a policy that enables the agent to maximize cumulative rewards over time through trial-and-error interactions [20].

In the context of autonomous drones, RL provides a powerful framework for enabling agents to learn complex behaviors such as coordination, navigation, and pursuit in dynamic and uncertain environments without requiring explicit programming or handcrafted rules.

2.1.1 Key Components of Reinforcement Learning

Figure 2.1 illustrates the interaction between the agent and its environment. At each time step t, the agent observes the current state s_t, chooses an action a_t, receives a

reward r_t, and transitions to a new state s_{t+1}. This process is continuous and allows the agent to learn from the consequences of its actions. [8]

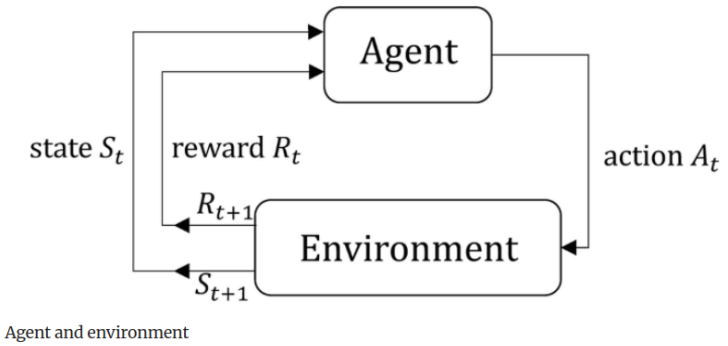


Figure 2.1: Illustration of the RL loop showing agent, environment, state, action, and reward. [8]

The main components involved are:

- **Agent**: The learner or decision-maker. [8]
- **Environment**: The world or system the agent interacts with.[8]
- **State** (s): The environment configuration at a given point.[8]
- **Action** (a): A decision the agent makes to affect the environment.[8]
- **Reward** (r): Scalar feedback from the environment evaluating the performed action.[8]
- **Policy** (π): A function that maps states to actions, representing the agent's behaviour strategy.[8]
- **Trajectory** (τ): A sequence of state-action-reward transitions that the agent experiences during an episode.[8]

2.1.2 Markov Decision Process Framework

Reinforcement learning problems are typically modeled as Markov Decision Processes (MDPs), which provide a mathematical formalization of decision making. [**?**]. An MDP is defined by the tuple (S, A, P, R, γ), where:

- S: Set of possible states

- A: Set of available actions
- $P(s'|s,a)$: Transition probability function
- $R(s,a)$: Reward function
- γ: Discount factor for future rewards

2.1.3 Rewards and Return

The reward r_t serves as an essential feedback signal for guiding the agent's behavior. While immediate rewards can inform short-term decisions, RL aims to maximize the *return* G_t, defined as the total discounted reward over time:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where $\gamma \in [0,1]$ is the discount factor that determines the importance of future rewards. A lower γ places emphasis on immediate rewards, while a higher γ values long-term outcomes. This formulation allows agents to balance short-term gains with long-term performance. [8]

2.1.4 Exploration vs Exploitation

A central challenge in reinforcement learning is the *exploration-exploitation trade-off* as illustrated in Figure 2.2. On one hand, the agent must *exploit* known information to maximize rewards. On the other, it must *explore* new actions to discover potentially better strategies. Balancing these objectives is essential for effective learning [22].

2.1.5 Deterministic vs Stochastic Transitions

The transition between states can be either deterministic or stochastic. In a *deterministic* environment, the next state s_{t+1} is fully determined by the current state s_t and action a_t. In contrast, a *stochastic* environment introduces uncertainty, where s_{t+1} is sampled from a probability distribution $P(s_{t+1}|s_t,a_t)$.

Stochastic environments are more realistic for drone coordination and swarm scenarios, where external disturbances, sensor noise, or unpredictable elements may influence the state transitions. Designing policies that are robust to such randomness is a critical part of reinforcement learning in practical settings [24].

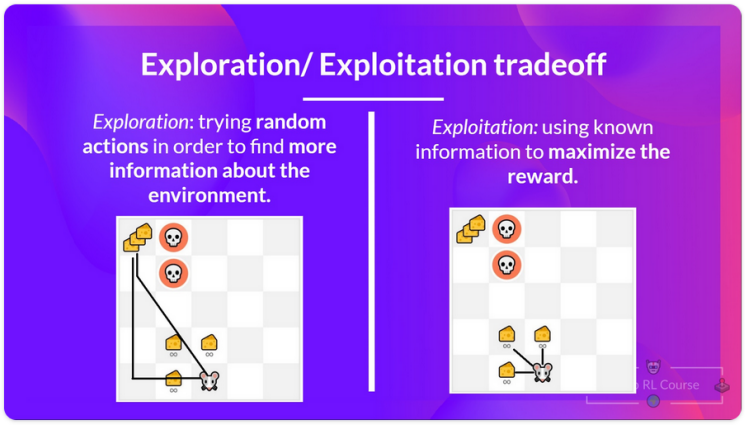


Figure 2.2: The Exploration/Exploitation trade-off visualisation[9]

2.1.6 Episodes and Trajectories

A *trajectory* (or episode) in reinforcement learning is a complete sequence of interactions from the initial state to a terminal state. This sequence includes all states, actions, and rewards experienced by the agent:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \dots)$$

Episodes can be finite or infinite. In continuous control tasks, episodes may terminate after a fixed number of steps. Each trajectory provides training data for learning the policy, and aggregating experience over many episodes helps the agent generalize better. [8]

2.2 Multi-Agent Reinforcement Learning (MARL)

While single-agent reinforcement learning has shown significant progress, many real-world problems involve multiple agents interacting in a shared environment. Examples are swarm robotics and drone coordination, where multiple autonomous agents must learn to act collaboratively or competitively. Multi-Agent Reinforcement Learning (MARL) extends traditional RL into this multi-agent domain, introducing unique challenges and requiring novel algorithmic strategies [4]. Core challenge in MARL is **partial observability**. Each agent often has access only to a local view of the environment, leading to partial observability. This limitation hinders coordination, as agents must act under uncertainty about the global state.[32]

Two primary training paradigms are common in MARL: Decentralised Training and Execution (DTE) and Centralised Training with Decentralised Execution (CTDE).

2.2.1 Decentralised Training and Execution (DTE):

It is characterized by each agent learning its policy independently, using only its local observations, actions, and shared rewards. This is highly relevant to autonomous drone swarms, where communication constraints and scalability challenges make centralised approaches impractical. DTE methods allow each drone (agent) to learn in isolation, making the system more robust and flexible in dynamic or partially observable environments.

Value-Based Decentralised Methods

One of the foundational approaches in DTE is *Independent Q-Learning (IQL)*. In IQL, each agent learns its own Q-function, denoted as $Q_i(h_i, a_i)$, based solely on its local action a_i and observation history h_i, without modeling the behavior of other agents. Although simple and widely used, IQL can suffer from instability and poor convergence due to the non-stationarity introduced by concurrently learning agents. Nevertheless, it provides a baseline for understanding decentralised learning and has influenced many subsequent methods.[2]

Policy Gradient and Actor-Critic Methods in DTE

To overcome the limitations of value-based approaches, *policy gradient* methods have been adapted to the decentralised setting. These methods directly optimize stochastic policies and are well-suited to environments with continuous action spaces, such as those encountered in aerial pursuit tasks.

These methods directly optimize a stochastic policy by ascending the gradient of expected rewards. In the decentralised setting, a foundational approach is *Independent REINFORCE*, a multi-agent adaptation of the Monte Carlo policy gradient method. Here, each agent independently samples full-episode rollouts, estimates its own policy gradient, and updates its parameters accordingly. Although conceptually simple and theoretically sound, REINFORCE suffers from high variance and sample inefficiency, limiting its practical use in real-time or complex multi-agent tasks.

A more practical approach is the *Independent Actor-Critic (IAC)* method, in which each agent maintains both a policy (actor) and a value function (critic). The critic

evaluates the policy's performance and provides feedback to update the actor incrementally, leading to more sample-efficient learning. This structure enables real-time learning and adaptation, which is crucial in dynamic multi-agent environments.

One of the most effective implementations of the actor-critic framework in decentralised MARL is *Proximal Policy Optimization (PPO)*. Although originally developed for single-agent reinforcement learning, PPO can be applied independently to each agent in a DTE setting. Each agent learns its own policy and value function, using PPO's clipped surrogate objective to ensure stable and constrained policy updates. When extended with recurrent neural networks to handle partial observability, PPO becomes especially suitable for swarm robotics tasks. [2]

2.2.2 Centralised Training for Decentralised Execution

Centralised Training for Decentralised Execution (CTDE) is a widely used framework in multi-agent reinforcement learning (MARL). The key idea is that during training, agents can use centralised information (like the global state or other agents' actions), but during execution, they must act independently based only on their local observations. This setup helps with learning coordinated behaviors while still allowing decentralised policies at test time. [31] CTDE encompasses a broad spectrum of approaches that differ in the extent of centralised information used during training. Some methods introduce limited centralised components such as shared critics or communication models—into an otherwise decentralised learning process. Others take a more centralised approach, training joint policies with full state information before attempting to decompose them into decentralised policies for execution. In practice, most CTDE algorithms strike a balance between these extremes, leveraging centralised insights to enhance training while preserving decentralised autonomy at execution time [2].

Value-Based CTDE Methods

One common group of CTDE methods is based on learning a joint Q-function and then factoring it into individual Q-values for each agent. These methods assume that the joint Q-function can be approximated by combining the local Q-values:

$$Q_{tot}(h,a) = f(Q_1(h_1,a_1),\ldots,Q_n(h_n,a_n))$$

Here, f is a mixing function that is trained using centralized information but ensures that each agent's policy depends only on its own observation and action. This allows

the agents to act independently at execution time. [2]

Policy Gradient CTDE Methods

Major group of methods uses centralised critics during training. These are actor-critic approaches where each agent has its own actor (policy), and the training signal comes from a centralised critic that has access to more information. These methods are particularly useful because they can stabilize training in multi-agent environments and handle continuous or stochastic policies more easily than value-based methods. [2]

More recently, PPO has been extended to multi-agent settings using CTDE. In MAPPO [29], each agent has its own actor, and a centralised critic is used during training to compute the advantage estimates. Once training is done, the critic is discarded, and only the decentralised actors are used during execution.

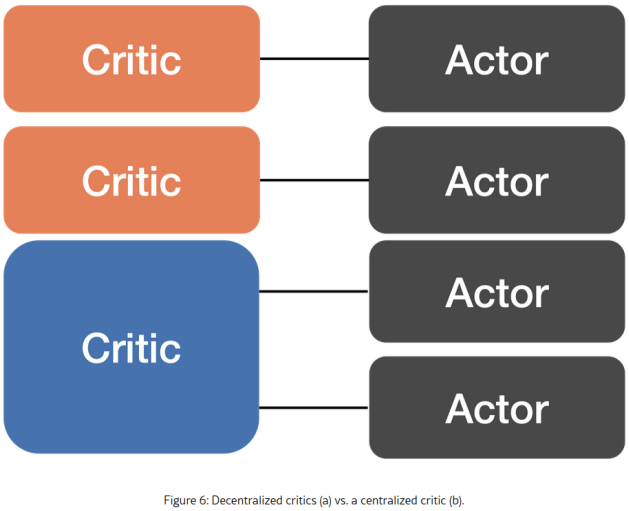


Figure 2.3: Centralised vs Decentralised critic[2]

Figure 2.3 illustrates the distinction between decentralised and centralised critic architectures. In the top two rows, each agent has its own critic, which evaluates the policy based solely on that agent's local observation. This setup is fully decentralised. In contrast, the bottom row represents the centralised training setup where a shared centralised critic is trained using information from all agents to compute more accurate advantage estimates. During execution, this centralised critic is discarded, and each agent relies only on its own independent actor, making the policy fully decentralised in deployment.

While Proximal Policy Optimization (PPO) is originally designed for single-agent

reinforcement learning, it serves as a strong baseline for decentralised multi-agent scenarios when applied independently to each agent. In this project, I adopt a fully decentralised PPO setup, where each agent learns its own policy and maintains a local critic, without access to global state or teammate information. Although this setting omits the benefits of centralised critics used in frameworks like MAPPO [29], it allows for a clean examination of how decentralised policies perform in isolation. The goal is to explore how performance can be enhanced through architectural or observational tweaks (e.g., partial observability-aware mechanisms), and subsequently compare the results to those obtained using MAPPO.

These methods are particularly useful because they can stabilize training in multi-agent environments and handle continuous or stochastic policies more easily than value-based methods. [2]

Information Sharing in Partially Observable Environments

One of the most significant challenges in multi-agent systems is learning under partial observability, where each agent only perceives a limited view of the environment. Recent work [17] has formalized the study of information sharing in Partially Observable Stochastic Games (POSGs), showing that under certain assumptions, sharing common information among agents can significantly reduce the computational complexity of planning and learning.

2.3 Navigation and Pursuit in Drone Swarms

Drones, or Unmanned Aerial Vehicles (UAVs), have become increasingly prevailing in domains such as mapping, inspection, delivery, and disaster response due to their agility, scalability, and ability to operate in challenging environments. As individual drone capabilities mature, there is growing interest in deploying drone swarms.

Autonomous drone swarms represent a growing frontier in robotics and control systems, where multiple Unmanned Aerial Vehicles (UAVs) coordinate to achieve shared objectives without centralised control. Inspired by biological systems in the nature such as flocks of birds or swarms of insects, these systems offer enhanced fault-tolerance, and spatial efficiency [6]. Applications range from search and rescue, and environmental monitoring to surveillance, area coverage and cooperative transportation.

However, designing effective decentralised control mechanisms for drone swarms remains a significant challenge, particularly in dynamic and partially observable environments. Each drone must reason and act based on local observations and, often bandwidth-constrained communication.

A particularly challenging and insightful scenario within swarm robotics is the **pursuit** task, where a team of drones (pursuers) attempts to capture one or more evasive targets. These scenarios test not only navigational agility but also strategic coordination under uncertainty. Pursuit tasks are well-suited to study the emergence of collaborative behavior, especially when each agent has only partial information about the environment and limited access to teammate states or actions. This makes them an ideal benchmark for evaluating decentralised policies, communication strategies, and learning frameworks in real-world inspired drone missions. The specific pursuit-evasion setup in this project is inspired by the HOLA-Drone framework [16] where the objective is for pursuing agents to collaboratively capture evasive targets.

2.3.1 Advancements in Autonomous Drone Swarms

Recent research has explored a variety of methods for learning coordinated behavior in swarms. For example, *Vásárhelyi et al.* [26] demonstrated real-world decentralised flocking using only local relative positioning, showing the feasibility of large-scale coordination without centralised control. In the reinforcement learning domain, *Huttenrauch et al.* [12] proposed a decentralised actor-critic approach tailored to robotic swarms, enabling the emergence of coordinated behavior based solely on local observations.

There has also been interest in learning communication protocols between agents to improve coordination under limited bandwidth. *Singh et al.* [23] proposed a model where agents learn not only their actions but also when and what to communicate, optimizing coordination under strict communication constraints. Similarly, attention-based methods have been applied to enable agents to selectively attend to the most relevant teammates when computing their policies [33].

Overall, autonomous drone swarms offer a rich and practically relevant setting for advancing Multi Agent Reinforcement Learning (MARL). Their inherent constraints, partial observability, local-only sensing, and real-time requirements, pose significant challenges but also unlock opportunities for scalable, adaptive, and intelligent control systems in complex environments.

2.4 Proximal Policy Optimization Algorithm (PPO)

Proximal Policy Optimization (PPO), introduced by Schulman et al. [19], is a widely adopted reinforcement learning algorithm known for its balance between training stability, sample efficiency, and ease of implementation. PPO has an actor-critic architecture: an actor network outputs a stochastic policy, while a critic network estimates the value function to guide learning. This design is particularly well-suited for continuous control tasks, such as autonomous UAV navigation, where smooth and precise actions are crucial. The core innovation of PPO lies in its clipped surrogate objective, which limits the deviation between the new and old policy during updates, avoiding destructive policy changes that can destabilize training. Combined with Generalized Advantage Estimation (GAE) for low-variance, unbiased advantage computation, and entropy regularization to encourage exploration, PPO offers a robust learning framework. As addition to the clipped surrogate objective or as its substitution, the authors proposed penalty on KL divergence as baseline only since they showed it always underperformed when compared to the clipped surrogate objective.

While PPO was originally designed for single-agent problems, it has proven useful in multi-agent reinforcement learning (MARL) too, especially in decentralised setups where each agent learns its own policy using only local observations and rewards. Even though it lacks a centralised critic for coordination, its simplicity and training efficiency make it a practical candidate for testing in MARL tasks, particularly when combined with smart tweaks or staged training strategies. [19]

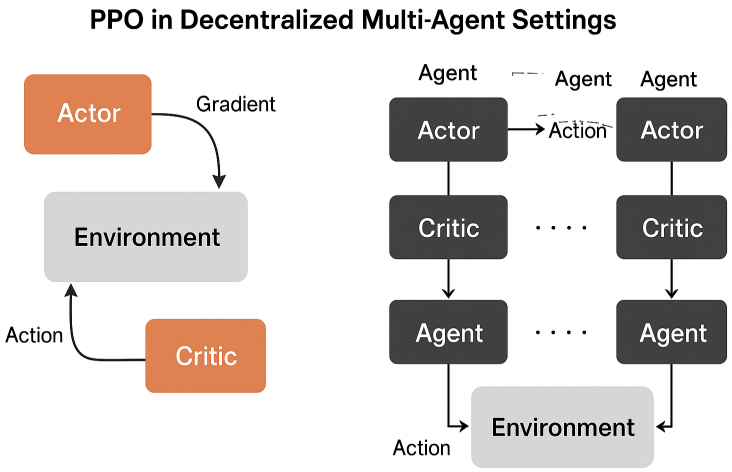


Figure 2.4: Single and multi-agent scenario in decentralised PPO architecture

Figure 2.1 depicts how Proximal Policy Optimization (PPO) operates in a decentralised multi-agent setting. In a single-agent scenario, the actor selects actions based on local observations, and the critic evaluates the resulting outcomes to improve the policy. In a multi-agent environment, each agent maintains its own actor-critic pair, relying solely on local observations and rewards. However, since all agents share the same environment, their actions influence one another indirectly, leading to complex multi-agent dynamics despite the absence of a global critic or centralised coordination.

2.5 Related Work

With the growing interest in autonomous UAVs and swarms of agents working together to catch a moving target, there has been a lot of research, both with physical drones and in simulation environments, focusing on how to best train these systems. Different reinforcement learning (RL) approaches have been explored to handle the complexity of multi-agent pursuit scenarios.

Ye et al. [28] studied a pursuit-evasion game between two spacecraft, where the evader was trained using the Proximal Policy Optimization (PPO) algorithm and the pursuer used Particle Swarm Optimization. Even though the setting involved space rather than drones, the findings are still relevant: PPO helped the evader make quicker and more reactive decisions in close-range situations. This supports our use of PPO for decision-making in drone pursuit tasks.

In a more UAV-specific setup, Kouzeghar et al. [14] proposed a multi-UAV system that used a modified version of MADDPG with different agent roles like scouts and pursuers. They designed a reward system based on Voronoi partitions to help drones cover more space efficiently. Their approach showed strong real-world performance on Crazyflie drones, proving that multi-agent RL can go beyond simulations. While their method focused more on role distribution and spatial coverage, our work shifts focus toward training tweaks in PPO to improve generalization across environments and observation settings.

Several recent papers have focused specifically on modifying the PPO algorithm specifically for cooperative multi-agent reinforcement learning (MARL) scenarios. For example, Zhan et al. [30] introduced a MAPPO (Multi-Agent PPO) algorithm that uses centralised training with decentralised execution. A global critic is used during training to provide more informative feedback to each agent, while each drone still makes decisions based on local observations. Their method showed better coordination and

higher rewards compared to traditional PPO, especially in complex environments like Unity3D air combat simulations. This motivates our work to see how far PPO alone can go with the right training structure, and whether it can compete with centralised approaches like MAPPO.

Other studies focus on real-world constraints, such as those in surveillance or disaster-response scenarios. For example, [10] proposed a distributed deep RL framework for UAV swarms doing tasks like wildlife monitoring. Because onboard processing power is limited, they split neural networks across drones and used coordination to compute shared inferences. These setups highlight the need for lightweight yet robust decision-making frameworks—something PPO can offer due to its simplicity and reliability.

Importantly, several papers have explored PPO on its own, showing that it is a strong baseline even without centralised critics. In [29], PPO and its variants (MAPPO and IPPO) were benchmarked across popular MARL environments like MPE, SMAC, GRF, and Hanabi. Surprisingly, PPO-based methods often outperformed more complex off-policy algorithms like MADDPG and QMix, even with minimal hyperparameter tuning. The authors also highlighted best practices for getting good results with PPO in multi-agent settings, like parameter sharing and GAE. This strengthens the case for using PPO as a main algorithm, not just a baseline.

Chikhaoui et al. [7] also showed how PPO can be applied to train UAVs in complex 3D urban environments, focusing on challenges like collision avoidance and battery-aware routing. Their PPO-trained model navigated dense obstacle fields and rerouted to charging stations when needed, showing that PPO works well even under physical and environmental constraints.

2.6 Summary

Despite all these promising results, there has not been a focused investigation on how PPO performs in a drone swarm pursuit task when trained under different observation setups (global, local, mixed). This is the gap our work aims to fill. We explore how competitive PPO can be against more complex variants like MAPPO by tweaking its training setup and comparing its effectiveness at catching an evader in swarm pursuit missions. Our work contributes to understanding whether PPO, when used creatively, can stand on equal footing with more centralised and complex MARL algorithms.

In this chapter, we outlined the theoretical foundations, key concepts, and recent

advancements relevant to this project. This structured overview provides the necessary background for understanding the technical concepts that underpin this project.

Chapter 3

Methodology

3.1 Overview

This and the next chapter outline the methodology implemented to develop the core deliverables of the project. The thesis follows a structured pipeline covering the entire life cycle of the system, from neural network design to real-world deployment. Each stage is discussed with a focus on the technical implementation, design rationale, and evaluation procedures. The major development stages are:

- **PPO Neural Network Development**
- **PPO Agent Development**
- **Custom Environments Development**
- **Training the PPO Agent**
- **Simulation, Visualization and Evaluation**
- **Real-world deployment**

This chapter covers the PPO neural network and agent development, and the custom environment development.

3.2 PPO Neural Network Development

At the heart of this project is a custom neural network for PPO, built to handle multi-agent pursuit-evasion challenges. It uses an actor-critic setup where both the actor and

critic share early layers that process information about the agent, nearby pursuers, and the evader. The network smartly adapts to different visibility conditions by adjusting its inputs based on what each agent can actually observe. It is implemented through two main functions: one to set up the network layers, and another to run inputs through the model, outputting actions and value estimates.

3.2.1 Network Initialization

This subsection details the initialization process of the `ActorCriticPPO` network, including the architectural structure of the actor and critic branches, and the initialization strategy for weights and biases.

Table 3.1 summarizes the initialization of the `ActorCriticPPO` network and the functionality of its layers as used by the `forward` method.

Layer	Dimensions	Description
1	$(2 \rightarrow 128)$	Encode relative positions of other agents (actor)
2	$(2 \rightarrow 128)$	Encode positions of evaders (actor)
3	$(2 \rightarrow 128)$	Encode agent's own position (actor)
4	$(384 \rightarrow 128)$	Fuse features from targets, teammates, self (actor)
5	$(128 \rightarrow 128)$	Hidden layer for deeper feature processing (actor)
6	$(128 \rightarrow a)$	Output mean of Gaussian action distribution (actor)
logstd	$(1 \times a)$	Learnable log standard deviation (actor)
7	$(2 \rightarrow 128)$	Encode relative positions of other agents (critic)
8	$(2 \rightarrow 128)$	Encode positions of evaders (critic)
9	$(2 \rightarrow 128)$	Encode agent's own position (critic)
10	$(384 \rightarrow 128)$	Fuse features for value estimation (critic)
11	$(128 \rightarrow 128)$	Hidden layer for deeper feature extraction (critic)
12	$(128 \rightarrow 1)$	Output scalar state value estimate (critic)

Table 3.1: Summary of PPO Actor and Critic Network Layers

Weights and Biases

Weight initialization plays a pivotal role in the training of neural networks, significantly impacting training stability, convergence speed, and the model's ability to generalize[15]. In this project, we employed orthogonal initialization for the weights and set the biases to zero across all layers of the Actor-Critic PPO architecture.

Orthogonal Weight Initialization

Orthogonal initialization sets the weight matrices such that their rows or columns are mutually orthogonal, meaning their dot product is zero (i.e., they are perpendicular). In practice, I used the built-in PyTorch function `torch.nn.init.orthogonal_()` to initialize the weights of each layer in the network. This method produces orthonormal vectors—those that are not only orthogonal but also have unit norm (length equal to 1).

Mathematically, a matrix $W \in \mathbb{R}^{m \times n}$ is orthogonal if

$$W^T W = I$$

when $m = n$, where I is the identity matrix. For non-square matrices, the term *semi-orthogonal* is often used, indicating that either the rows or columns are orthonormal.

The main advantage of orthogonal initialization lies in its ability to preserve the norm of input signals across layers, which helps mitigate the issues of vanishing or exploding gradients commonly encountered in deep neural networks. By maintaining gradient scale, this method contributes to more stable and efficient training [11].

Bias Initialization

Each neuron in a neural network includes a bias term to allow flexibility in shifting the activation function. This helps the model fit data more effectively by removing the constraint of passing through the origin. Mathematically, the output of a neuron is computed as:

$$y = f(Wx + b)$$

where W is the weight matrix, x is the input vector, b is the bias term, and f is a non-linear activation function.

All biases in the network were initialized to zero. This choice ensures that, at the start of training, the neuron activations are centered, promoting symmetric learning. It also avoids introducing any unintended bias into the initial output distributions. Since the early network dynamics are primarily governed by the weights, zero bias initialization is a safe and commonly used strategy that supports stable training without compromising the model's ability to learn diverse functions.

3.2.2 Forward Pass Breakdown

The `forward` method in the ``ActorCriticPPO'' class computes both the policy distribution parameters (used for action sampling) and the value estimate from the critic.

Input Structure and Preprocessing

The input tensor contains the agent's own position, the relative positions of teammates, and the absolute positions of evaders.

Actor Network Computation

- **Inter-agent Encoding (`layer1`):**
  The relative positions of other oursuing agents are reshaped into 2D vectors and passed through a shared fully connected layer with `tanh` activation. The output embeddings are averaged to obtain a fixed-size feature vector representing the inter-agent relations:

$$h_{ij} = \frac{1}{N-1} \sum_{k=1}^{N-1} \tanh(\texttt{layer1}(x_{ij}^{(k)}))$$

  where N is the number of agents, and $x_{ij}^{(k)}$ denotes the relative position of the k-th other agent. N - 1 is used since the agent itself is counted in the number of agents N.

- **Evader Encoding (`layer2`):**
  The positions of all evaders are individually processed using another shared fully connected layer with `tanh` activation. The resulting embeddings are averaged to form a fixed-size representation:

$$h_{target} = \frac{1}{M} \sum_{i=1}^{M} \tanh(\texttt{layer2}(x_{target}^{(i)}))$$

  where M is the number of evaders.

- **Self-Position Encoding (`layer3`):**
  The observed agent's own position is passed through a separate fully connected

layer with `tanh` activation:

$$h_{obs} = \tanh(\texttt{layer3}(x_{obs}))$$

- **Feature Concatenation & Transformation:**
  The encoded evaders representation, self-position, and inter-agent vectors (the outputs of the three previous layers) are concatenated and processed through two additional fully connected layers with `tanh` activations:

$$h = \tanh(\texttt{layer4}([h_{target}, h_{obs}, h_{ij}]))$$

$$h = \tanh(\texttt{layer5}(h))$$

- **Mean Action Output (`layer6`):**
  The actor network models a **stochastic policy** by outputting a mean action vector μ, representing the expected action for a given state. This is computed as:

$$\mu = \tanh(\texttt{layer6}(h))$$

  where h is the latent feature representation. The `tanh` activation bounds each action dimension to $[-1, 1]$, aligning with normalized continuous action spaces (e.g., velocity or acceleration). The dimensionality of μ equals `action_dim`.

- **Log Standard Deviation (learnable parameter):**
  To complete the Gaussian policy, a learnable vector of **log standard deviations** ($\log \sigma$) is maintained for each action dimension:

$$\log \sigma \in \mathbb{R}^{\texttt{action\_dim}}$$

  It is expanded to match the batch size during the forward pass. This ensures numerical stability and positivity of standard deviations via exponentiation. The resulting distribution:

$$\pi(a \mid s) = \mathcal{N}(a \mid \mu, \sigma^2)$$

  supports exploration and is fundamental to PPO's policy gradient formulation.

Critic Network Computation

The critic estimates the state value function $V(s)$, reflecting the expected return from a state. While structurally similar to the actor, it uses independent layers and parameters to decouple value estimation from action selection.

- **Inter-agent Encoding (`layer7`):**
  Relative positions of other agents are encoded via a shared fully connected layer with `tanh`, then averaged to summarize local agent context:

$$h_{ij} = \frac{1}{N-1} \sum_{k=1}^{N-1} \tanh(\texttt{layer7}(x_{ij}^{(k)}))$$

- **Evader Encoding (`layer8`):**
  Evader positions are individually encoded and averaged:

$$h_{\text{target}} = \frac{1}{M} \sum_{i=1}^{M} \tanh(\texttt{layer8}(x_{\text{target}}^{(i)}))$$

- **Self-Position Encoding (`layer9`):**
  The agent's own position is passed through a separate fully connected layer:

$$h_{\text{obs}} = \tanh(\texttt{layer9}(x_{\text{obs}}))$$

- **Concatenation and Transformation:**
  The three encoded features are concatenated and processed through two additional `tanh` layers:

$$h = \tanh(\texttt{layer10}([h_{\text{target}}, h_{\text{obs}}, h_{ij}]))$$

$$h = \tanh(\texttt{layer11}(h))$$

- **State Value Output (`layer12`):**
  The final layer outputs a scalar value estimate of the current state:

$$V(s) = \texttt{layer12}(h)$$

Final Outputs of the Network:

The overall forward pass of the model returns the key components used in Proximal

Policy Optimization (PPO):

$$\mu, \log \sigma, V(s)$$

where μ and $\log \sigma$ define the stochastic policy for the actor, and $V(s)$ is the state value function used for advantage estimation and policy updates.

Choice of Activation Function: Why `tanh`

The `tanh` activation was chosen for its ability to output values in $[-1, 1]$, which helps maintain gradient flow and stabilize learning. Its symmetry reduces internal covariate shift, and strong derivatives near zero help avoid vanishing gradients, making it a reliable choice for actor-critic reinforcement learning setups [25].

3.3 PPO Agent Development

To operationalize the actor-critic architecture in a multi-agent pursuit environment, a flexible and extensible `PPOAgent` was developed.

3.3.1 Modular Architecture for Flexibility

Policy and Value Networks.

At the heart of the agent is an `ActorCriticPPO` module that integrates both policy and value networks. This setup enables shared parameters and synchronized updates, streamlining the learning process. An *Adam optimizer* manages network updates, with *MSE loss* guiding the critic's training.

Agent Duplication and Mutation

To support evolutionary strategies and flexible experimentation, two utility functions were introduced:

- `self_copy()`: Duplicates the agent, preserving network weights and hyperparameters.
- `explore_self()`: Creates a perturbed version of the agent by adjusting hyperparameters such as `gae_lambda` or learning rate, useful for population-based training or optimization studies.

Initialization

Agent behavior is controlled via a `config_args` dictionary, allowing dynamic specification of key PPO hyperparameters:

- `batch_size`: The threshold number of transitions to collect before optimization.
- `ppo_epoch`: Number of passes over the data during optimization.
- `c1, c2`: Coefficients for the value loss and entropy bonus in the PPO loss function.
- `gae_lambda`: GAE parameter for smoothing advantages.
- `eps_clip`: Clipping parameter to bound policy updates.
- `learning_rate`: Learning rate for the Adam optimizer.
- `gamma`: Discount factor for future rewards.
- `minibatch_size`: Size of minibatches for each PPO epoch.
- `num_evader`: The number of evader agents in the environment.

3.3.2 Experience Collection

During interaction with the environment, each transition is stored as a flattened vector:

```
[state, action, reward, next_state, done]
```

These are appended to `self.memory` until a predefined batch size is reached. Training is only triggered when `requires_more_data` confirms that sufficient experiences have been gathered.

3.3.3 Action Selection via Stochastic Policies

To choose an action at each timestep, the agent passes the current state through its policy network. The network outputs two key components: the mean μ and the logarithm of the standard deviation $\log \sigma$ of a Gaussian distribution. These parameters define a continuous probability distribution over possible actions.

1. **State Preprocessing**: The input state tensor is reshaped depending on whether it is a single-agent or multi-agent setting. It is then converted to a PyTorch tensor and moved to the appropriate computation device (CPU or GPU).

2. **Network Inference**: The preprocessed state is fed into the policy network (`eval_net`), which returns:

   - μ: the mean of the action distribution
   - $\log\sigma$: the log standard deviation

3. **Sampling the Action**: The agent samples from a Gaussian distribution parameterized by these outputs:

   $$a \sim \mathcal{N}(\mu, \sigma^2) \quad \text{where } \sigma = \exp(\log\sigma)$$

   This sampling introduces stochasticity, allowing the agent to explore different actions rather than always taking the most likely one.

4. **Optional Probability Output**: If the flag `if_probs` is set, the function also computes and returns:

   - The log-probabilities of the sampled action under the current policy
   - The exponentiated probabilities, which may be used in policy gradient updates during training

This mechanism is central to stochastic policy gradient methods like PPO, where both the action and its associated probability are used to compute the surrogate loss. The sampled action is detached from the computation graph and returned as a NumPy array for use in the environment.

3.3.4 Advantage Estimation with GAE

The `calculate_advantage` function estimates the advantage and target values for critic training by processing trajectory data in reverse time order. It leverages Generalized Advantage Estimation (GAE) to compute smoother and more stable advantage estimates over time.

For each transition, the temporal-difference (TD) error is computed as:

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

where r_t is the immediate reward, $V(s_t)$ is the estimated value of the current state, $V(s_{t+1})$ is the estimated value of the next state, and γ is the discount factor.

Using this TD error, the advantage is recursively estimated as:

$$A_t = \delta_t + \gamma\lambda A_{t+1}$$

where $\lambda \in [0,1]$ is the GAE smoothing parameter. This recursive formulation reduces variance while maintaining bias control, effectively balancing the bias-variance trade-off.

Special handling is performed based on the episode termination signal (encoded in the `done` flag). For terminal states, the advantage simplifies to:

$$A_t = r_t - V(s_t)$$

and for goal-reaching transitions, only a single step's future value is considered.

After processing the full trajectory, two tensors are returned:

- **Advantage**: The estimated advantage values A_t, used to weight policy gradient updates.
- **Reference Value (TD Target)**: Computed as $A_t + V(s_t)$, serving as the training target for the critic.

These are reversed to match the original time sequence before being returned to the PPO optimizer.

3.3.5 Log-Probability Computation

The `calculate_log_probs` function ensures numerical stability by handling `NaN` values, constructs a Gaussian distribution, and returns log-probabilities and entropies essential for training.

3.3.6 Policy Optimization with PPO

The `optimize` function performs the core Proximal Policy Optimization (PPO) update, training both the actor and critic components of the agent's policy network using experiences stored in memory.

1. **Memory Preprocessing**: All experience tuples are stored in a replay buffer. These tuples are combined into a single NumPy array and then split into individual tensors:

   - `state` – the current observation
   - `action` – the action taken
   - `reward` – the scalar reward received
   - `state_next` – the observation after taking the action
   - `done` – a binary indicator (1 if the episode ended, 0 otherwise)

2. **Advantage Estimation**: The advantage function measures how good an action was compared to the expected value. It is computed using the current and next states, reward, and done flag. The result is normalized to stabilize learning:

$$A = \frac{A - \mu_A}{\sigma_A + \varepsilon}$$

   where $\varepsilon = 10^{-5}$ to avoid division by zero.

3. **Log-Probability Calculation**: The old action log-probabilities are computed using the current policy network outputs (mean and log standard deviation). These will be used to compute the probability ratio needed for PPO's surrogate loss.

4. **Batching and Mini-Batch Updates**: The dataset is split into mini-batches, and each is used to perform a policy update. For each mini-batch:

   (a) The current policy network outputs the value estimate, mean, and log standard deviation.

   (b) The critic loss is computed as the mean squared error between predicted state values and reference values.

$$\mathcal{L}_{\text{critic}} = \frac{1}{\text{std}(V_{\text{target}})} \cdot \text{MSE}(V(s), V_{\text{target}})$$

   (c) The actor loss is computed using the PPO clipped objective. The probability ratio r is:

$$r = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$$

The surrogate loss is:

$$\mathcal{L}_{\text{actor}} = -\mathbb{E}\left[\min(rA, \text{clip}(r, 1-\varepsilon, 1+\varepsilon)A)\right]$$

(d) The entropy loss is added to encourage exploration:

$$\mathcal{L}_{\text{entropy}} = -\mathbb{E}[H(\pi_\theta)]$$

(e) The total loss combines actor, critic, and entropy losses:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{actor}} + c_1 \cdot \mathcal{L}_{\text{critic}} + c_2 \cdot \mathcal{L}_{\text{entropy}}$$

(f) The gradient is computed and backpropagated. Gradients are clipped to prevent exploding values:

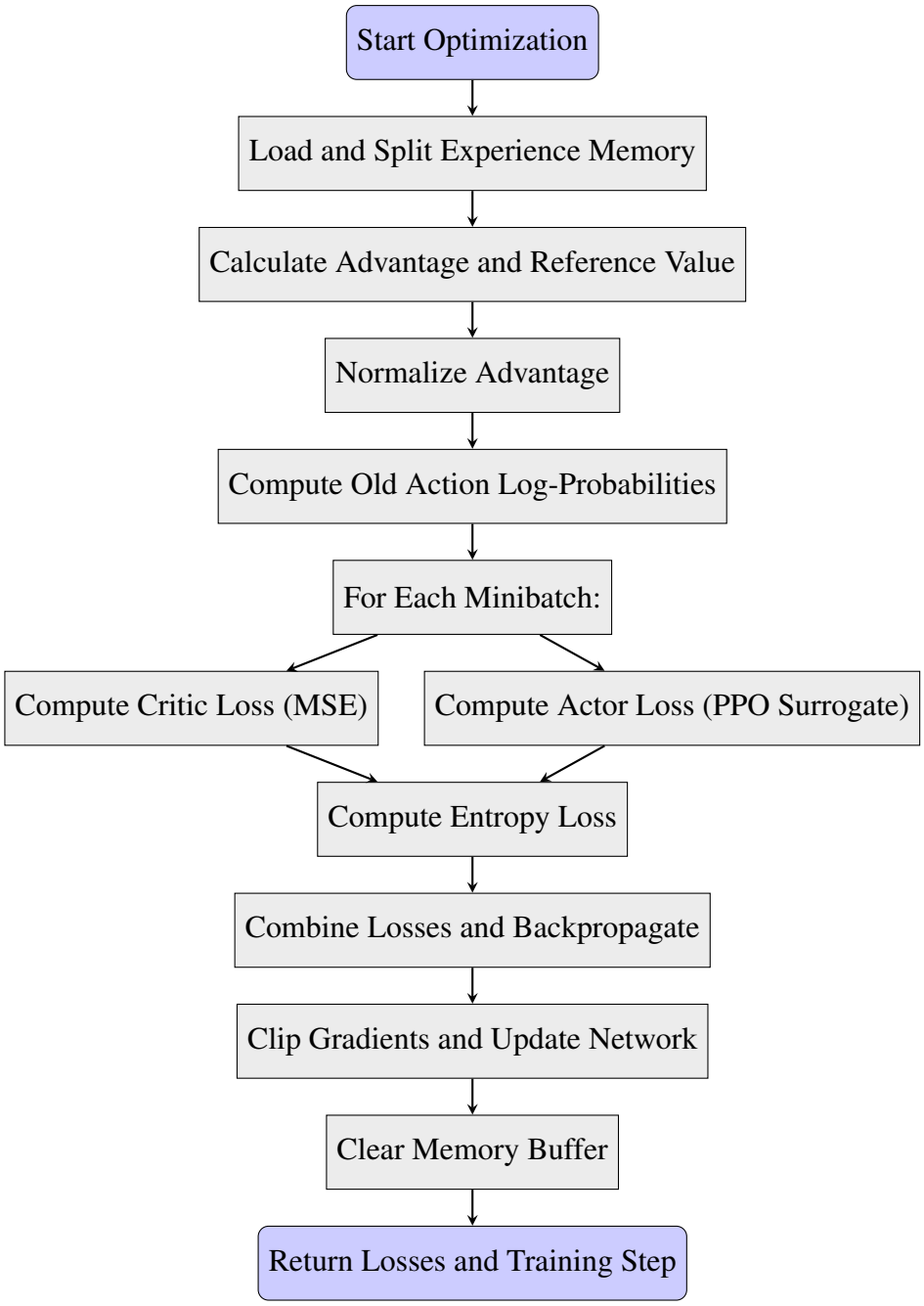
$$\|\nabla\theta\| \leq 0.5$$

The optimizer then performs a step to update the network parameters.

5. **Loss Tracking and Cleanup**: The average actor, critic, and entropy losses are recorded for logging. After all updates, the memory is cleared to prepare for the next episode.

This PPO optimization step improves both the policy (actor) and the value estimation (critic), balancing exploration and exploitation while ensuring stable learning through clipped objectives and normalized advantages.

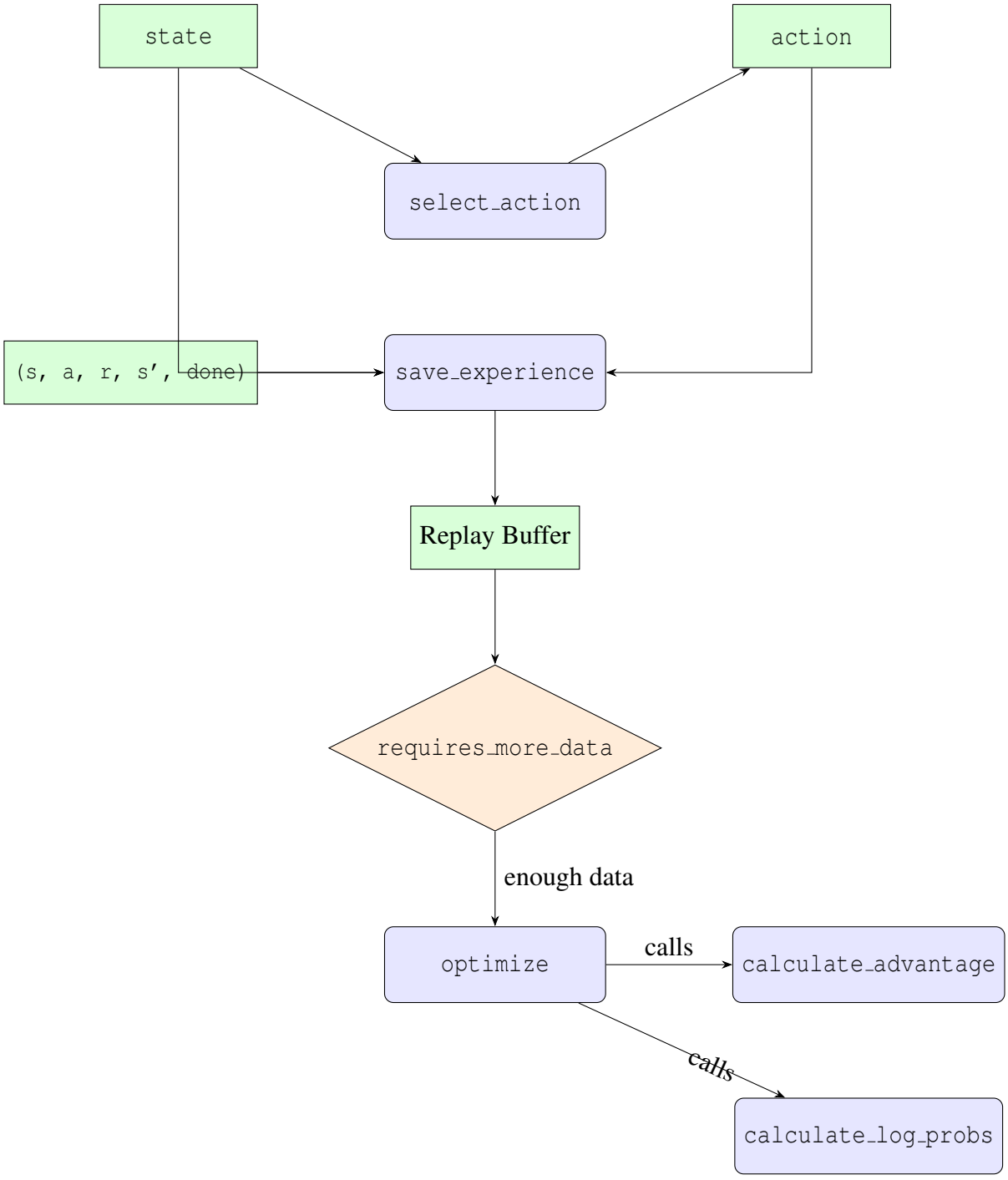
The following diagram visualizes how the `optimize` function works.


Start Optimization
Load and Split Experience Memory
Calculate Advantage and Reference Value
Normalize Advantage
Compute Old Action Log-Probabilities
For Each Minibatch:
Compute Critic Loss (MSE)
Compute Actor Loss (PPO Surrogate)
Compute Entropy Loss
Combine Losses and Backpropagate
Clip Gradients and Update Network
Clear Memory Buffer
Return Losses and Training Step

Reproducibility

All random sampling operations use NumPy or PyTorch built-ins with fixed seeds (not shown here but recommended for full reproducibility). Hyperparameter values and training steps are configurable through the external `config_args`, and the network architecture is modular, allowing for integration with other learning frameworks.

The following figure shows the data flow in the `AgentPPO` class for easier understanding and visualization:



3.4 Escaper Agent

The escaper was implemented with reference to Janosov et al.[13], modeling evasive behavior under multi-agent pursuit scenarios to simulate how faster prey dynamically

reacts to coordinated group tactics.

3.5 Environments Development

We provide **four distinct environments** for training and evaluating pursuit-evasion strategies, each with unique obstacle configurations and spatial layouts.

One of the most critical parameters for environment selection is `env_idx`, found in the `env_config.json` file. This index determines which environment layout is loaded during training or evaluation. Custom environments can be created or modified using the `env_utils.py` script which has predefined border shapes.

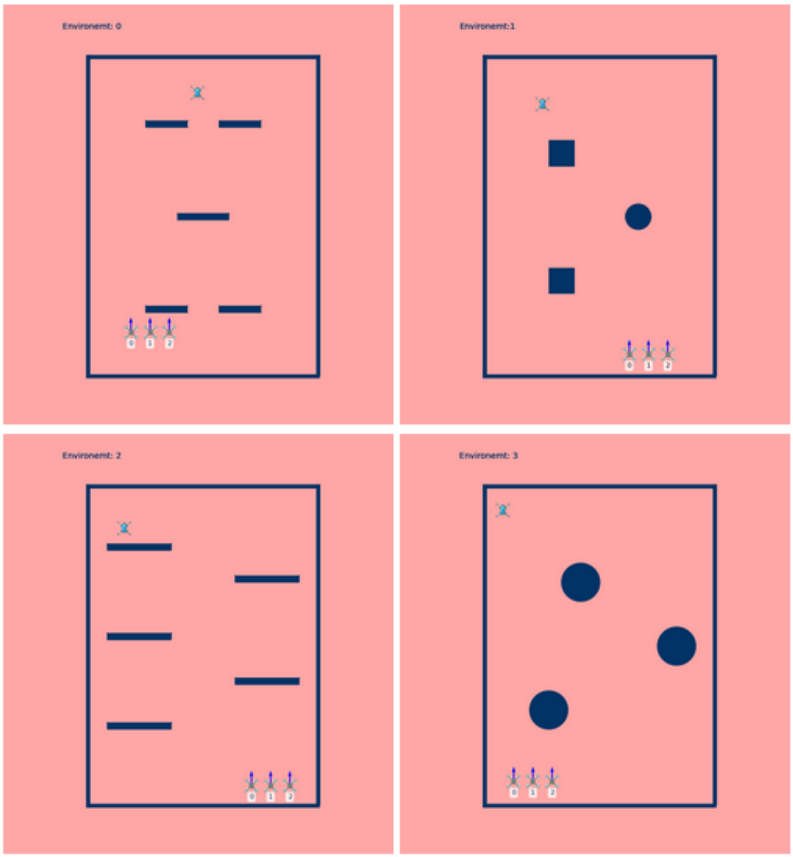


Figure 3.1: Four distinct predefined environments (0, 1, 2, and 3) used in training and testing.

Environment Configuration Parameters

The behavior and structure of the environment are controlled by a number of hyperparameters, which can be adjusted in the configuration file. These are listed in Table 3.2.

Table 3.2: Environment configuration parameters

Parameter	Description
`num_p`	Number of pursuer drones
`r_velocity`	Velocity of pursuer drones in mm/s
`num_e`	Number of evader drones
`e_velocity`	Velocity of evader drones in mm/s
`env_idx`	Index specifying which environment layout to use
`obstacle_num`	Number of obstacles placed in the environment
`neg_reward`	Penalty for collisions or failure to catch evaders
`mode`	Mode of operation, e.g., `"Train"`
`more_r`	Extra reward for maintaining distance among teammates
`old_env`	Full observability of teammates and targets if true
`num_action`	Number of discrete actions per agent
`delta_t`	Time step interval in seconds
`random_env`	Randomizes the environment layout each episode
`r_perception`	Perception range of pursuers in mm

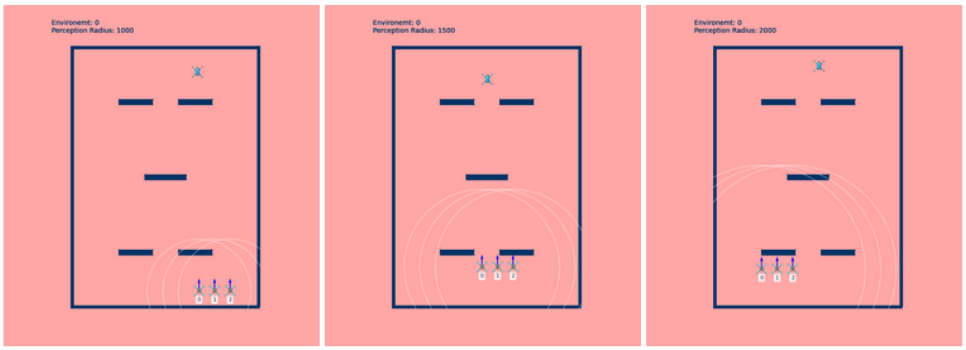


Figure 3.2: Environment 0 initialized with different values for `r_perception`

Implementation of the Environment Class

The core environment logic is implemented within the `Environment` class, which manages agent initialization, environment generation, state updates, and reward calculations for multi-agent pursuit scenarios. It is designed for flexibility, reproducibility, and integration with PyTorch-based reinforcement learning frameworks.

On initialization, the environment loads parameters from a configuration file and sets up agents, obstacles, and boundaries based on a predefined or randomized layout index. Obstacle configurations combine fixed shapes with randomly placed obstacles, supporting a variety of spatial constraints. Environment's `reset` function restores the simulation to its initial state, reinitializing agent and evader positions, and reconstructing the obstacle field.

Agents receive a positive reward for capturing evaders and are penalized for collisions, being too close to teammates or obstacles, and failing to reduce their distance to evaders. The episode terminates when the evader is captured, the maximum time step is reached, or an agent violates proximity constraints. Additional shaping rewards are applied when more r is enabled, encouraging agents to spread out and improve coverage during pursuit.

The environment logic is driven by a structured loop that processes agent actions, updates the simulation state, and computes outcomes at each timestep. When agents select actions, these are converted into 2D force vectors that steer their movement, while evaders follow custom escape behaviors like wall-following and zigzagging to avoid capture. The environment then updates all positions, recalculates rewards based on proximity, collisions, and captures, and checks termination conditions such as evader capture or timeouts. State representations are maintained in two forms: a local state, encoding each agent's perception of nearby obstacles, evaders, and teammates within its sensory range, and a global state, which holds complete environment information. Visibility is determined through geometric calculations involving distance and viewing angles. The captured evader can be dynamically converted into virtual obstacle by generating a circular cluster of points around its position, which augments the existing obstacle set and affects subsequent agent navigation. Throughout each episode, the environment tracks obstacle proximity for each agent to inform movement decisions, records the last known positions and orientations for logging, and provides a flexible rendering system for real-time visualization or image logging, depicting agents, obstacles, and perception ranges clearly. This modular, perception-aware structure ensures realistic multi-agent interactions while remaining adaptable for a range of experimental setups.

Chapter 4

Training, Simulation and Evaluation

This chapter focuses on the training of the PPO agents, the evaluation logic, and both simulation and real-life run of evaluation episodes.

4.1 Training Script Overview

The PPO agent is trained using a custom training script, which sets up the multi-agent environment, initializes the PPO agent, and repeatedly collects experience data through environment interactions. The script handles episodic training, optimizes the actor-critic network, and periodically evaluates and saves model checkpoints.

4.1.1 Initialization and Environment Setup

The training script begins by parsing configuration parameters through a `ConfigLoader` utility. These configurations include environment settings, training hyperparameters, model save paths, and device selection. A set of parallel environments is created using the `ParallelEnv` class, each consisting of multiple agents interacting simultaneously.

The PPO agent is initialized either from scratch or using a pretrained model for fine-tuning purposes. Each agent is paired with an actor-critic neural network (implemented in `ActorCriticPPO`) to generate actions based on observations and update its policy.

4.1.2 Experience Collection

During each episode, agents interact with their respective environments to collect transitions:

$$(\text{state}, \text{action}, \text{reward}, \text{next\_state}, \text{done})$$

Agents select actions using their current policies, and each interaction is stored in a buffer. Once enough data is gathered (checked via `requires_more_data()`), policy updates are triggered.

4.1.3 Policy Optimization

PPO updates are performed in batches using the collected experience. The optimization minimizes a clipped surrogate loss and includes three components:

- **Actor loss**: Encourages better action selection
- **Critic loss**: Improves value estimation
- **Entropy bonus**: Maintains exploration

A linearly decaying learning rate schedule is applied for stability. Key metrics such as average reward and episode length are logged throughout.

4.1.4 Evaluation and Checkpoints

At regular intervals (or when performance improves), the current policy is evaluated using a fixed number of validation episodes. The evaluation results include success rate, collision rate, and average episode duration. These metrics are printed and optionally logged.

If a new best-performing model is found based on the success rate, it is saved along with a summary of configuration parameters and training metrics.

4.1.5 Training Completion

The training process continues until the total number of training steps reaches the predefined maximum limit. At this point, all environments are closed, and training is concluded.

4.1.6 Key Implementation Components

The training loop involves the following core modules:

- `AgentPPO` – The core PPO agent implementation
- `ActorCriticPPO` – The actor-critic neural network used for policy and value estimation
- `ParallelEnv` – Handles parallelized multi-agent environment instances
- `validate_s_env` – Performs evaluation of agent performance
- `ConfigLoader` – Loads training and environment configuration files

4.2 Training Modes and Experiments

Since PPO is theoretically built for single agent scenarios, the motivation was to investigate how smart tweaks in the training can improve the results while keeping the architecture lightweight. Alongside the standard PPO setting, two additional PPO training variants were implemented and compared. Each mode provides a unique perspective on how agents can learn in a multi-agent pursuit-evasion setting.

Standard PPO Training

In the standard setting, agents operate under **strict partial observability**, limited solely to what they can perceive within their local sensing radius. The sensitivity radius is defined, and it can be easily changed in the hyperparameters list. This setup simulates a realistic, decentralized environment where each agent must learn independently without access to global information.

While this configuration reflects the original algorithm scenario, it presents a significant learning challenge. Agents often struggle with coordination, and the resulting behaviors may be overly conservative or selfish. However, this mode is essential for evaluating policy robustness under constrained perception and for testing generalization.

PPO with Full Observability

To explore the benefits of centralized training strategies like MAPPO, which employs a global critic with access to the full state of all agents, I introduced a minimal variant where agents are provided with the positions of their teammates and the evader during training. Although this offers significantly less information than a full global critic omitting velocities, intentions, or action histories, it represents a lightweight and practical approximation.

The goal of this variant is to accelerate early-stage learning by easing coordination demands, and to investigate whether sharing minimal global information can enhance collaborative behaviours. This setup mimics centralised training with decentralised execution principles, enabling faster convergence and stronger group strategies without the overhead of a full global critic.

Fine-Tuned PPO Training

This mode adopts a **two-stage learning process** that leverages the strengths of both previous approaches. Agents are first trained under full observability to develop a baseline pursuit strategy and cooperative tendencies. The resulting policy is then fine-tuned in the standard, partially observable environment using a reshaped reward function to emphasize detection and proximity.

This hybrid approach addresses the stagnation and passive behaviors observed in standard training. By bootstrapping with a well-informed policy and refining it under realistic conditions, agents achieve a better balance between individual exploration and team coordination. Fine-tuning improves pursuit effectiveness while maintaining robustness to sensory limitations.

4.2.1 Best Training Parameters per PPO Configuration

The best training parameters and corresponding evaluation results for each PPO configuration are summarized in Table 4.1. The training process was monitored using key performance metrics including success rate, collision rate, and the average number of steps per episode. The resulting performance graphs for each configuration are presented in Figures 4.1, 4.2, and 4.3.

Table 4.1: Best Hyperparameters for Each PPO Configuration

Parameter	**Classic PPO**	**PPO Finetune**	**PPO Full Obs**
Success Rate (%)	74	70	88
Collision Rate (%)	26	26	12
Avg Steps	398.81	208.79166666	222.2127
Training Steps	268800	6050	86400
Episode	2240	50	720
Learning Rate	0.0003	0.0003	0.0003
Gamma	0.99	0.95	0.95
GAE Lambda	0.95	0.95	0.95
Epsilon Clip	0.2	0.1	0.15
PPO Epochs	10	10	10
Batch Size	1024	1024	1024
Mini-batch Size	256	256	256
Environments Number	6	20	10
Pursuers' Velocity	300	200	200
Evader's Velocity	600	300	300
Perception Radius	2000	2000	2000
Obstacle Number	5	5	5
Random Environment	False	False	False
Old Environment	False	False	True
Delta T	0.1	0.1	0.1
Pre-trained Model	No	Yes	No

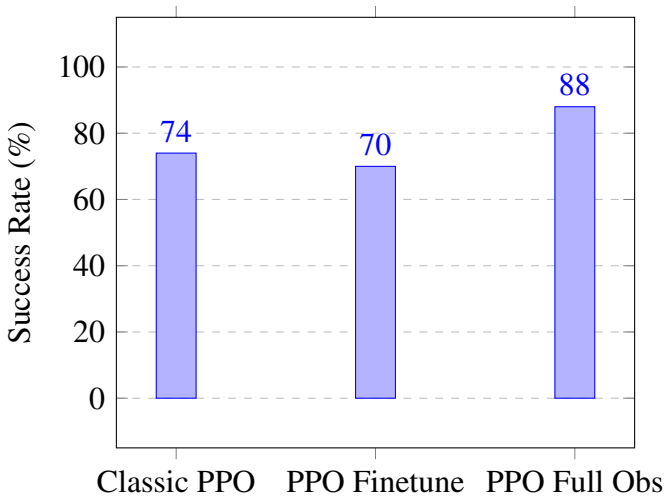


Figure 4.1: Training Success Rate comparison across PPO configurations

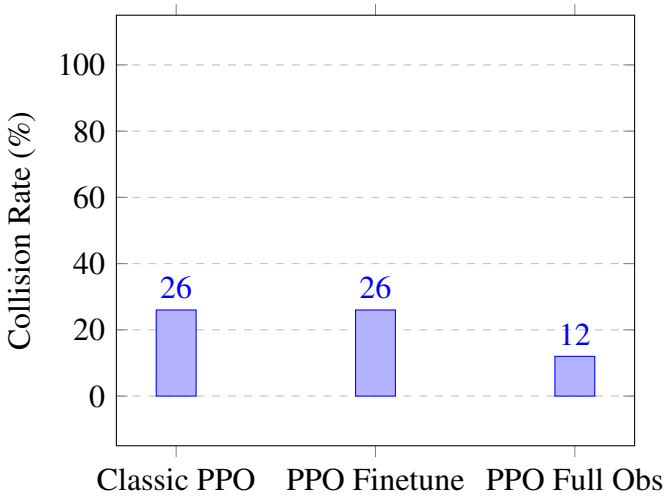


Figure 4.2: Training Collision Rate comparison across PPO configurations

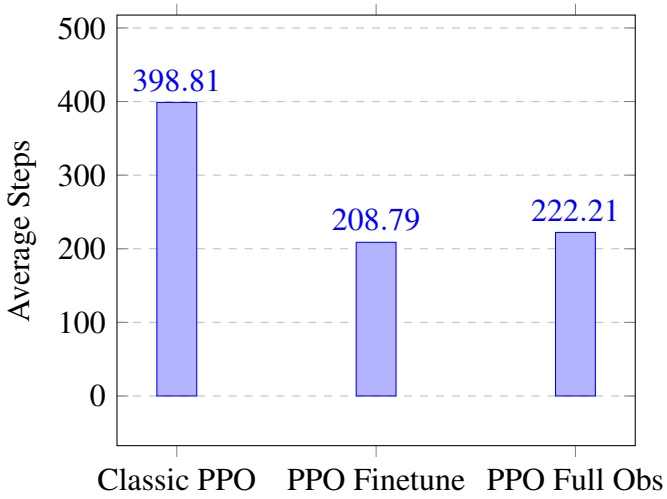


Figure 4.3: Training Average Steps to Capture across PPO configurations

4.3 Simulation

To evaluate and demonstrate the behavior of trained agents, we developed a set of rendering tools that simulate and visualize validation episodes. These simulations allow us to observe the interactions between the pursuing drones and the evader drone, typically culminating in a successful capture or, in rare cases, a collision.The main tool, `run_sim.py`, loads a model and environment, and renders agent interactions live or as a GIF. It is configurable through a range of command-line arguments shown in Table 4.2.

Table 4.2: Command-line arguments for `run_sim.py`

Flag	Short	Type	Default	Description
`--model_config`	`-mc`	string	`./config/training_config.json`	Path to the model con
`--env_config`	`-ec`	string	`./config/env_config.json`	Path to the environme
`--model`	`-m`	string	`./results/best_models/PPO/best.pt`	Path to the trained mo
`--save_path`	`-s`	string	`./render_results/`	Directory where ren s/images) will be save
`--show_config`	`-sc`	flag	`False`	If set, prints the curren console.
`--env_idx`	`-i`	integer	`None`	Index of the environm
`--delta_t`	—	float	`None`	Time interval (in sec lation steps.
`--r_velocity`	`-rv`	integer	`None`	Velocity of the pursue
`--e_velocity`	`-ev`	integer	`None`	Velocity of the evade
`--r_perception`	—	integer	`None`	Perception range of th
`--obstacle_num`	—	integer	`None`	Number of obstacles

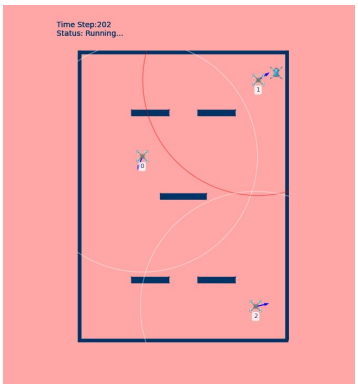
4.3.1 Example Usage

This command sets up a specific scenario, renders the episode, and saves the output as a GIF.

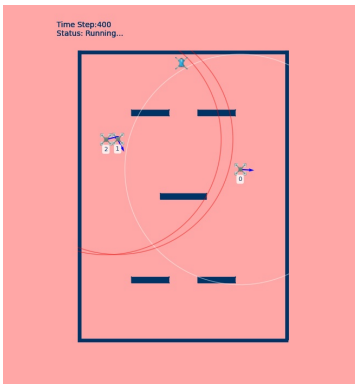
```
python run_sim.py --model_config ./config/training_config.json \
                  --env_config ./config/env_config.json \
                  --model ./results/best_models/PPO/best.pt \
                  --save_path ./render_results/ \
                  --env_idx 2 \
                  --delta_t 0.1 \
                  --r_velocity 300 \
                  --e_velocity 600 \
                  --r_perception 1500 \
                  --obstacle_num 5 \
                  --show_config
```

4.3.2 Capturing Final States

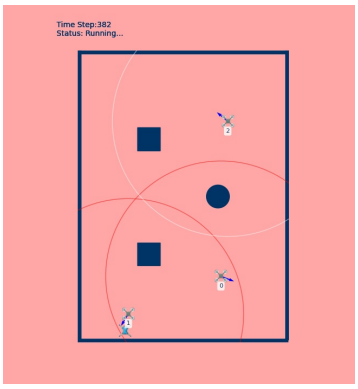
For visual summaries of episode outcomes, `captures_caught_state.py` is provided. This script renders the full episode and saves an additional static image of the final state, either a successful capture, or a collision. Examples are shown in Figure 4.4.



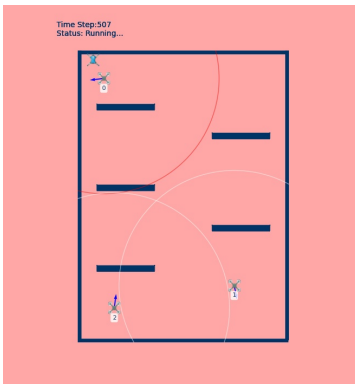
(a) Captured evader Environment 0



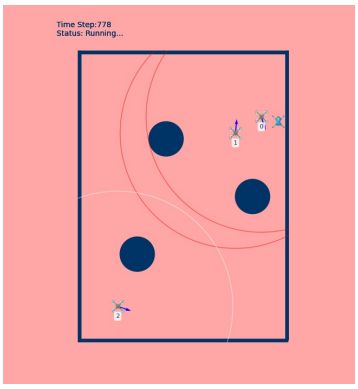
(b) Collision Environment 0



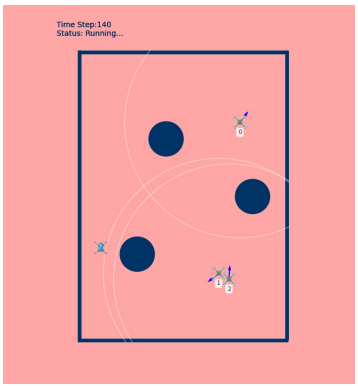
(c) Captured evader Environment 1



(d) Captured evader Environment 2



(e) Captured evader Environment 3



(f) Collision Environment 3

Figure 4.4: Final simulation snapshots across environments, showcasing successful captures and collision events during evaluation.

4.4 Evaluation

4.4.1 Experimental Setup

To rigorously assess the performance of the proposed multi-agent learning framework, we evaluated three PPO configurations on four environments with increasing complexity. All models were exclusively trained on **Environment 0**, and then tested in all four environments to assess generalization. During training, the model checkpoint with the best validation performance was saved and later used for evaluation. Despite its architectural simplicity compared to MAPPO, PPO showed promising results, particularly when trained with thoughtful modifications. This section evaluates three PPO variants, highlighting how small yet strategic shifts in the training pipeline can lead to strong multi-agent coordination and competitive generalization.

Each policy was tested for 150 episodes per environment. The evaluation criteria focused on the following metrics:

- **Success Rate (%)**: Percentage of episodes where all pursuers successfully captured the evading target.
- **Collision Rate (%)**: Episodes in which drones collided with each other or obstacles.
- **Average Steps to Capture**: Average number of steps taken per episode before reaching an outcome.

The evaluation results across all four environments are shown in Table 4.3.

4.4.2 Standard PPO Performance Analysis (Observability within Range)

The standard PPO model demonstrated modest performance in the environment it was originally trained on (Environment 0), achieving a success rate of 50.7%, which, while better than random, remained unimpressive compared to other models. Its limitations became even more apparent in unseen environments, with success rates sharply declining to 27.3% in Environment 1, 24.0% in Environment 2, and reaching a concerning low of 18.0% in Environment 3. This suggests that PPO struggled significantly with generalization beyond its training scenario.

In terms of safety, reflected by the collision percentage, PPO consistently showed the highest collision rates across all environments, peaking at 75.3% in Env 2 and remaining above 44% in Env 0. This indicates a lack of reliable collision-avoidance behavior and poor spatial coordination among agents, both in familiar and novel settings.

Additionally, the average number of steps required to complete an episode with PPO was consistently the highest or among the highest across the environments. It required 414.42 steps in Environment 0 and as many as 575.90 in Environment 1 and 551.56 in Environment 3, reflecting inefficiency in pursuit and capture tasks. The combination of low success rates, high collision frequency, and lengthy episode durations highlights PPO's inability to develop robust, transferable pursuit strategies within this multi-agent environment framework.

Table 4.3: Evaluation Results across all 4 environments

Metric	MAPPO	PPO_finetune	PPO	PPO_old_env
Env 0 Success %	88.7%	63.3%	50.7%	54.7%
Env 0 Collision %	11.3%	36.0%	44.7%	44.7%
Env 0 Avg Steps	256.17	360.05	414.42	279.02
Env 1 Success %	80.0%	77.3%	27.3%	68.0%
Env 1 Collision %	18.7%	21.3%	52.7%	31.3%
Env 1 Avg Steps	284.68	339.25	575.90	275.75
Env 2 Success %	69.3%	56.7%	24.0%	48.0%
Env 2 Collision %	30.0%	43.3%	75.3%	52.0%
Env 2 Avg Steps	267.63	335.68	463.64	190.76
Env 3 Success %	67.3%	68.7%	18.0%	64.7%
Env 3 Collision %	31.3%	29.3%	60.0%	35.3%
Env 3 Avg Steps	288.73	321.81	551.56	224.53

4.4.3 Full Observability PPO Performance Analysis

The full observability PPO model showed notably better performance than standard PPO, with success rates ranging from 48.0% to 68.0% across environments. While not surpassing MAPPO, it consistently closed the gap, particularly in Environment 3, where it reached 64.7%, just 3% below MAPPO. Its performance distribution followed a similar pattern to MAPPO, excelling in the same environments and struggling in

others, suggesting that its training setup allowed it to capture some of the environment-specific coordination benefits.

In terms of collisions and efficiency, PPO_full_obs also improved over standard PPO, with lower collision rates (31.3–52.0%) and notably shorter episode lengths. It achieved the lowest average number of steps per episode in Environment 2 and Environment 3 among all PPO variants, approaching or even outperforming MAPPO in these environments. While not the strongest model overall, it demonstrated that even without a centralized critic, thoughtful training adjustments and full observability during training can significantly enhance multi-agent performance.

4.4.4 Fine-tuned PPO Performance Analysis

The PPO_finetune model was the strongest performer among the three PPO variants, and notably, it even outperformed MAPPO in Environment 3, achieving a success rate of 68.7%, exceeding MAPPO's 67.3%. Across all environments, it consistently outclassed the standard PPO, with a particularly impressive result in Environment 1, where it achieved the highest single success rate of 77.3%—a 50% absolute improvement over standard PPO's 27.3% in the same setting. Despite lacking a centralized critic, the fine-tuning strategy, transitioning from full to partial observability, proved highly effective in capturing coordination benefits typically associated with CTDE frameworks, without the added training complexity.

In terms of collisions, PPO_finetune maintained moderate rates between 21.3% and 43.3%, significantly lower than standard PPO's problematic 44.7%–75.3% range. It also delivered better episode efficiency, with consistently fewer steps required to complete episodes than baseline PPO, particularly in Environment 0 and Environment 1, where it reduced average steps by over 50–200 steps. These results confirm that a simple fine-tuning approach can substantially elevate decentralized PPO performance, delivering both higher success rates and more reliable, efficient behavior across varied environments. Figures 4.5, 4.6, 4.7 summarize the evaluation results.

4.4.5 Summary of the Evaluation Results

The results we obtained support our initial hypothesis, inspired by prior research, that PPO can still remain competitive in multi-agent environments with appropriate adjustments. By keeping the architecture lightweight and creatively introducing partial forms of shared information, such as incorporating agent location into observations

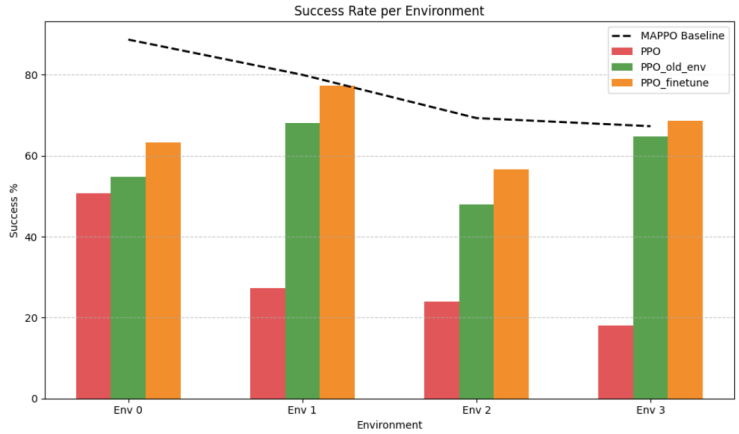


Figure 4.5: Success rate across the 3 PPO variants with MAPPO as a baseline

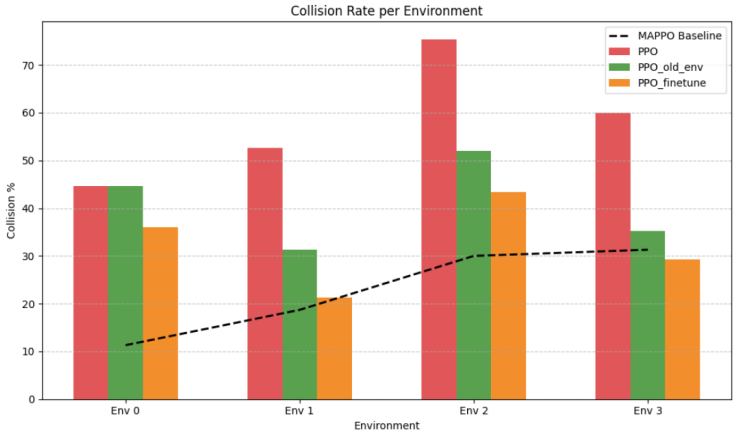


Figure 4.6: Collision rae across the 3 PPO variants with MAPPO as a baseline

without resorting to a centralized critic, we observed notable performance improvements. Notably, the *PPO full observability* variant consistently required the fewest steps on average to complete episodes, indicating greater efficiency in navigation and decision-making.

Interestingly, the fine-tuned PPO variant exhibited lower training accuracy (70%) compared to the standard PPO (74%), yet it significantly outperformed all the variants in evaluation, even outperforming MAPPO in one environment. This suggests that early exposure to richer observability during training helped the agents develop more generalizable coordination strategies, even when later restricted to partial observability. It also indicates that higher training accuracy does not always correlate with better generalization, especially in dynamic multi-agent settings where coordination is crucial.

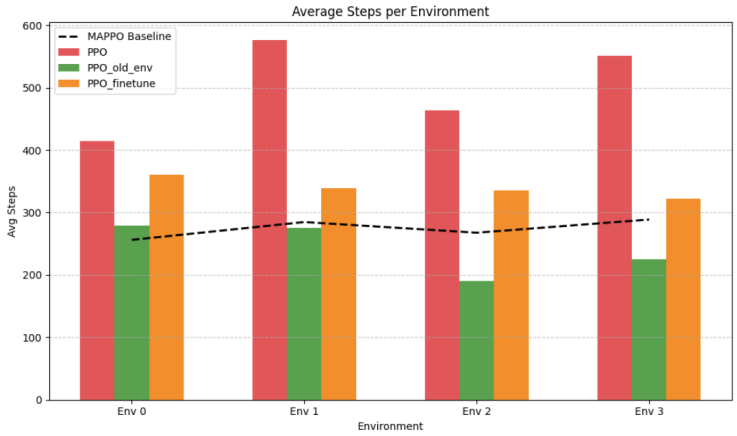


Figure 4.7: Average Number of Steps across the 3 PPO variants with MAPPO as a baseline

- The results reinforce that small, targeted modifications in observability or training schedules can substitute for more complex centralized structures in certain cooperative tasks.
- The reduction in average episode steps across variants hints at improved policy efficiency, which is especially valuable for real-world applications with time or resource constraints.
- The discrepancy between training and evaluation performance highlights the importance of robust evaluation protocols in multi-agent RL, where overfitting to partial observability during training can limit emergent coordination.

4.5 Real World Deployment

To test the trained model outside of simulation, we deployed it on real "Crazyflie" 2.1 drones equipped with FlowDecks for optical flow sensing shown in Figure 4.8. The "Crazyflie" drones weigh just 27g and are small enough to fit in the palm of a hand, making them ideal for testing scenarios where safety is a concern, as collisions pose no risk to people or the environment.

The system used the drones' onboard sensor to estimate their positions, then applied a coordinate transformation to align those positions with the layout of the virtual environment. Based on this alignment, the model decided the next move, which was sent directly to the drone for execution. Since "Crazyflie" drones do not have perception radius for obstacles, they were added in the virtual environment. Four "Crazyflie"

drones and one “CrazyRadio” to connect with the drones were used. It is crucial to make sure the radio addresses to control the drones are correctly set up.

The real-world testing was successful, with the drones displaying similar behavior to what we observed during simulation. Coordination, movement patterns, and decision-making trends carried over well, showing that the model was able to generalize from virtual to physical environments. This marks the completion of the full cycle, from building the neural network to running it in a real-world multi-agent scenario. Figure 4.9 shows a snapshot of a real-world deployment episode.

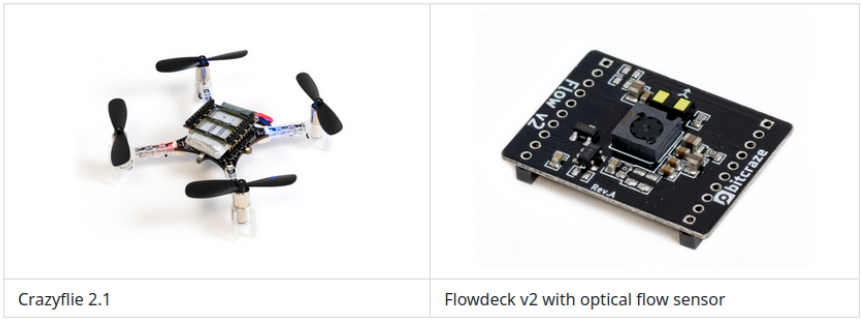


Figure 4.8: Crazyflie 2.1 drone and optical disk

Figure 4.9: Snapshot of a real-world deployment video

Chapter 5

Summary and Conclusions

5.1 Summary

This thesis encompassed the full development cycle of a decentralized multi-agent pursuit system, from neural network design to deployment on physical drones. The core motivation was to investigate whether an algorithm originally intended for single-agent applications, Proximal Policy Optimization (PPO), could be adapted for multi-agent scenarios, specifically within pursuit-evasion tasks where 3 pursuers need to catch an evader. Compared to its multi-agent counterpart, MAPPO, PPO offers a lightweight, computationally efficient alternative, making it attractive for resource-constrained, real-world applications like drone swarms.

A custom 12-layer actor-critic neural network was implemented, incorporating a learnable gamma parameter to dynamically adjust temporal weighting during training. This architecture was designed to capture inter-agent spatial dependencies while maintaining decentralized control, where each agent learns independently, penalized for collisions and rewarded for coordinated captures within a defined radius.

To probe the limits of PPO's decentralized framework, two additional variants were introduced. The first extended observability to include the positions of all agents, approximating the benefits of a centralized knowledge without explicit coordination. The second, a fine-tuned PPO model, began training with full observability before reverting to local observability mid-training. This design was inspired by concepts from centralized training with decentralized execution (CTDE), hypothesizing that early exposure to global state information could prime agents for more effective decentralized coordination.

Four custom environments of increasing complexity were developed, alongside

tools for generating new layouts and rendering simulations for qualitative analysis. Training was conducted in a single environment, while inference and evaluation spanned all four, ensuring the models' ability to generalize to unseen scenarios.

Among the trained models, the PPO fine-tune variant achieved the highest success rates, peaking at 77.3% in one environment, surpassing even MAPPO in other one. This result highlights the potential of carefully designed training schemes in bridging the performance gap between decentralized and centralized multi-agent systems. The full-observability PPO also demonstrated notable improvements over the standard PPO, though its performance declined in cluttered environments, indicating possible overfitting or limitations in generalizing from globally observable to partially observable conditions. The standard PPO baseline performed weakest, underscoring the challenges posed by partial observability in multi-agent settings.

Beyond success rates, additional metrics such as collision rates and average episode lengths were considered. Notably, the PPO with full observability achieved the shortest episodes on average, even outperforming MAPPO in this aspect, suggesting improved coordination efficiency despite lacking a centralized critic.

Simulation tools were developed for recording episodes, detecting collisions, and capturing final outcomes. Ultimately, the best-performing models were successfully deployed on real Crazyflie 2.1 drones equipped with onboard optical flow sensors. Real-world trials validated the policies trained in simulation, completing a full pipeline from algorithm design to physical deployment.

Overall, this project showed how even small changes in how agents perceive their environment during training can have a big impact on how well they learn to work together. It reinforced the idea that balancing between giving agents enough information to learn good strategies while keeping the system decentralized is tricky, but achievable with the right setup. The work also highlighted how important it is to have flexible and customizable tools when experimenting with multi-agent systems, especially when aiming to eventually move from simulation to real-world tests. These insights could be useful for anyone working on similar problems, whether in research or practical applications like drone swarms or search-and-rescue scenarios.

5.2 Achievements and Limitations

This project successfully delivered a complete system for multi-agent pursuit learning and deployment, meeting and, in some aspects, exceeding the initial objectives. A

modular and extensible simulation framework was developed from scratch, supporting flexible environment generation with configurable agent numbers, perception ranges, evader speeds, and layout designs. This allowed for systematic experimentation with complex pursuit dynamics where the evader maintained a speed advantage, necessitating emergent coordination among agents rather than reliance on individual speed.

A fully custom PPO training architecture was implemented, featuring a 12-layer actor-critic neural network with a learnable temporal discount factor. This design enabled dynamic adjustment of temporal dependencies, allowing agents to adapt their decision-making based on spatial configurations and proximity to teammates and the evader. The training process incorporated Generalized Advantage Estimation, clipped policy updates for stability, and entropy regularization to balance exploration and exploitation. The codebase was intentionally designed for clarity and adaptability, enabling straightforward modifications to network architectures, training configurations, or environment setups, making it well-positioned for future extensions and comparative research.

Three distinct PPO variants were proposed and evaluated, with the fine-tuned model transitioning from full to partial observability demonstrating superior generalization, achieving a 77% success rate in previously unseen environments. This variant outperformed standard PPO and, in specific cases, surpassed MAPPO's performance, underscoring the potential of thoughtful training adjustments to mitigate the limitations of decentralized learning. Furthermore, the successful sim-to-real transfer was demonstrated by deploying trained policies on Crazyflie 2.1 drones using onboard optical flow sensors, validating the system's end-to-end viability from simulation to physical deployment.

Limitations and Areas for Improvement

Despite these achievements, several limitations were identified that constrain the broader applicability and realism of the approach. The most notable is the omission of drone-specific kinematics and actuation constraints. The simulation assumes holonomic motion in discrete steps, oversimplifying the physical constraints of real aerial robots, such as momentum, acceleration bounds, and minimum turning radii. Incorporating low-level control models would improve policy realism and facilitate more reliable sim-to-real transfers.

Another significant limitation is the 2D environment constraint. All agents operated on a fixed plane, omitting vertical movement and altitude management, which

are essential components of real-world aerial pursuit and evasion. Extending the environment to a full 3D domain would introduce new strategic complexities and make coordination behaviors more representative of actual drone swarm applications.

Additionally, the simulation assumed perfect sensing and instantaneous, error-free decision-making within each agent's perception radius. In realistic multi-agent systems, sensing is subject to noise, occlusion, and latency. Integrating probabilistic sensing models, intermittent data availability, and communication constraints would increase policy robustness and prepare agents for more unpredictable, real-world deployments.

A further limitation lies in the simplicity of the evader's policy, which followed a reactive, greedy avoidance strategy without long-term planning. While this provided a useful testbed for agent coordination, it did not fully challenge the adaptability of pursuit strategies. Introducing more advanced evaders with predictive behaviors or adversarial reinforcement learning could expose weaknesses in current policies and drive the development of more sophisticated pursuit tactics.

5.3 Future Work

The current framework provides a solid foundation for further research in multi-agent pursuit-evasion tasks, and several promising directions can extend its capabilities both in simulation and real-world applicability.

A major strength of the system is its modular design. The simulation environment and benchmarking tools are agent-agnostic, meaning future work can swap out the PPO agent with other learning algorithms or decision-making architectures, such as DQN variants, Transformer-based policies, or even traditional rule-based heuristics. As long as a suitable neural network and agent logic are provided, the environment can serve as a reusable benchmark for comparative studies. This opens the door for a wide range of experiments in policy evaluation, transfer learning, and decentralized control.

Building on the current 2D environment, a natural and ambitious extension would be to develop a full 3D simulation framework. This would enable modeling vertical movement, complex obstacle geometries, and realistic drone kinematics. A 3D system could also support learning richer navigation strategies, such as altitude-based evasion or stacking formations among pursuers. Importantly, this would bring the simulator closer to real drone dynamics, enhancing the fidelity of sim-to-real transfer.

To improve real-world deployment accuracy, future work could also explore using

a calibrated 3D motion capture system, such as OptiTrack or Vicon, for precise localization. This would eliminate the need to align optical flow–based positioning with the virtual environment manually, simplifying the deployment pipeline and reducing noise. Additionally, expanding the testing grounds to outdoor environments would significantly raise the complexity and realism of the task introducing challenges like GPS drift, lighting variability, wind, and physical obstructions. These conditions could be paired with online adaptation or reinforcement learning methods to create policies robust to dynamic, real-world changes.

From a control perspective, one critical direction is to integrate low-level control learning or model predictive control (MPC) within the decision-making loop. This would allow the agent to consider acceleration, inertia, and non-holonomic constraints during action selection, making learned policies more directly executable by real UAVs without needing intermediary smoothing or translation layers.

Finally, incorporating realistic noise models, including perception uncertainty, communication dropouts, and partial observability with occlusions, would allow training agents that are resilient to the imperfections of real-world sensing. This would better prepare agents for deployment in unpredictable environments and move the system closer to applications in search-and-rescue, surveillance, or autonomous patrol.

5.4 Final Remarks

This project has been an invaluable learning opportunity, providing hands-on experience in developing, training, and deploying reinforcement learning algorithms for multi-agent systems. It offered valuable insights into both the technical and practical challenges of sim-to-real transfer and multi-agent coordination.

Importantly, the results validated the initial hypothesis that with smart and well-designed training adjustments, it is possible to significantly enhance the performance of decentralized agents in complex pursuit-evasion tasks. The fine-tuned PPO variant not only outperformed the standard decentralized baseline but, in one environment, even surpassed the performance of MAPPO, which served as a centralized, state-of-the-art baseline throughout this study. This result highlights how targeted modifications, like staged observability or environment-specific tuning, can bridge the performance gap between decentralized and centralized approaches.

Overall, this work demonstrated the potential of lightweight, decentralized reinforcement learning methods for real-world, resource-constrained multi-agent scenarios. I hope these findings, alongside the developed simulation tools and deployment framework, can support and inspire further research in this exciting field.

Bibliography

[1] Yunes Alqudsi and Murat Makaraci. Uav swarms: research, challenges, and future directions. *Journal of Engineering and Applied Science*, 72(1):12, 2025.

[2] Christopher Amato. An introduction to centralized training for decentralized execution in cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2409.03052*, 2024.

[3] Matthew Ayamga, Selorm Akaba, and Albert Apotele Nyaaba. Multifaceted applicability of drones: A review. *Technological Forecasting and Social Change*, 167:120677, 2021.

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

[5] Berk Cetinsaya, Dirk Reiners, and Carolina Cruz-Neira. From pid to swarms: A decade of advancements in drone control and path planning - a systematic review (2013–2023). *Swarm and Evolutionary Computation*, 89:101626, 2024.

[6] Wu Chen, Jiajia Liu, Hongzhi Guo, and Nei Kato. Toward robust and intelligent drone swarm: Challenges and future directions. *IEEE Network*, 34(4):278–283, 2020.

[7] Khalil Chikhaoui, Hakim Ghazzai, and Yehia Massoud. Ppo-based reinforcement learning for uav navigation in urban environments. In *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2022.

[8] Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong. Introduction to reinforcement learning. *Deep reinforcement learning: fundamentals, research and applications*, pages 47–123, 2020.

[9] Hugging Face. Hugging face: Deep rl course. https://huggingface.co/learn/deep-rl-course/en/unit1/exp-exp-tradeoff. Accessed: 2025-04-10.

[10] Yue Guan, Sai Zou, Haixia Peng, Wei Ni, Yanglong Sun, and Hongfeng Gao. Cooperative uav trajectory design for disaster area emergency communications: A multiagent ppo method. *IEEE Internet of Things Journal*, 11(5):8848–8859, 2024.

[11] Wei Hu, Lechao Xiao, and Jeffrey Pennington. Provable benefit of orthogonal initialization in optimizing deep linear networks. *arXiv preprint arXiv:2001.05992*, 2020.

[12] Maximilian Hüttenrauch, Adrian Šošić, and Gerhard Neumann. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research*, 20(54):1–31, 2019.

[13] Milán Janosov, Csaba Virágh, Gábor Vásárhelyi, and Tamás Vicsek. Group chasing tactics: how to catch a faster prey. *New Journal of Physics*, 19(5):053003, 2017.

[14] Maryam Kouzeghar, Youngbin Song, Malika Meghjani, and Roland Bouffanais. Multi-target pursuit by a decentralized heterogeneous uav swarm using deep multi-agent reinforcement learning. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3289–3295, 2023.

[15] Siddharth Krishna Kumar. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863*, 2017.

[16] Yang Li, Dengyu Zhang, Junfan Chen, Ying Wen, Qingrui Zhang, Shaoshuai Mou, and Wei Pan. Hola-drone: Hypergraphic open-ended learning for zero-shot multi-drone cooperative pursuit. *arXiv preprint arXiv:2409.08767*, 2024.

[17] Xiangyu Liu and Kaiqing Zhang. Partially observable multi-agent reinforcement learning with information sharing. *arXiv preprint arXiv:2308.08705*, 2023.

[18] Dolores Gracja Piwek. Commercialization of unmanned aerial vehicles. prospects and challenges in the second decade of the 21st century. *Rocznik Bezpieczeństwa Miedzynarodowego*, 18(2):125–154, 2024.

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

[20] Volker H. Schulz. Book reviews. *SIAM Review*, 63(2):419–431, 2021.

[21] A. Sheik Abdullah, Abdul Aziz A.B, and S. Geetha. Decentralidrone: A decentralized, fully autonomous drone delivery system for reliable, efficient transport of goods. *Alexandria Engineering Journal*, 88:1–30, 2024.

[22] Max Simchowitz and Aleksandrs Slivkins. Exploration and incentives in reinforcement learning. *Operations Research*, 72(3):983–998, 2024.

[23] Rashmi Singh, Aarti Singh, and Saurabh Mukherjee. A critical investigation of agent interaction protocols in multiagent systems. *International journal of Advancements in Technology*, 5(2):72–81, 2014.

[24] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[25] Tomasz Szandała. Review and comparison of commonly used activation functions for deep neural networks. In *Bio-inspired neurocomputing*, pages 203–224. Springer, 2020.

[26] Gábor Vásárhelyi, Csaba Virágh, Gergo Somorjai, Norbert Tarcai, Tamás Szörényi, Tamás Nepusz, and Tamás Vicsek. Outdoor flocking and formation flight with autonomous aerial robots. *CoRR*, abs/1402.3588, 2014.

[27] Chathura Wanniarachchi, Prasad Wimalaratne, and Kasun Karunanayaka. Formation control algorithms for drone swarms and the single point of failure crisis: A review. In *2024 IEEE 33rd International Symposium on Industrial Electronics (ISIE)*, pages 1–6, 2024.

[28] Ruiqing Ye, Wudong Deng, Yanning Guo, and Chuanjiang Li. Spacecraft chase and escape game based on ppo algorithm. In *Chinese Conference on Swarm Intelligence and Cooperative Control*, pages 641–651. Springer, 2023.

[29] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in neural information processing systems*, 35:24611–24624, 2022.

[30] Guang Zhan, Xinmiao Zhang, Zhongchao Li, Lin Xu, Deyun Zhou, and Zhen Yang. Multiple-uav reinforcement learning algorithm based on improved ppo in ray framework. *Drones*, 6(7), 2022.

[31] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Decentralized multi-agent reinforcement learning with networked agents: Recent advances. *Frontiers of Information Technology & Electronic Engineering*, 22(6):802–814, 2021.

[32] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control*, pages 321–384, 2021.

[33] Guangchong Zhou, Zhiwei Xu, Zeren Zhang, and Guoliang Fan. Mastering complex coordination through attention-based dynamic graph. In *International Conference on Neural Information Processing*, pages 305–318. Springer, 2023.