Westfälische Hochschule

University of Applied Sciences
Gelsenkirchen Bocholt Recklinghausen

Masterthesis

Titel der Arbeit // Title of Thesis

Hierarchical Reinforcement Learning for autonomous UAV-Based Fire Extinguishing

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree

Master of Science (M.Sc.)

Autorenname, Geburtsort // Name, Place of Birth

Julien Noel Meine, Haltern am See

Studiengang // Course of Study

Technische Informatik

Fachbereich // Department

Informatik und Kommunikation

Erstprüferin/Erstprüfer // First Examiner

Prof. Dr.-Ing. Dipl. Inform. Hartmut Surmann

Zweitprüferin/Zweitprüfer // Second Examiner

Prof. Dr. Wolfram Conen

Abgabedatum // Date of Submission

10.11.2025

Westfälische Hochschule

University of Applied Sciences
Gelsenkirchen Bocholt Recklinghausen

Eidesstattliche Versicherung

Meine, Julien Noel

Name, Vorname // Name, First Name

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel

Hierarchical Reinforcement Learning for autonomous UAV-Based Fire Extinguishing

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Gelsenkirchen, den November 25, 2025

Ort, Datum, Unterschrift // Place, Date, Signature

Abstract

This thesis presents Reinforcement-learning Oriented Simulation for Hierarchical Agent Navigation (ROSHAN), a Hierarchical Reinforcement Learning (HRL) framework for coordinating multiple Unmanned Aerial Vehicle (UAV)s in autonomous wildfire suppression. The system operates within a controllable Cellular Automata (CA) fire simulation and decomposes the task into three layers. A low-level `FlyAgent` responsible for local navigation and fire extinguishing, an `ExploreAgent` that maintains map coverage, and a high-level `PlannerAgent` that allocates suppression goals across agents. The `FlyAgent` is trained using Proximal Policy Optimization (PPO) and achieves stable and robust performance across collision-free and collision-enabled settings after careful tuning of reward shaping, encoder configuration, and update horizon. The `PlannerAgent` is evaluated against a deterministic greedy assignment heuristic and is shown to outperform this baseline under appropriately chosen rollout horizons and planning frequencies, achieving higher success rates and lower median Time-to-Extinguish (TTE). Across experiments, both agent layers exhibit sensitivity to hyperparameter selection, underscoring the importance of aligning temporal abstraction and training schedules. The results demonstrate that HRL can effectively coordinate multiple UAVs for wildfire suppression within the tested environment.

Keywords: Simulation; Cellular Automata; Fire Modeling; Reinforcement Learning; Hierarchical Reinforcement Learning; Collision Avoidance;

Acknowledgement

I would like to express my sincere gratitude to Prof. Dr. Surmann, who not only supervised this thesis, but also accompanied my academic development throughout my Bachelor's and Master's studies. His teaching and guidance in artificial intelligence, robotics, and computer vision were instrumental in shaping my interests and the direction of this work. I also thank Prof. Dr. Conen for serving as the second examiner and for providing valuable insights on how to evaluate and structure this project. My appreciation extends to my colleagues at the institute, whose discussions, perspectives, and careful reading of drafts contributed to the clarity and refinement of this thesis. Finally, I wish to express my deepest gratitude to my father and my grandmother, whose continuous support and encouragement have accompanied me throughout my studies.

Contents

Acronyms

CA Cellular Automata. i, 2, 4, 21–23, 30, 31, 51, 81

CLC CORINE Land Cover. 33, 49

CNN Convolutional Neural Network. 24

DDPG Deep Deterministic Policy Gradient. 13

DQN Deep Q-Network. 23

GAE Generalized Advantage Estimation. 10, 11

GDAL Geospatial Data Abstraction Library. 49

HRL Hierarchical Reinforcement Learning. i, 3, 14–16, 20, 31, 78, 79, 81, 82

IQL Implicit Q-Learning. 11, 12, 31, 41, 45, 47, 50, 51, 74

ISD Independent Subtask Discovery. 19

KL Divergence Kullback–Leibler Divergence. 50, 51, 68, 70, 71, 73, 76, 81, 87

LHP Learning Hierarchical Policies. 16, 19

MADDPG Multi-Agent Deep Deterministic Policy Gradient. 27, 28

MADQN Multi-Agent Deep Q-Network. 26

MAHRL Multi-Agent HRL. 19

MARL Multi-Agent Reinforcement Learning. 19, 27, 29

MDP Markov Decision Process. 5, 15, 17, 24, 25

OOD out-of-distribution. 9, 11, 12

OSM OpenStreetMap. 49

POMDP Partially Observable Markov Decision Process. 25

PPO Proximal Policy Optimization. i, 10, 11, 31, 41, 45, 47, 50, 57–59, 61, 63, 65, 66, 68, 71, 74, 79, 81

RL Reinforcement Learning. 2–5, 9–11, 14, 18, 19, 23–25, 30, 31

RND Random Network Distillation. 44

ROSHAN Reinforcement-learning Oriented Simulation for Hierarchical Agent Navigation. i, 3, 30, 33, 34, 44, 45, 48–50, 53, 57, 81

SDL2 Simple DirectMedia Layer 2. 49

SMDP semi-Markov Decision Process. 15, 17–19

TD temporal-difference. 10

TD3 Twin Delayed Deep Deterministic Policy Gradient. 13, 31, 41, 45, 47, 50, 51, 57–59, 74, 81

TransferHRL Transfer Learning with HRL. 20

TRPO Trust Region Policy Optimization. 10

TTE Time-to-Extinguish. i, 56, 57, 77–81

UAV Unmanned Aerial Vehicle. i, 1–3, 21–23, 25–28, 30, 35, 81, 83

UNI Unification with Subtask Discovery. 19

1 Introduction

1.1 Motivation

Wildfires represent one of the most destructive natural hazards worldwide and have become an increasingly critical challenge, particularly across Europe's Mediterranean regions. Over the past decades, their frequency, intensity, and duration have grown significantly due to land-use transformation and prolonged drought periods due to climate change [1, 2, 3]. In recent years, several catastrophic fires in southern Europe have underlined the urgency of improving prevention, detection, and suppression strategies. Forecasts indicate that both the burned area and the length of the fire season will continue to expand under future climate scenarios [1]. These fires cause extensive ecological degradation through erosion, soil alteration, and biodiversity loss, while also threatening human lives, settlements, and infrastructure [1, 2, 3]. The protection of forests is therefore implemented through an integrated three-phase approach consisting of *prevention*, *detection*, and *suppression* [4]. Prevention activities mostly include reducing the human-caused fire occurence. Detection focuses on early identification of wildfires, whereas suppression includes both direct and indirect firefighting operations, typically carried out by ground units and aerial support.

While traditional firefighting methods remain the backbone of wildfire management, they are often constrained by safety, cost, and response-time limitations. Direct extinguishing operations expose personnel to extreme danger due to proximity to the fire front, and aerial firefighting, though effective, comes with high operational costs. UAVs and autonomous robotic systems are promising technologies to complement human operations in the context of firefighting. Their ability to rapidly survey large areas and reach otherwise inaccessible terrain makes them useful for both *detection* and *suppression* tasks.

1.2 Problem Statement

Designing a fully autonomous wildfire suppression system based on UAVs remains an open research problem. Such a system must coordinate multiple aerial agents capable of detecting, navigating, and extinguishing fires in complex, dynamic, and uncertain environments. One difficulty lies in enabling these agents to make effective decisions under changing fire conditions while maintaining safety, efficiency, and coordination. Testing such capabilities in real wildfire scenarios is both dangerous and logistically difficult, preventing systematic experimentation in real world scenarios. Consequently, research in this area depends on realistic and controllable simulation environments that can capture the essential dynamics of wildfire behavior and agent interaction.

A challenge in building such environments is modeling the fire itself with sufficient fidelity while maintaining computational efficiency. CAs have proven particularly suitable for this purpose, offering a balance between physical realism and performance. Their discrete, grid-based formulation allows simulation of wildfire spread across heterogeneous landscapes influenced by fuel type, wind, and topography. However, while such models provide a reliable physical foundation, they do not by themselves address *how* autonomous agents should act within this environment.

The problem, therefore, extends beyond simulating fire dynamics to learning effective suppression strategies. Manually designing control policies for UAVs in such nonlinear and stochastic environments is difficult. Instead, agents must be able to learn adaptive behaviors through interaction with the simulated environment. Reinforcement Learning (RL) provides a framework for this by allowing agents to acquire goal-directed strategies that maximize long-term performance through trial and error. In this work, RL is used to train autonomous UAV agents for fire detection and suppression, while the CA serves as the underlying wildfire simulation model that defines their environment and interactions.

1.3 Thesis Outline

This thesis is structured into six chapters.

Chapter 1 – Introduction. Outlines the motivation and research problem of autonomous wildfire suppression using UAVs. It discusses the limitations of traditional firefighting methods and motivates the use of RL for autonomous decision-making in complex and dynamic fire environments.

Chapter 2 – Background. Provides the theoretical foundation of the work. It briefly reviews wildfire simulation models and introduces key RL concepts. The chapter concludes with an overview of the used algorithms and the principles of HRL.

Chapter 3 – Related Work. Examines four representative works on UAV based wildfire suppression to assess feasibility, system design considerations, and coordination strategies relevant to ROSHAN.

Chapter 4 – Methodology. This chapter presents the ROSHAN simulation framework, its hierarchical control structure, and the associated observation, action, and reward designs. It also summarizes the software architecture, code organization, and key environment and agent configurations, as well as the integration of multiple RL algorithms into a unified training pipeline.

Chapter 5 – Evaluation. Presents the experimental setup, evaluation metrics, and baselines. The performance of the proposed hierarchical agents is analyzed across multiple experiments, each followed by an interpretation of the results.

Chapter 6 – Conclusion. Summarizes the key findings and contributions, discusses lessons learned, and highlights directions for future improvements of the implemented system.

2 Background

2.1 Wildfire Simulation Models

Simulation models are an essential tool for understanding and managing wildfire dynamics, supporting applications from risk assessment to evacuation planning. Three broad model families are commonly distinguished: *empirical*, *semi-empirical*, and *physically based* approaches [5, 6].

Empirical models use statistical relationships derived from historical fire data to predict fire behaviour. They are simple and fast, making them valuable for real-time decision support, but their reliance on past data limits accuracy in novel conditions. Examples include *McArthur's model for Australian grassland and forest fires* and the *Canadian Forest Fire Behavior Prediction System* [7].

Semi-empirical models combine empirical observations with simplified physical relationships, striking a balance between realism and computational cost. *Rothermel's spread model* [8] is a classic example and underpins software such as *FARSITE* [9] and *Prometheus* [10], which are widely used for operational fire management.

Physically based models aim at highest fidelity by directly simulating heat transfer, combustion, and atmospheric dynamics. Systems such as the *Coupled Atmosphere–Wildland Fire–Environment model* [11] or the *Wildland Fire Dynamics Simulator* [12] can reproduce complex fire–atmosphere feedbacks, but demand extensive input data and significant computing power, which limits their use for rapid decision-making.

A complementary and computationally lighter approach is provided by CA. CA models discretise the landscape into cells that update in parallel according to simple neighbourhood rules, yet generate complex fire-spread patterns[13, 14]. Their efficiency makes them attractive for interactive or RL scenarios where many thousands of simulations are required, and they can be extended with stochastic or hybrid techniques to capture wind, topography, or ember transport [15, 16]. From fast empirical tools for immediate response, through semi-empirical models for operational planning, to high-resolution physical simulators for research, CA sit slightly aside this spectrum, offering a flexible compromise between simplicity, speed, and the ability to incorporate more detailed physics when needed.

2.2 Reinforcement Learning

RL is a branch of machine learning concerned with sequential decision-making through interaction and feedback. Unlike supervised learning, which relies on labeled data, or unsupervised learning, which uncovers patterns in unlabeled data, RL centers on an *agent* that learns by trial and error in a dynamic environment. Instead of a fixed

training set, the agent gathers knowledge from its own experience, gradually improving its behavior to maximize long-term reward. At the core of RL is a continuous feedback loop in which the agent:

- observes the current state S_t of the environment,
- selects and executes an action A_t according to a decision strategy,
- receives a scalar reward R_t reflecting the immediate outcome,
- and moves to a new state S_{t+1}.

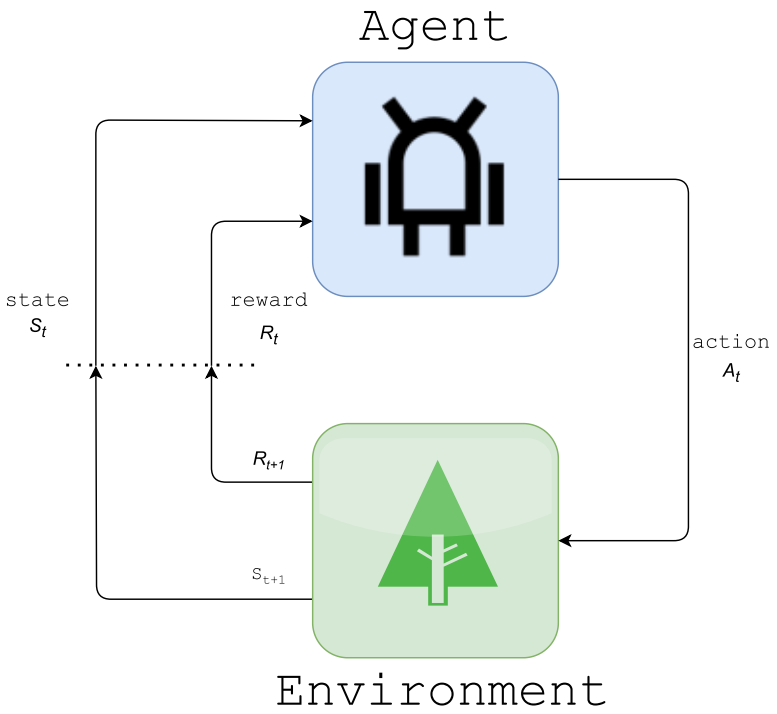


Figure 2.1: Reinforcement learning paradigm. The *agent* perceives the environment through *states*, which describe the current situation.

This cycle drives the agent to improve its decisions, so that the cumulative reward over time is maximized. The *agent* acts as a decision-maker that chooses actions and the *environment* is everything outside the agent that evolves in response to those actions (see Figure 2.1). Each action influences future states and rewards, creating a closed feedback loop. This interaction may involve uncertainty, delayed effects, and changing conditions, all of which make RL suitable for complex control tasks [17].

2.2.1 Markov Decision Process

A large part of the theoretical foundation of RL is provided by the Markov Decision Process (MDP). A MDP formalizes sequential decision-making in an environment with stochastic dynamics and serves as the mathematical framework for most RL algorithms [17, 18].

A MDP is defined by the tuple

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, r \rangle,$$

where

- $\mathcal{S}$ is the set of possible **states** describing the environment.
- $\mathcal{A}$ is the set of possible **actions** available to the agent.
- T is the **state-transition probability function**: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, such that $T(s, a, s') = P(s'|s, a)$, i.e., the probability of reaching state s' when taking action a in state s.
- r is the **reward function**: $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $r(s, a)$ is the scalar cost or reward of taking action a in state s.

The *Markov property* states that the next state and reward depend only on the current state and action, not on the sequence of past states:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t).$$

This assumption simplifies analysis and enables efficient algorithms.

2.2.2 Return and Objective

Assuming the agent starts at $s_0 \in \mathcal{S}$ and continues to take actions, it will result in trajectory:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, r_T)$$

Note that (s_t, a_t, r_t) for state, action and reward at time t, r_t is the immediate reward resulting from executing action a_t in state s_t. The *expected return* G_t of such a trajectory is the expected total reward obtained:

$$G_t = r_t + r_{t+1} + r_{t+2} + ... = \sum_{k=0}^{T-t} r_{t+k} = \sum_{k=0}^{T-t} r(s_{t+k}, a_{t+k}).$$

For continuing tasks where $T \to \infty$, G_t may diverge. To keep the objective finite and to emphasize near-term rewards, a **discount factor** $\gamma \in [0, 1]$ is introduced:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k} = \sum_{k=0}^{\infty} \gamma^k r(s_{t+k}, a_{t+k}).$$

This also helps to balance the importance of immediate and future rewards, even in finite trajectories.

2.2.3 Policy and Value Functions

A *policy* π describes the agent's decision-making strategy. For every state $s \in \mathcal{S}$ it specifies a probability distribution over actions,

$$\pi(a|s) = P(A_t = a \mid S_t = s),$$

i.e., the probability of selecting action a when the agent observes state s. Policies can be *deterministic*, where a single action is chosen with probability one, or *stochastic*, which allows for randomness and is often advantageous in environments with uncertainty or when exploration is required.

The quality of a policy is measured by the expected return it achieves when interacting with the environment. Starting from state s, the *value function*

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

captures the long-term reward when following π. Likewise, the *action-value function* measures the expected return when taking action a in state s and then continuing with π

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

The goal of reinforcement learning is to discover an *optimal policy* π^* that maximizes this expected return for all states:

$$\pi^* = \arg\max_\pi V^\pi(s), \qquad \forall s \in \mathcal{S}.$$

Equivalently, using the optimal action-value function Q^*, an optimal policy selects actions

$$\pi^*(s) \in \arg\max_a Q^*(s,a), \qquad \forall s \in \mathcal{S}.$$

An optimal policy balances immediate and future rewards, and satisfies the principle of optimality, which means if π^* is optimal, then the remaining decisions must also form an optimal policy from every successor state onward.

2.2.4 Bellman Equation

The key property that makes these functions computationally useful is their *recursive* structure. Because the process satisfies the Markov property, the expected return from a state can be decomposed into the immediate reward and the discounted value of the next state:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'} P(s'|s,a)\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'} P(s'|s,a)V^\pi(s')\Big]. \end{aligned}$$

Likewise, the action–value function satisfies

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') Q^{\pi}(s',a').$$

This relationship is known as the *Bellman equation* and expresses the value of a state or action in terms of the values of successor states and actions. Dynamic programming, temporal-difference learning, Q-learning, and modern policy gradient methods all rely on the Bellman equation to iteratively estimate value functions and to improve policies towards optimality.

2.2.5 Advantage Function

While value functions $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ measure the long-term quality of states or state–action pairs under a policy π, it is often useful to express their difference. The *advantage function* quantifies how much better (or worse) it is to take a specific action compared to the average action prescribed by the policy in a given state:

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

If $A^{\pi}(s,a) > 0$, action a is better than the policy's expected behavior at state s and if $A^{\pi}(s,a) < 0$, it is worse [19].

2.2.6 Exploration–Exploitation Dilemma

To find an optimal policy, an agent must both *explore* unknown actions to gain information about rewards and transitions, and *exploit* current knowledge to maximize rewards. Pure exploitation may quickly converge to a suboptimal policy if early experiences are misleading, while pure exploration wastes opportunities to collect reward. An effective agent balances the two, devoting effort to exploration when its knowledge is uncertain and shifting towards exploitation as learning progresses. There are many ways to balance exploration and exploitation, some built directly into the architecture design or the algorithm, others are simpler, easily implemented mechanisms. Some examples follow to give an idea:

Epsilon-greedy action selection: With probability $1-\varepsilon$, the agent chooses the action with the highest estimated value (greedy exploitation). With probability ε, it picks a random action to explore alternatives. Annealing ε over time allows the policy to become more deterministic as confidence grows.

Entropy regularization: In policy-gradient and actor–critic methods, adding an entropy term to the loss encourages stochasticity in the policy. A higher entropy coefficient promotes diverse action sampling early in training, while the coefficient can be reduced later to favor exploitation.

Intrinsic motivation and curiosity-driven exploration: In sparse-reward or deceptive environments, agents benefit from intrinsic signals that reward state space novelty or information gain. Curiosity-based approaches [20, 21] provide internal bonuses proportional to prediction error or state novelty, guiding the agent toward unexplored regions of the state space. Other methods compute exploration bonuses based on pseudo-counts [22] or prediction uncertainty [23].

Choosing and tuning an exploration strategy depends on the task's complexity and reward structure. Environments with dense, frequent rewards may need only mild stochasticity, whereas sparse or deceptive environments often require sophisticated exploration incentives.

2.2.7 Online vs. Offline

In the classical *online* RL setting, the agent actively interacts with the environment during training, continually collecting new experience and updating its policy. This interaction loop, together with the recursive structure of value functions given by the Bellman equation, forms the standard paradigm of RL described above. Online RL thus allows continual policy improvement, but in practice may require large amounts of interaction, which can be costly, time-consuming, or unsafe in real-world systems.

In contrast, *offline* RL (also called batch RL) trains agents using a fixed dataset of past interactions, without further environment queries. This makes offline RL suitable for domains such as autonomous driving or robotics, where exploration is dangerous or expensive. A key difficulty is the tension between *policy improvement* beyond the behavior policy that collected the dataset, and the risk of deviating too far from that behavior, which causes *distributional shift*. Estimating the value of unseen, out-of-distribution (OOD), actions can lead to severe overestimation and instability [24]. Offline RL algorithms therefore focus on constraining the policy or modifying the learning targets to remain close to the support of the dataset.

2.2.8 Policy Gradient

Policy-gradient methods directly optimize a parameterized policy $\pi_\phi(a \mid s)$ by ascending an estimate of the gradient of the expected return, given by the policy gradient theorem:

$$\nabla_\phi J(\pi_\phi) = \mathbb{E}_{\pi_\phi}\left[\sum_{t=0}^{\infty} \Psi_t \, \nabla_\phi \log \pi_\phi(a_t \mid s_t)\right].$$

.

The return estimator Ψ_t can be chosen to be:

- $\sum_{t=0}^{\infty} r_t$: Total reward of the trajectory.
- $\sum_{t'=t}^{\infty} r'_t$: Total reward after action a_t.
- $\sum_{t'=t}^{\infty} r'_t - b(s_t)$: Baselined version of total reward after action a_t.
- $Q^\pi(s_t, a_t)$: State-Action Value function.

- $A^\pi(s_t, a_t)$: Advantage function.
- $r_t + \gamma V^\pi(s_t + 1) - V^\pi(s_t)$: temporal-difference (TD) residual.

While these methods provide an unbiased estimate of the policy gradient, they can suffer from high variance and instability if the policy parameters are updated too aggressively. To mitigate this, trust-region and clipped-objective approaches such as Trust Region Policy Optimization (TRPO) and PPO constrain the step size or ratio between old and new policies, stabilizing learning and improving sample efficiency [25].

2.3 Algorithms

2.3.1 Proximal Policy Optimization

PPO is a widely used on-policy RL algorithm belonging to the family of policy-gradient methods. PPO is designed for environments where direct interaction is possible and stable policy improvement is required. It was introduced by Schulman et al. as a simplified alternative to TRPO, removing the need of solving a complex constrained optimization at each update while offering comparable performance[25].

PPO implements a *clipped surrogate objective*, which balances between sufficient policy improvement and stable updates. The probability ratio is defined by

$$r_t(\phi) = \frac{\pi_\phi(a_t \mid s_t)}{\pi_{\phi_{\text{old}}}(a_t \mid s_t)},$$

comparing the new policy to the old one under which the data was collected. The clipped objective then is

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

where ϵ is a small hyperparameter (e.g. 0.1–0.3). This prevents $r_t(\phi)$ from deviating too far from 1, effectively bounding the policy update size.

In practice, PPO combines several components into its learning procedure:

- **Clipped policy-gradient loss.** As described above, the clipped surrogate ensures stable updates while still enabling improvement when the policy change is modest. This avoids catastrophic collapse when advantage estimates are noisy.
- **Value function learning.** PPO trains a parameterized value network $V_\psi(s)$ by regression to the empirical return (or a bootstrapped target), minimizing

  $$L_V(\psi) = \mathbb{E}_t\Big[(V_\psi(s_t) - \hat{R}_t)^2\Big],$$

  where $\hat{R}_t$ is a discounted return or Generalized Advantage Estimation (GAE) target.

- **Entropy regularization.** To encourage sufficient exploration, PPO augments its loss with an entropy bonus:

$$L_H(\phi) = \mathbb{E}_t\Big[-\beta\,\mathcal{H}(\pi_\phi(\cdot \mid s_t))\Big],$$

  where β is a weighting coefficient. This term prevents premature convergence to deterministic policies.

The full PPO objective is thus a weighted sum of the clipped policy loss, the value loss, and the entropy term:

$$L^{\text{PPO}}(\phi, \psi) = L^{\text{CLIP}}(\phi) + c_1 L_V(\psi) + c_2 L_H(\phi).$$

PPO is typically trained in iterations, a batch of trajectories is collected using the current policy $\pi_{\phi_{\text{old}}}$, advantages are estimated (often using GAE), and multiple epochs of stochastic gradient descent are performed on this fixed dataset while keeping ϕ_{old} frozen for computing $r_t(\phi)$. After updates, ϕ_{old} is replaced by ϕ, and a new batch of trajectories is sampled.

2.3.2 Implicit Q-Learning

Implicit Q-Learning (IQL) is an offline RL-algorithm that aims to learn good policies from a fixed dataset without requiring querying the value of OOD actions. In online Q-learning or actor–critic methods, one typically uses a Bellman backup

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

or alternatively sampling from a policy for backup. However, in the offline RL setting, performing a $\max_{a'}$ over all actions (including those not seen in the dataset) can lead to *extrapolation error*, the Q-network may produce wildly optimistic values on OOD actions, destabilizing learning. Many offline RL methods therefore add explicit regularization or constraints to keep the learned policy close to the data distribution (e.g. via Kullback Leibler penalties, behavior cloning terms, or pessimism) [24, 26].

IQL never uses unseen actions in its backups. Instead, it uses an in-sample value estimation, combined with a specially chosen expectile loss to *implicitly* favor higher-value actions among those seen in the dataset, and then extracts a policy via advantage-weighted regression. IQL was first introduced by Kostrikov et al. under the title *Offline Reinforcement Learning with Implicit Q-Learning* [26].

The dataset is collected by some behavior policy π_β and contains a fixed set of transitions $\mathcal{D} = \{(s, a, r, s')\}$. IQL uses a parameterized critic Q_θ and value network V_ψ, which are updated every gradient step and a target critic $Q_{\hat{\theta}}$ which is a slow-moving copy of Q_θ. IQL has three main components:

- **Value estimation via expectile regression.** IQL sets $V_\psi(s)$ to predict the *upper expectile* of the distribution of $r + \gamma \max_{a'} Q_{\hat{\theta}}(s', a')$ drawn from the datasets action distribution. This results in the value objective

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[\ell_\tau\Big(Q_{\hat{\theta}}(s, a) - V_\psi(s)\Big)\Big],$$

where

$$\ell_\tau(u) = \begin{cases} \tau\, u^2 & \text{if } u \geq 0, \\ (1-\tau)\, u^2 & \text{if } u < 0, \end{cases}$$

with $\tau \in (0, 1)$. This down weights the contributions of $Q_{\hat{\theta}}(s, a)$ values smaller than $V_\psi(s)$ while giving more weights to larger values. Once $V_\psi(s)$ is fit, it implicitly encodes an upper expectation of in-dataset Q-values.

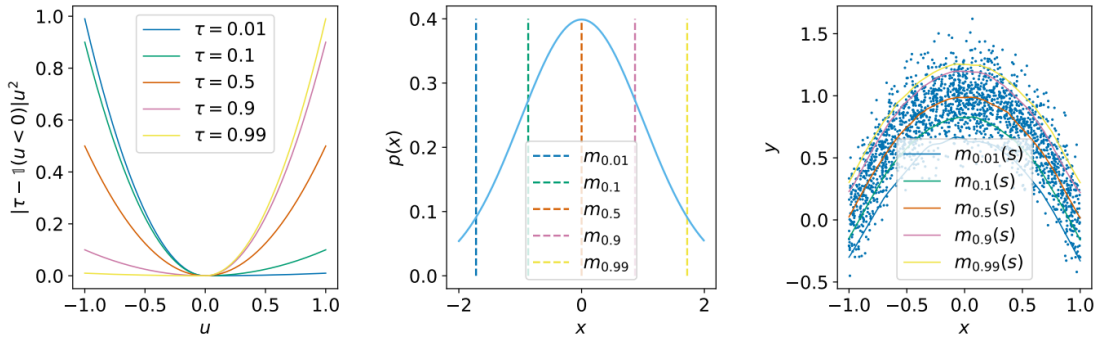


Figure 2.2: Left: Illustration of the asymmetric squared loss used in expectile regression. A setting of $\tau = 0.5$ recovers the standard mean squared error, whereas a larger value, such as $\tau = 0.9$, places more weight on positive differences. **Center:** Expectiles of a normal distribution for different values of m_τ. **Right:** Example of conditional expectile regression for a two-dimensional random variable. For each input x a distribution over y values is obtained. While $\tau = 0.5$ estimates the conditional mean, choosing $\tau \approx 1$ yields an approximation of the maximum operator over in-support values of y[26].

- **Q-function update.** Having V_ψ, IQL updates the Q-network with:

$$L_Q(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\,(r + \gamma\, V_\psi(s') - Q_\theta(s,a))^2\Big].$$

  Both losses never require new action sampling and only use the actions from the dataset. Hence, no OOD actions are evaluated.

- **Policy extraction via advantage-weighted behavior cloning.** To extract a policy $\pi_\phi(a \mid s)$ from the learned critics, IQL uses a *weighted regression* objective:

$$L_\pi(\phi) = -\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\exp\Big(\alpha \cdot (Q_{\hat{\theta}}(s,a) - V_\psi(s))\Big)\, \log \pi_\phi(a \mid s)\right],$$

  where $\alpha \in [0, \text{inf})$ is a temperature hyperparameter. For small values this behaves like behavioral cloning, for larger values it tries to maximize the Q-Function.

The full algorithm alternates between gradient updates of V_ψ and Q_θ, with soft target updates of $Q_{\hat{\theta}}$ after each step. The policy can be extracted either only after the critics have converged or intermittently during training, since the critic losses themselves do not depend on π_ϕ. Performing intermittent policy extraction allows IQL to seamlessly transition to *online fine-tuning*. After the purely *offline* phase has produced an initial policy, this policy can be deployed to gather fresh experience, which is then added to

the dataset. Although the value update equations remain unchanged, the continually expanding dataset means that the critics are indirectly refined by the new on-policy transitions, enabling further improvement beyond the fixed offline data.

2.3.3 Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an off-policy actor–critic algorithm for continuous action spaces that extends Deep Deterministic Policy Gradient (DDPG) [27]. Introduced by Fujimoto et al. as *Addressing Function Approximation Error in Actor–Critic Methods* [28], it reduces instability and overestimation bias common in deterministic actor–critic approaches.

Like other actor–critic methods, TD3 maintains a deterministic policy (the *actor*) $\pi_\phi(s)$ and two action–value functions (the *critics*) Q_{θ_1} and Q_{θ_2}. The critics are trained on off-policy data from a replay buffer $\mathcal{D}$, while the actor is optimized to maximize the Q-values predicted by the critics. A key issue in DDPG is the systematic overestimation of Q-values, which can lead to divergent policies. TD3 addresses this with three core techniques:

- **Target policy smoothing.** Instead of directly using the target action $\pi_{\phi'}(s')$ in the Bellman backup, TD3 adds small clipped Gaussian noise to encourage smoother value estimates:

$$a'(s') = \text{clip}(\pi_{\phi'}(s') + \epsilon, a_{\min}, a_{\max}), \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c).$$

  This prevents the actor from exploiting local overestimation spikes and ensures smoother Q-functions. The added clipped noise is also clipped to a valid action range, to avoid impossible actions.

- **Clipped Double Q-learning.** Two critics are trained independently, and the smaller of their target estimates is used in the Bellman update to reduce overestimation bias:

$$y(r, s') = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_{\phi'}(s') + \epsilon),$$

  where θ_i' and ϕ' are target network parameters. Each critic minimizes the squared Bellman error:

$$L_{Q_i}(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\Big(Q_{\theta_i}(s,a) - y(r,s')\Big)^2\Big].$$

- **Delayed policy updates.** The actor is updated less frequently than the critics (e.g. every d critic updates, with $d = 2$ in the original paper). This ensures that the Q-functions are more accurate before being used to update the policy. The actor is trained with the deterministic policy gradient:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s\sim\mathcal{D}}\Big[\nabla_a Q_{\theta_1}(s,a)\Big|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)\Big].$$

Before training starts the critic networks Q_{θ_1}, Q_{θ_2} and the actor network π_ϕ are initialized with random parameters and their parameters are copied to their target network counterparts Q'_{θ_1}, Q'_{θ_2} and π'_ϕ. The replay buffer $\mathcal{D}$ is initialized with sufficient data.

The training loop then alternates between selecting action a from $\pi_\phi(s)$ with exploration noise and storing the transition tuple (s, a, r, s') in $\mathcal{D}$, then a mini-batch of N transitions is sampled from $\mathcal{D}$ and both critics are updated with the clipped double Q-learning loss. Every d of steps the actor is updated with the deterministic policy gradient. At this point the target networks are also soft-updated, with a slow-moving update rate, parameterized by τ:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

2.4 Hierarchical Reinforcement Learning

HRL extends classical RL by introducing temporal and structural abstractions into the decision-making process. Instead of directly learning a mapping from states to primitive actions, HRL decomposes long-horizon problems into hierarchies of *subtasks*, each governed by its own policy. A higher-level policy selects among these subtasks, effectively choosing *what* to do, while lower-level policies decide *how* to do it. This decomposition allows agents to operate over multiple timescales and to reuse learned skills across tasks, being more efficient in exploration and learning in complex environments [29].

Many real-world tasks involve large state and action spaces as well as long temporal horizons, making exploration and credit assignment difficult for standard reinforcement learning methods. An agent attempting to learn a policy directly in such a space faces exponential growth in the number of possible trajectories, often leading to poor data efficiency and unstable learning. HRL overcomes these challenges by introducing *temporal abstraction*, the ability to reason in terms of extended sequences of actions, or *options*, that persist for multiple timesteps. By defining higher-level decisions over temporally extended subtasks, HRL reduces the effective horizon of the learning problem and allows rewards to be propagated over longer timescales. At the same time, lower-level subtasks typically operate on smaller state spaces with denser feedback, making them easier to learn. This hierarchical organization has several advantages:

- **Improved exploration:** Subtasks act as structured macro-actions that encourage the agent to explore semantically meaningful regions of the environment rather than individual low-level transitions.
- **Efficient credit assignment:** Temporal abstraction allows the agent to associate long-term outcomes with higher-level decisions, facilitating learning in sparse-reward settings.
- **Reusability and transfer:** Learned subtasks or skills can be reused across related tasks, enabling transfer learning and reducing sample complexity.
- **Interpretability:** The hierarchical decomposition offers a natural way to describe and inspect the behavior of complex agents at multiple levels of abstraction.

Because of these properties, HRL has been shown to outperform flat RL in domains such as continuous control [30, 31], long-horizon games [32], and robotic manipulation [33, 34].

2.4.1 Formalization

HRL can be described as a hierarchy of policies operating within a semi-Markov Decision Process (SMDP). Unlike in MDPs where the transition between states happen at fixed intervals, state transitions in SMDPs can happen in variable amounts of time [35]. The higher-level policy π_Γ chooses subtasks $\omega \in \Omega$, where Ω is the set of subtasks, each defined by a tuple

$$\omega = \langle I_\omega, \pi_\omega, \beta_\omega, r_\omega, g_\omega \rangle,$$

where I_ω denotes its initiation set, π_ω the subpolicy, β_ω the termination condition, r_ω a subtask-specific reward and g_ω a subgoal representation. Each subtask itself may invoke further subtasks or primitive actions, forming a hierarchy of depth L.

The optimal value function of the hierarchical agent extends the standard Bellman equation to the SMDP setting, where each high-level decision (a subtask ω_t) may last for multiple timesteps. When the agent is in state s_t and selects a subtask ω_t, that subtask is executed for a variable duration c_{ω_t} (determined by its termination condition β_{ω_t}). During this period, the lower-level policy π_{ω_t} generates a sequence of primitive actions and intermediate states, accumulating an expected return

$$R(s_t, \omega_t) = \mathbb{E}_{a \sim \pi_{\omega_t}} \left[\sum_{i=0}^{c_{\omega_t}-1} \gamma^i r(s_{t+i}, a_{t+i}) \,\middle|\, s_t, \omega_t \right].$$

After the subtask terminates, the environment transitions to a new state $s_{t+c_{\omega_t}}$. The joint probability of both the next state and the duration of the subtask is expressed as

$$P(s', c_{\omega_t} \mid s_t, \omega_t) = P(s' \mid s_t, \omega_t, c_{\omega_t})\, P(c_{\omega_t} \mid s_t, \omega_t),$$

where $P(s' \mid s_t, \omega_t, c_{\omega_t})$ is the probability of reaching state s' after executing ω_t for exactly c_{ω_t} timesteps, and $P(c_{\omega_t} \mid s_t, \omega_t)$ is the probability distribution over possible durations before termination. This joint transition model captures both the temporal and stochastic nature of hierarchical actions in a SMDP.

Using these definitions, the optimal hierarchical action–value function can be expressed as

$$Q(s_t, \omega_t) = R(s_t, \omega_t) + \sum_{s', c_{\omega_t}} \gamma^{c_{\omega_t}} P(s', c_{\omega_t} \mid s_t, \omega_t) \max_{\omega'} Q(s', \omega'),$$

which generalizes the Bellman optimality equation to temporally extended actions. $R(s_t, \omega_t)$ is the expected cumulative reward of the current subtask, and the second term is the discounted expected value of the next subtask ω' chosen after termination, weighted by the probability of ending in each successor state s' and the duration c_{ω_t}. The exponent $\gamma^{c_{\omega_t}}$ correctly discounts future rewards over the entire execution time of the subtask. The overall optimization objective can be expressed as

$$\Omega^*, \pi^*_{\text{hier}} = \arg\max_{\Omega} \arg\max_{\pi_{\text{hier}}|\Omega} Q_{\text{hier}}(s, a),$$

which captures the challenge of *learning hierarchical policies* and *discovering useful subtasks*. Automatic subtask discovery is not essential when subtasks are handcrafted beforehand. Learning hierarchical policies can either be done sequential, one hierarchy level at a time, or they could be learned simultaneously in an end-to-end manner.

2.4.2 Taxonomy of HRL Approaches

According to Pateria et al. [29] HRL can be aranged into five general categories along three key dimensions:

1. Subtasks are *manually defined* or *automatically discovered*?
2. Framework involves a *single* agent or *multiple* agents?
3. Learning occurs on a *single* task or across *multiple* tasks?

(1) Learning Hierarchical Policies (LHP). This approach assumes that subtasks are predefined, for single agent and single task setting. Since this is *mostly* the setting used in this thesis, this section is described in more detail than the other approaches. It includes the two subclasses *Feudal Hierarchies* and *Policy Tree Approaches*.

Feudal Hierarchies

Feudal Reinforcement Learning [36] introduced one of the earliest formalizations of hierarchical control in reinforcement learning. Inspired by the structure of a feudal hierarchy, the model defines a recursive *manager–worker* organization in which each manager sets tasks for its subordinate workers, who in turn may act as managers for lower levels. Only the lowest-level workers directly interact with the environment, while higher levels operate over progressively more abstract state and temporal resolutions. In this framework, a manager issues commands in the form of *subgoals*, which are treated as local tasks by the worker. Each worker maintains its own Q-values to learn how to satisfy its manager's subgoals and is rewarded solely based on its success in completing these local tasks, regardless of whether they ultimately advance the manager's own objective. This mechanism, referred to as *reward hiding*, allows each level to learn autonomously, sub-managers learn to obey their immediate supervisors, while higher levels learn which subgoals best serve their own goals. Similarly, *information hiding* ensures that each level only observes the state space relevant to its decisions, abstracting away unnecessary details from lower or higher levels. This separation of control across abstraction levels divides the problem of learning into coordinated but independent subproblems, higher layers focus on *what to achieve*, while lower layers focus on *how to achieve it*.

Kulkarni et al. [32] introduced h-DQN, a HRL framework that integrates temporal abstraction with intrinsic motivation. The architecture consists of two deep Q-networks operating at different temporal scales. A *meta-controller* that selects goals (such as spatial locations or objects) and a *controller* that executes primitive actions to achieve them. Each goal defines an intrinsic reward function, allowing the agent to learn behaviors even in environments with extremely sparse external feedback. The

controller is trained to maximize intrinsic rewards for achieving the current goal, while the meta-controller maximizes cumulative extrinsic rewards by learning which goals lead to long-term success. This structure decomposes complex exploration problems into manageable subgoals and enables stable learning via separate replay buffers and update frequencies for each level.

A critical challenge in multi-level architectures is *non-stationarity*, as the lower-level policy evolves, the same high-level subgoal may yield different outcomes over time, destabilizing the higher-level learning process. Nachum et al. [31] introduced HIRO (*Hierarchical Reinforcement Learning with Off-Policy Correction*), a two-level continuous-control framework designed to address the instability that arises when high-level and low-level policies learn concurrently. In HIRO, the higher-level policy outputs subgoals in the agent's state space, and the lower-level policy is trained to reach these subgoals within a fixed horizon. To correct for non-stationarity, HIRO introduces an *off-policy subgoal relabeling* mechanism. For each stored high-level transition, the original subgoal g_t is retrospectively replaced with a new subgoal $\tilde{g}_t$ that best explains the observed transition according to the current low-level policy. This hindsight correction ensures that the replayed transitions remain consistent with the low-level policy's evolving behavior, allowing stable and sample-efficient off-policy training of both hierarchical levels.

Jiang et al. [37] introduced HAL (*Hierarchical Abstraction with Language*), a framework that uses natural language as the abstraction between hierarchical policies. Instead of representing subgoals as numerical coordinates or latent vectors, the higher-level policy $\pi_h(g \mid s)$ outputs *language instructions* g (e.g., "place the red sphere to the left of the blue cube"), which parameterize the low-level policy $\pi_\ell(a \mid s, g)$. A Boolean function $\Psi(s, g) \in \{0, 1\}$ evaluates whether a given instruction is satisfied in state s, serving as a binary intrinsic reward signal for the low-level controller. This language-conditioned hierarchy offers several advantages. Language provides a flexible, compositional representation of subgoals, allowing high-level actions to correspond to abstract relational concepts rather than single states. It also enables policy reuse and very good human interpretability, because high-level behaviors can be expressed, analyzed, and even modified through natural-language instructions.

Policy Tree Approaches

Complementary to *Feudal Hierarchies* are the *Policy Tree Approaches*, where higher levels select among predefined subpolicies rather than goal vectors.

The Options Framework [35] provides the formal foundation for temporal abstraction in reinforcement learning and serves as the theoretical basis of most hierarchical approaches that followed. It extends the conventional MDP paradigm to a SMDP described in Section 2.4.1, where actions can persist over variable time intervals. Rather than reasoning only in terms of primitive actions, an agent can choose among *options*, temporally extended behaviors that encapsulate internal policies and termination conditions. Each option represents a closed-loop subpolicy that can execute for several timesteps before returning control to the higher-level decision process. This allows learning and planning to occur simultaneously at multiple temporal scales, short-horizon

controllers manage fine-grained actions, while long-horizon policies reason over extended activities. Primitive actions are thus treated as one-step options.

The key contribution of the framework is to show that any fixed set of options defined over an environment induces a SMDP. Therefore standard RL algorithms such as Q-learning or policy iteration can be applied without modification, except that each update now spans the entire execution of an option rather than a single transition. This effectively compresses the temporal resolution of learning, value propagation and credit assignment occur over macro-actions instead of individual steps, leading to substantially faster convergence in structured tasks. Conceptually, options enable *hierarchical planning* by providing reusable subroutines that abstract away low-level details. For example, in a navigation task, an agent might possess options such as "traverse hallway" or "enter room," each internally composed of many primitive actions. High-level reasoning then operates at a symbolic or goal-oriented level, chaining these behaviors to solve complex problems with fewer decision points.

The framework also introduces mechanisms that make options *adaptive* rather than static:

- **Intra-option learning:** value and policy updates can occur even during an option's execution, improving sample efficiency and stability.
- **Option interruption:** options may terminate early if switching yields higher expected return, allowing the agent to act opportunistically within its hierarchy.
- **Subgoal discovery:** useful options can be discovered automatically by identifying bottleneck or frequently visited states, turning exploration statistics into reusable skills.

The MAXQ Value Function Decomposition framework [38] proposes to decompose the overall value function of a task into a set of smaller, more manageable subvalue functions. Instead of learning a single monolithic Q-function for the entire environment, MAXQ represents the value of a policy as the sum of the expected rewards from executing subtasks and the expected value of completing the remainder of the parent task. This decomposition allows each subtask to be trained independently while maintaining theoretical consistency with the global value function.

A complex task is represented as a hierarchy of subtasks, where each node in the hierarchy corresponds either to a *composite task* (a higher-level subgoal that invokes other subtasks) or a *primitive action*. Each task i is then decomposed to its own local value function $V_i(s)$ and completion function $C_i(s, a)$, which measures the expected cumulative reward of completing the parent task after finishing subtask a. The overall value function for task i is expressed as

$$Q_i(s, a) = V_a(s) + C_i(s, a),$$

where $V_a(s)$ represents the expected return from executing subtask a, and $C_i(s, a)$ estimates the remaining return needed to complete task i once a terminates. This additive decomposition preserves the Bellman consistency across levels of the hierarchy while enabling localized learning at each node.

The hierarchical structure allows subtasks to be solved and reused independently, leading to substantial improvements in sample efficiency and transferability. For instance, once

a subtask such as “pick up key” or “navigate to door” has been learned, it can be invoked by multiple higher-level tasks without retraining. This reuse of subpolicies supports modularity and facilitates transfer learning across related domains. MAXQ also separates the learning of *what* to do (the task hierarchy and decomposition structure) from *how* to do it (the parameterization of subtask policies). Each subtask is trained using its own local reward function and termination condition, consistent with the recursive execution semantics of hierarchical policies. As in the options framework, the execution of each subtask forms an SMDP, and standard RL algorithms can be applied locally within each node.

(2) Unification with Subtask Discovery (UNI). While LHP approaches rely on manually defined subtasks, unified approaches jointly learn both the subtask structure and the corresponding hierarchical policies end-to-end. Such unified frameworks eliminate the need for manual subtask design, improving generalization across unseen tasks. An example would be the *Option-Critic architecture* [39], where initiation, intra-option policies, and termination functions are learned via policy gradients derived from the main tasks rewards. While theoretically guaranteed to find an optimal policy using the policy gradients, those gradients are highly dependant on the main tasks reward, making Option-Critic perform poorly on sparse reward settings.

(3) Independent Subtask Discovery (ISD). In contrast, ISD approaches automatically discover subtasks in a *task-agnostic* manner, which means, that the task discovery process is *independant* of learning the optimal hierarchical policy. This happens often during a pretraining phase after which the discovered subtasks are then used to train the hierarchical policy on the main task. One approach of subtask discovery is to find bottlenecks in the state space, those bottlenecks are then used as subgoals for the corresponding subtask to maximize. Mc Govern et al. [40] propose a frequency-based approach for bottleneck discovery. The agent explores the state space using a random policy and collects various trajectories, each terminal state is than classified as either positive or negative, on the basis if it would be a desired goal state of a subtask. ISD methods emphasize modularity and reusability of behaviors, decoupling skill acquisition from task-specific optimization.

(4) Multi-Agent HRL (MAHRL). MAHRL extends hierarchical reinforcement learning to multi-agent systems, where several agents governed by a hierarchical policy jointly optimize a shared objective. Unlike standard Multi-Agent Reinforcement Learning (MARL), which coordinates primitive actions among agents, MAHRL decomposes a joint task into subtasks distributed across agents, allowing cooperation and specialization at higher levels of abstraction. Agents may coordinate only at the subtask level while executing their low-level policies independently. Designing MAHRL systems introduces challenges beyond those in MARL, including synchronization of subtask terminations across agents, coordination under partial observability, and non-stationarity caused by concurrent learning. Hierarchical policies also raise new issues such as subtask discovery that depends on other agents’ behaviors, and the difficulty of defining or transferring hierarchical structures in heterogeneous teams.

(5) Transfer Learning with HRL (TransferHRL). TransferHRL focuses on using hierarchically structured knowledge across multiple tasks. The idea behind transfer learning is that a certain subtasks, when solved, have learned a policy or collected experience data that can be shared in other related tasks. One problem is that an HRL agent might learn and collect massive amounts of data when transferring from one subtask to another, this is highly memory-inefficient. One early example that aims to solve this problem is H-DRLN [41], which transfers pretrained subtask policies via *multi-skill distillation*. This method compresses multiple low-level policies into a single distilled policy, improving memory efficiency and enabling multitask performance. However, H-DRLN policies rely on manually defined and manually trained subtasks and do not support continual discovery of new ones. Another problem that generally occurs in the context of TransferHRL is that the state space dimension of one subtask might not be the same when transferring to another task domain.

3 Related Work

3.1 Drone Swarms in Fire Suppression Activities: A Conceptual Framework

Ausonio et al. [42] propose a conceptual framework for a wildfire suppression system based on coordinated swarms of UAVs and fire modelling with CA. The idea is to deploy hundreds of drones capable of producing a continuous *rain-like* flow of extinguishing liquid over an active fire front. Each UAV periodically returns to a mobile support platform for automatic battery replacement and tank refilling, allowing continuous operation. It is important to note that the proposed framework does not explicitly simulate the motion dynamics or path-planning of individual UAVs within the CA environment. The UAVs are instead represented abstractly through a *continuous water flow rate* acting over a section of the fire front. The UAVs operational behavior, like battery exchange, refilling, and coordinated dispatch, is treated analytically rather than dynamically. The UAVs act as a stationary or semi-static boundary condition applying a constant extinguishing effect. The study therefore focuses on the *macroscopic effectiveness* of drone swarms (i.e., how much fireline can be contained or extinguished) rather than on individual drone trajectories, swarm coordination, or motion planning. The authors state that technical aspects of drone control, networking, and collision avoidance are left for future work.

Drone Swarm System

The envisioned system is composed of a mobile base platform that coordinates a swarm of multirotor drones (5–50 L payload each). Multiple platforms are envisioned, from which the drones are dispatched. A platform performs three key functions, automatic battery exchange and charging, automatic refilling of fire retardant and high-bandwidth communication for flight-plan updates. This design enables 24/7 autonomous operation, even in low visibility or nighttime conditions, and in rugged terrains inaccessible to manned vehicles. Although each drone carries only a small payload compared to firefighting aircraft, the aggregate and continuous discharge can achieve the desired *rain-like* effect, improving evaporation efficiency and water usage.

Critical Water Flow Rate

To evaluate feasibility, the authors developed analytical and simulation models linking fire intensity to required water flow and number of drones. The key variable is the *critical water flow rate* (CF), defined as the rate of water application necessary to extinguish a certain number of meters of the active fire front. CF mainly depends on

the fireline intensity, which can be expressed as a function of flame length or a function of fire front spread. There are multiple ways to predict CF, the authors use Hansen's Fire Point theory and energy balance equations [43]:

$$CF = \frac{1}{\eta_{water} L_{v,water}} \left[(\phi \Delta H_c - L_v) \dot{m}_{cr} + \dot{q}_E - \dot{q}_L \right],$$

where η_{water} is the efficiency of water application, ϕ is the heat release fraction transferred back to the fuel surface by convection and radiation, ΔH_c is the effective heat of combustion, L_v is the heat of fuel gasification, $\dot{m}_{cr}$ is the mass burning rate of the fuel and $\dot{q}_E$, $\dot{q}_L$ represent heat gains and losses.

Given drone payload L_d, round-trip time Δt, and number of drones n_d managed by a single platform, the system flow rate (DF) is:

$$DF = \frac{L_d n_d}{\Delta t}.$$

The length of fireline that can be suppressed by one platform is then

$$m_f = \frac{DF}{CF}.$$

For instance, a platform with 120 drones carrying 20 L each and completing a 6-minute cycle yields roughly 400 L/min water flow, sufficient to extinguish about 70 m of low-intensity fire front under moderate wind.

Fire Propagation Model

To simulate the dynamic interaction between the fire front and the drone swarm's suppression action, Ausonio et al. use a CA model that simulates wildfire propagation over a two-dimensional grid. The landscape is discretized into square cells of side length $l = 2\,\mathrm{m}$, each representing a homogeneous patch of vegetation. Every cell (i, j) can take one of five discrete states: *empty*, *unburned*, *burning*, *burnt*, and *extinguished*. An *unburned* cell may ignite if at least one of its cell in the Moore neighborhood was *burning* in the previous step, this neighborhood can increase in wind direction for up to two layers. The probability of ignition p_{burn} is given by a multiplicative model that combines several physical and environmental factors:

$$p_{\text{burn}} = p_0(1 + p_{\text{veg}})(1 + p_{\text{den}})p_w p_s p_m,$$

where p_0 is the base ignition probability under no-wind flat conditions, p_{veg} and p_{den} are vegetation type and density and p_w, p_s, and p_m are the influence of wind, slope, and moisture. Once a cell ignites, it transitions to the *burned* state in the next timestep, and cannot reignite. Cells classified as *extinguished* represent locations under continuous water application, where CF is satisfied by the drone swarm. These cells block further spread, effectively modeling a containment or firebreak created by UAV intervention.

Results

The study demonstrates that a coordinated swarm of UAVs can effectively contain or extinguish small to medium wildfires under moderate environmental conditions with realistic parameters at the drone swarm (payload L_d = 20–30 L, cycle time Δt = 6 min, n_d = 80–120 drones per platform). The CA simulations confirm that this level of sustained discharge can completely extinguish fires under low wind and significantly slow or contain the spread at higher wind speeds. Multiple platforms operating simultaneously were shown to enhance coverage and enable flank attacks or the creation of artificial firebreaks. The results validate the feasibility of continuous aerial suppression using drone swarms, particularly as a complementary system to traditional aerial and ground firefighting methods.

3.2 Advancing Forest Fire Prevention: Deep Reinforcement Learning for Effective Firebreak Placement

Murray et al. introduce a RL framework that learns efficient spatial configurations of firebreaks using the *Cell2Fire* simulator, a CA based wildfire model. Firebreaks are non-flammable zones that interrupt fuel continuity and are a crucial preventive tool for mitigating wildfire spread. The study explores three value-based algorithms Deep Q-Network (DQN), Double DQN, and Dueling Double DQN, enhanced by *learning from demonstrations* and pre-training on heuristic solutions. The research represents the first systematic application of RL to the *Firebreak Placement Problem* (FPP) and demonstrates that RL agents can surpass existing heuristic baselines in both accuracy and adaptability.

Problem Definition

The FPP is formulated on a discretized forest grid, formally N, where each cell $i \in N$ may either contain fuel or be converted into a firebreak, which can be expressed by a decision variable $x_i \in 0, 1$. A placed firebreak removes all fuel from the cell and renders it unburnable. Since the placement of firebreaks is an expensive operation, not every cell can be converted into a firebreak and instead the total number of firebreaks is expressed as a percentage α of N (set to 5% in the study). The objective, minimizing the total burning cells in a wildfire f, can then be expressed as:

$$\min_x \ \mathbb{E}_f[L(x, f)]$$

where $L(x, f)$ represents the number of burned cells under fire scenario f *after* firebreaks x have been placed. $L(x, f)$ is obtained using the simulation environment, rather than computing it analytically, which can be very hard.

Reinforcement Learning Framework

The problem is cast as a MDP where each state s_t represents the current fuel configuration of the map, and each action a_t selects a cell to convert into a firebreak(only if it was not chosen in a previous step). Episodes continue until the budget of $\alpha|N|$ firebreaks is exhausted. After the episode ends, the *Cell2Fire* simulates the fire with the chosen firebreaks.

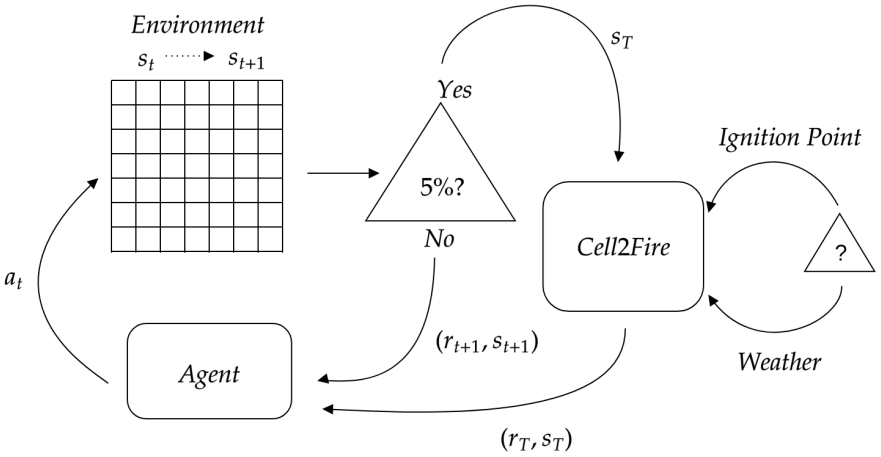


Figure 3.1: Agent-Environment-Simulation Interaction

The reward function is sparse:

$$r(s_t, a_t) = \begin{cases} 0, & t < T-1, \\ k \cdot n_b(s_t), & t = T-1, \end{cases}$$

where $n_b(s_T)$ is the number of burned cells after simulation and k a penalization factor.

Convolutional Neural Network (CNN)s approximate the action-value function $q_\theta(s, a)$, enabling spatial reasoning over the forest grid. Two network depths—*small-net* and *big-net*—were tested, alongside two transfer-learning backbones (*MobileNetV3* and *EfficientNet*). To accelerate training and guide exploration, the authors incorporated expert demonstrations based on the *Downstream Protection Value* (DPV) heuristic, a graph-based method estimating each cell's contribution to protecting downstream areas from fire propagation. Demonstrations populate an experience replay buffer and are reinforced through training.

Results

Two test maps from Alberta, Canada, Sub20 (20×20 grid) and Sub40 (40×40 grid), were used to evaluate the model. Fire ignition points and weather conditions were randomized within realistic constraints, with fuel maps derived from real forest data.

Each RL configuration was compared against two baselines, the DPV heuristic and a random firebreak allocation. Model performance was measured by the proportion of burned cells after multiple simulated wildfire events. Across both landscapes, all DRL variants achieved convergence and consistently outperformed the DPV and random

baselines. For the Sub20 forest, untreated terrain burned 18% on average. After intervention, DRL methods reduced this to 11.31–12.86%, outperforming the DPV heuristic (12.9%) and random allocation (16.1%). For Sub40, the burned area decreased from 31% (untreated) to 21.55–21.78%, again surpassing the baseline (23.25%) and random allocation (28.36%). No significant improvement was observed from deeper or transfer-learned architectures, suggesting the simpler CNNs sufficiently captured the relevant spatial dynamics. Generated attention maps revealed that the agent strategically selected clusters of adjacent firebreaks, enclosing critical areas and demonstrating implicit understanding of spatial fire dynamics.

3.3 Distributed Deep Reinforcement Learning for Fighting Forest Fires with a Network of Aerial Robots

Haskar et al. [44] present a decentralized control strategy in which teams of UAVs autonomously contain forest fires using RL. The authors model a forest as a two-dimensional lattice of trees, each in one of three states: healthy, on fire, or burnt. Fire spreads stochastically between neighboring trees with a known probability, and can be slowed or stopped by dropping fire retardant. UAVs move on the same grid, act within motion and communication limits, and can only extinguish fires directly beneath them. Each tree's evolution is a MDP, but because the forest contains many trees, the combined state space grows exponentially with forest size. Even if only realistically reachable states are considered, the configuration space remains astronomically large. Including limited sensing and partial observability means the full problem is closer to a Partially Observable Markov Decision Process (POMDP), whose exact solution is provably intractable.

Agent

UAVs fly at a constant altitude over the grid that represents the forest fire. Their movement is restricted to a discrete action space of nine options: staying in place or moving to any cell in the Moore neighborhood. The agent's internal state is mainly local. It includes a one-time memory flag that switches when the UAV first reaches a burning or burnt tree, a $h \times w$ grid image of the surrounding cells, a vector pointing toward the nearest other agent, and a rotation vector toward the initially reported fire location. Each UAV carries a limited amount of fire retardant, typically enough to extinguish about ten burning trees. When empty, the agent immediately returns to a base station on the forest edge, is refilled, and redeployed in the next control step. Empty UAVs are assumed to fly faster to minimize downtime.

Baseline heuristic

As a first step, the authors design a hand-tuned heuristic based on task decomposition. Each UAV initially moves toward the known ignition point and, once it detects nearby

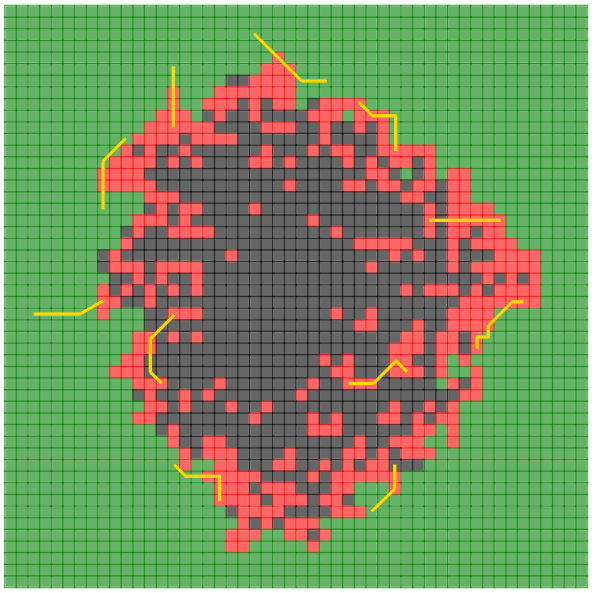


Figure 3.2: Example forest fire of Haskar et al. simulation [44].Color represent tree states: green(healthy), red(burning), black(burnt), yellow paths show UAV trajectories.

fire, switches to a "patrol mode", circling clockwise around the fire to drop retardant on burning boundary trees. The heuristic incorporates collision avoidance and simple cooperative behavior. Although effective, it still depends on careful manual design and cannot easily adapt to different forest sizes or numbers of agents.

Deep reinforcement learning approach

To go beyond the heuristic, the authors propose a distributed Multi-Agent Deep Q-Network (MADQN). Training is centralized with shared experience replay, but after training every agent executes the same neural policy independently, allowing unlimited scaling without retraining. The network has three fully connected layers and outputs a value for each possible move. Exploration during training is guided partly by the heuristic to stabilize and accelerate learning, the heuristic is also used to populate the experience replay buffer before training.

Experiments and results

The MADQN policy was trained in simulation on a 50×50 forest with ten agents and then tested under varied conditions: different fire sizes, agent numbers, limited retardant capacity, and faster-spreading fires. Performance was measured by the proportion of healthy trees remaining at the end of a fire. In extensive Monte-Carlo simulations (1,000 episodes per setting), MADQN consistently preserved more trees than the heuristic, particularly as the number of UAVs or fire severity increased. Notably, the trained policy generalized well beyond the conditions it had seen in training. Hardware trials

in the Georgia Tech Robotarium confirmed that the learned decentralized strategy can be implemented on real mobile robots.

3.4 UAV-based Firefighting by Multi-agent Reinforcement Learning

The paper by Shami Tanha et al. [45] proposes a cooperative fire-extinguishing strategy UAVs trained through a MARL framework. The authors design two discrete-time Markov fire-spreading models to simulate realistic fire dynamics and apply a *centralized training, decentralized execution* actor–critic algorithm based on Multi-Agent Deep Deterministic Policy Gradient (MADDPG). Their goal is to enable a swarm of UAVs to monitor, coordinate, and extinguish fires while learning optimal cooperative policies.

Fire Propagation Models

The authors model the forest as a two-dimensional discrete grid in which each cell contains a specific amount of fuel and interacts stochastically with its neighbors. All cells share identical structural properties, but differ in their current state and fuel quantity. For neighborhood definition, the Moore neighborhood is used. Fire propagation is modeled such that it follows the Markov property. Two probabilistic models are introduced to describe fire spread, both sharing common parameters such as the cell states shown in Figure 3.3, as well as the following quantities:

- $\alpha \in [0, 1]$: Probability of fire transfer from a red cell to a neighboring green cell;
- $\beta \in [0, 1]$: Average burning time of a cell;
- $\Delta\beta$, with $\Delta\beta < \beta$: Effect of applied fire retardant.

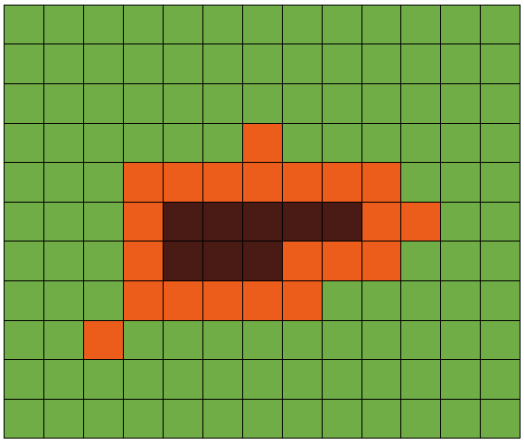


Figure 3.3: Example of the forest fire simulation by Shami Tanha et al. [45]. Colors represent cell states: green (healthy), red (burning), dark red (burned). What is showed as dark red here, is called gray in the paper.

Both variants also rely on the following probability for a cell *not* to catch fire:

$$p = (1 - \alpha)^{N_f},$$

where N_f denotes the number of burning neighboring cells. A red cell transitions to gray according to the burn-out rate β, which can be reduced by $\Delta\beta$ when a UAV releases fire retardant. Typical parameter values used in the experiments are $\alpha = 0.7$, $\beta = 0.9$, and $\Delta\beta = 0.54$ on a 50×50 grid.

The transition probabilities for both models are summarized in the following tables.

Table 3.1: Transition probabilities for the three-state fire model.

Current \ Next	Green	Red	Gray
Green	p	$1 - p$	0
Red	$\alpha\,\Delta\beta$	$\beta - \alpha\,\Delta\beta$	$1 - \beta$
Gray	0	0	1

Table 3.2: Transition probabilities for the four-state fire model.

Current \ Next	Green	Red	Gray	Blue
Green	p	$1 - p$	0	0
Red	0	$\beta - \alpha\,\Delta\beta$	$1 - \beta$	$\alpha\,\Delta\beta$
Gray	0	0	1	0
Blue	0	0	0	1

The second model introduces an additional *blue* state, which differentiates it from the three-state model. While in the first model cells that were previously extinguished may reignite, in the four-state model cells that transition to the blue state remain permanently extinguished and cannot catch fire again

Agents

Each agent models an UAV and their movement is restricted to a discrete action space, they can move into the eight cardinal directions or stay still. They carry a limited amount of fire retardent and can *sense* their Moore neighborhood, aswell as the cell below them. In case of hovering above a lit cell, the agent releases fire retardent to extinguish it. They are able to observe their own orientation, the nearest other agents coordinates and the states of their percieved surroundings.

Learning Framework

The authors employ the MADDPG algorithm [46], an actor–critic method extended for multiple agents. Each agent i maintains:

- an **actor network** $\mu_{\theta_i}(o_i)$ producing deterministic actions from its local observation;
- a **critic network** $Q_{\mu_i}(x, a_1, \ldots, a_N)$ evaluating the joint action $a_1, \ldots, a_N$ given global state x.

During training, critics have centralized access to all agents' observations and actions, but actors rely only on local data during execution. Likewise rewards are computed individually and only depend on local interactions:

- $+200$ if the agent is positioned over a burning cell (direct extinguishing);
- $+50$ if after movement the new position or observation still contains fire;
- -25 otherwise (idle or ineffective movement);
- small penalty (-0.5) for proximity to other agents closer than a collision distance to discourage clustering.

Each agent stores their transitions in a shared replay buffer D and updates both networks using mini-batches. Target networks and soft updates stabilize training. The *centralized training, decentralized execution* setup ensures cooperation without requiring communication at runtime.

Experiments and results

Experiments were conducted on a 50×50 grid with same fuel values for all cells. Fire was initialized in two configurations (4×4 and 10×10 ignition zones) and compared against heuristic, individual DQN, and hybrid heuristic–DQN baselines. Performance was measured as the fraction of remaining unburned (green or blue) cells averaged over episodes:

$$m_{\text{model1}} = \frac{1}{E}\sum_e \frac{\text{green}_e}{\text{total}}, \quad m_{\text{model2}} = \frac{1}{E}\sum_e \frac{\text{green}_e + \text{blue}_e}{\text{total}}.$$

The proposed MARL system achieved markedly higher preservation ratios and faster reward convergence. For the three-state model, the proposed method outperformed the next best method (hybrid heuristic-DQN) by more than a factor of two, the four-state model was not as effective and was outperformed by the other methods.

4 Methodology

4.1 Simulation Environment - ROSHAN

ROSHAN is a simulation framework designed for modeling vegetation fire dynamics and training RL agents in wildfire scenarios. The simulation builds upon the principles of CA to represent the fire spread process and provides a graphical interface that enables real-time observation and interaction. The user interface offers a dynamic visualization of the evolving fire front while allowing control over a range of simulation parameters.

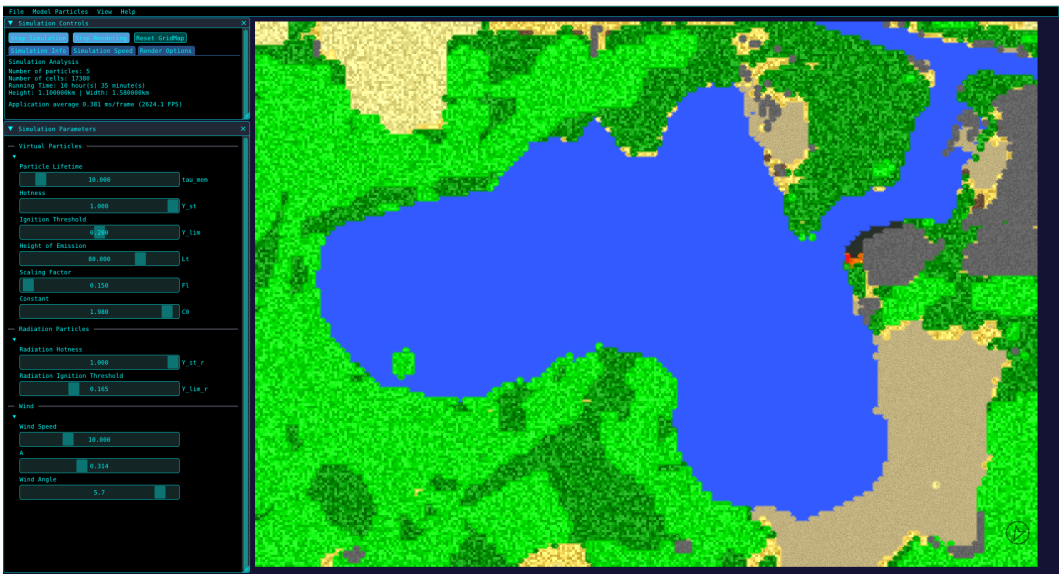


Figure 4.1: Overview of the ROSHAN simulation. The loaded map shows an artificial lake in Haltern am See at "Die Haard". On the right side of the lake, a visible burn scar marks the area affected by the fire. The fire eventually extinguished naturally, unable to cross the water barrier or ignite the surrounding low-flammability cells.

At its core, the fire spread model in ROSHAN is driven by the generation and propagation of virtual particles (see Figure 4.2). These particles act as carriers of heat and ignition potential, capable of setting adjacent cells ablaze based on local probabilities and environmental conditions(see Subsection 4.1.2 for exact model dynamics). Each particle can be visually represented within the interface, with its size and color indicating its relative intensity and lifespan, bright yellow particles represent newly ignited, high-energy fires, while smaller red particles correspond to those nearing extinction.

ROSHAN also has a RL module. Within this module, autonomous agents, modeled as UAVs, are trained to detect and extinguish fires in their environment. By learning to navigate and act under dynamically changing fire conditions, these agents demonstrate

Figure 4.2: Close-up of an active fire region. Each dot represents a particle whose size and color indicate its intensity. Smaller, redder particles correspond to older, cooling embers that are close to extinguishing.

adaptive behavior in complex, stochastic environments. This integration of real-time simulation and HRL provides a platform for developing, testing, and evaluating intelligent fire-suppression strategies.

4.1.1 Previous Work and Enhancements

This work builds upon an earlier implementation developed during the *Wissenschaftliche Vertiefung*. The underlying CA-based fire model was preserved in structure and functionality but underwent several low-level optimizations and refactoring steps, significantly improving computational performance and stability. In contrast, the RL framework was rebuilt almost entirely. The new design introduces a hierarchical multi-agent architecture that allows the concurrent operation of distinct agent types, each assigned to different layers of decision-making. The framework now supports multiple algorithms within a unified training interface, including PPO, IQL, and TD3, each with its own dedicated configuration structure and training pipeline.

Additional advancements include a fully restructured observation space that better captures spatial and temporal dependencies, extensive configurability through a configuration file, and improved evaluation and logging infrastructure. The neural network architectures were redesigned to better accommodate spatial and temporal features. Although the fire model and underlying dataset were already described in detail in the preceding *Wissenschaftliche Vertiefung*, they are briefly revisited in Sections 4.1.2 and 4.1.3 for completeness and contextual consistency.

4.1.2 FireSPIN Fire Spread Model

The *FireSPIN* (Fire Stochastic Particle Integrator) model, introduced by Mastorakos et al. [16, 47], provides a physically inspired fire spread model for simulating wildfire dynamics. It combines a CA representation of the landscape with a stochastic Lagrangian particle system that models the convection and radiation processes that drives fire propagation.

FireSPIN discretizes the environment into a grid of cells, each representing a patch of terrain characterized by properties such as vegetation type, the presence of structures, or surface materials like asphalt or water. Every cell has a dynamic state describing its current fire condition, which updates at discrete intervals throughout the simulation according to local interactions and environmental influences. A distinctive aspect of *FireSPIN* is its two classes of virtual fire particles: *convection particles* and *radiation particles*. *Convection particles* emulate the transport of hot gases and embers via turbulent air motion, while *radiation particles* capture radiative heat transfer. Together, they model how both convective and radiant mechanisms contribute to ignition and flame spread across the landscape.

Convection Particle Dynamics. The evolution of convective particles is governed by a set of stochastic differential equations:

$$\frac{dY_{st,p}}{dt} = -\frac{Y_{st,p}}{\tau_{mem}}, \tag{4.1}$$

$$dX_{i,p} = F_l U_{i,p}\, dt, \quad i \in \{1,2\}, \tag{4.2}$$

$$dU_{i,p} = -\frac{(2+3C_0)}{4}\frac{u'}{L_t}(U_{i,p} - U_{w,i})\, dt + (C_0 \epsilon\, dt)^{1/2}\mathcal{N}_i. \tag{4.3}$$

Equation 4.1 models the decay of a particles thermal intensity $Y_{st,p}$ with timescale parameter τ_{mem}. Particles with $Y_{st,p} > Y_{lim}$ can ignite new cells, where Y_{lim} is a configurable threshold. Equations 4.2–4.3 describe stochastic motion in a turbulent wind field: $U_{i,p}$ is the velocity component of the particle, $U_{w,i}$ is the wind velocity component, u' the turbulent velocity fluctuation which can be derived from $U_{w,i}$, and L_t the integral length scale parameter. Turbulent kinetic energy is expressed as $\epsilon = \frac{(u')^3}{L_t}$. The coefficients $F_l \approx 0.1$ and $C_0 \approx 2$ serve as model constants, and $\mathcal{N}_i$ represents Gaussian noise. Parameters u' and L_t determine the random walk's intensity, ensuring that particle dispersion reflects wind-driven turbulence.

Radiation Particle Dynamics. Radiative transfer is approximated through a simplified stochastic ray-tracing method:

$$dr_p = \overline{S_{f,0}}\, dt + \sigma_{S_{f,0}}(dt)^{1/2}\mathcal{N}_r, \tag{4.4}$$

$$d\theta_p = 2\pi\mathcal{N}_\theta. \tag{4.5}$$

Here, dr_p and $d\theta_p$ represent random changes in the radial and angular coordinates of a radiation particle. $\mathcal{N}r$ and $\mathcal{N}\theta$ are random variables derived from a normal distribution with unit variance. The flame propagation spread rate $\overline{S_{f,0}}$ can either be a fixed value, e.g. 0.1 m/s or be sampled from a normal distribution parameterized per vegetation type. The decay time of radiation particles follows $\tau_{mem} = \frac{L_r}{S_{f,0}}$, where L_r is a characteristic radiation length scale. A particle's position at time t is obtained from

$$\begin{aligned} X_{1,p} &= X_{1,mc} + r_p \cos(\theta_p), \\ X_{2,p} &= X_{2,mc} + r_p \sin(\theta_p), \end{aligned} \tag{4.6}$$

where $(X_{1,mc}, X_{2,mc})$ denotes the coordinates of the emitting cell.

Combustion and Ignition Model. Each cell class is associated with ignition and burn times τ_{ign} and τ_{burn}, defining how quickly ignition occurs and how long a fire persists. When a particle visits a cell, an ignition timer is started. If exposure exceeds τ_{ign}, the cell transitions to a burning state for τ_{burn} timesteps, during which new particles are emitted according to the local fuel properties.

4.1.3 CORINE Land Cover and the CLC+ Backbone

The CORINE Land Cover (CLC) database, developed under the Copernicus Land Monitoring Service [48], has served for more than three decades as a cornerstone for harmonized land use and land cover information across Europe. Its long temporal coverage and continent-wide consistency make it an essential foundation for environmental assessment, spatial planning, and climate analysis. Nevertheless, the original CLC product suffers from a relatively coarse spatial resolution and limited thematic granularity. To address these constraints, the *CLC+ Backbone* was introduced as an advanced successor, substantially enhancing spatial detail and class diversity.

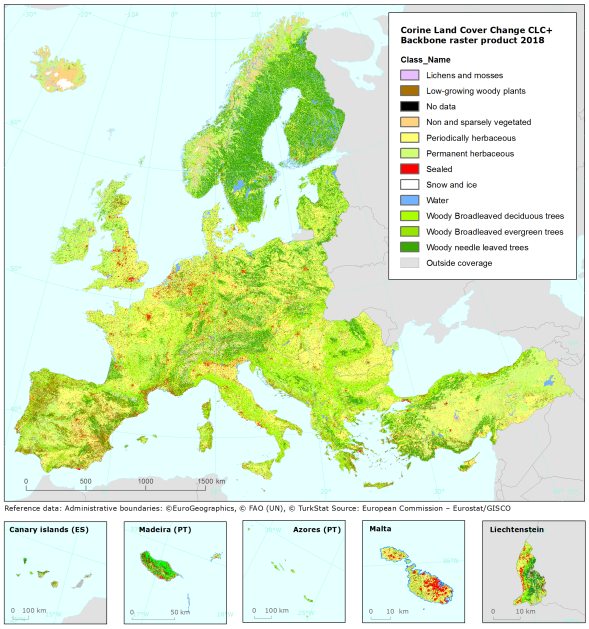


Figure 4.3: Coverage of the CLC+ Backbone raster dataset.

Improved Spatial Resolution

The CLC+ Backbone [49] provides a raster product with a spatial resolution of 10 m, representing a major improvement over the traditional CORINE layers. This fine-grained resolution is critical for wildfire modeling, as it allows the ROSHAN simulation to resolve vegetation structures and land-use boundaries with much greater precision.

Extended Land Cover Classification

Another enhancement of the CLC+ Backbone is its expanded typology of land cover. The raster product distinguishes eleven primary land cover classes, covering a broad spectrum of natural and anthropogenic surface types. For ROSHAN, this classification is used to model heterogeneous vegetation patterns and their corresponding fuel properties, allowing the model to simulate how fire behavior changes across diverse landscapes. The classification includes different forms of woody and herbaceous vegetation, bare areas, and water or ice surfaces, each associated with specific ignition and burn characteristics.

Land Cover Classes Overview

The CLC+ Backbone raster distinguishes eleven core land cover categories, each describing a unique environmental surface type relevant to fire behavior modeling.

CLC+ Backbone Land Cover Classes:

1. **Sealed**: Impervious surfaces such as concrete and asphalt, typical of urban or industrial areas.
2. **Woody-Needle-Leaved Trees**: Forested regions dominated by coniferous tree species.
3. **Woody-Broadleaved Deciduous Trees**: Areas composed primarily of deciduous tree species that shed their leaves seasonally.
4. **Woody-Broadleaved Evergreen Trees**: Zones characterized by evergreen broadleaved vegetation retaining foliage year-round.
5. **Low-Growing Woody Vegetation**: Shrublands and bushy vegetation of limited height.
6. **Permanent Herbaceous**: Grasslands or similar vegetation that remain stable throughout the year.
7. **Periodically Herbaceous**: Areas with seasonal variation in herbaceous coverage.
8. **Lichens and Mosses**: Surfaces predominantly covered by cryptogamic vegetation such as moss or lichen.
9. **Non- or Sparsely-Vegetated**: Bare or semi-barren areas, including rocks, gravel, or desert-like surfaces.
10. **Water**: Lakes, rivers, and other bodies of standing or flowing water.
11. **Snow and Ice**: Permanently or seasonally glaciated regions.
12. **Outside Area**: Pixels outside the defined geographic extent of the CLC+ Backbone raster product.

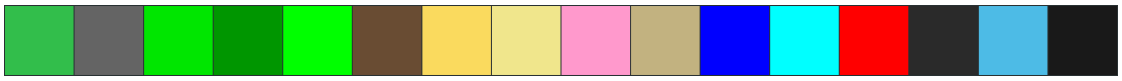


Figure 4.4: Cell state colors used in the ROSHAN simulation. From left to right: Generic Unburned (not in CLC+), Sealed, Woody-Needle-Leaved Trees, Woody—Broadleaved Deciduous Trees, Woody-Broadleaved Evergreen Trees, Low-Growing Woody Vegetation, Permanent Herbaceous, Periodically Herbaceous, Lichens and Mosses, Non- and Sparsely-Vegetated, Water, Snow and Ice, Generic Burning (not in the model), Generic Burned (not in CLC+), Generic Flooded (not in CLC+), Outside Area.

4.2 Reinforcement Learning Framework

4.2.1 Hierarchical Task Decomposition

Wildfire suppression represents a complex control problem defined by spatial scale, environmental dynamics, and temporal dependencies. In the simulated environment, multiple ignition points can emerge and evolve simultaneously across a heterogeneous grid map influenced by wind and fuel distribution. Autonomous aerial agents must detect active fires, navigate efficiently across the terrain, and coordinate to extinguish them before the fire front expands beyond control. These objectives are tightly coupled: movement decisions affect detection performance, exploration influences planning efficiency, and extinguishing success depends on both. Consequently, the suppression task is approached through a decomposition into complementary subtasks that govern *strategic planning*, *map exploration*, and *low-level flight control*.

A `PlannerAgent` functions as a high-level manager assigning mission goals, whereas the `ExploreAgent` and `FlyAgent` act as subordinate policies executing more primitive subtasks. This feudal organization enables independent learning at each level, local policies optimize their own rewards conditioned on immediate subgoals. This separation promotes temporal abstraction, and allows for modular reuse of learned policies.

PlannerAgent (High Level). The `PlannerAgent` operates on a global representation of the environment, maintaining information about all active *spotted* fires and drone positions. It decides which fires to extinguish next and controls multiple UAVs(`PlannerFlyAgent`). Its decision frequency is deliberately low to ensure stability in long-horizon reasoning and to prevent oscillatory goal selection.

ExploreAgent (Mid Level). The `ExploreAgent`'s purpose is to map unseen areas of the environment while gathering information for the `PlannerAgent`. One `ExploreAgent` controls multiple subordinates(`ExploreFlyAgent`). Although early versions experimented with reinforcement learning-based exploration policies, these approaches were eventually omitted due to limited convergence and computational overhead, due to the use of convolutional layers in its network architecture. In the final implementation, a deterministic goal selector(see Section 4.2.6) was introduced, assigning goals to exploration drones that execute systematic meander patterns across the map. Despite not

being learned through reinforcement learning, the `ExploreAgent`'s remain useful to the hierarchy, as they collect observations for the `PlannerAgent`.

FlyAgent (Low Level). At the lowest level, the `FlyAgent` performs continuous control of individual drones. It is the only *physical* agent in the framework. It receives detailed observations of the local environment, including nearby fires, obstacles, velocity, and goal direction. The agent outputs smooth velocity or acceleration commands depending on the configured control mode. Depending on the configuration setting and hierarchy level that incorporates the `FlyAgent` it also handles collision avoidance. This level operates at the highest temporal resolution, updating every simulation timestep.

4.2.2 Observations

In the context of the reinforcement learning formulation introduced in Section 2.2, an *observation* o_t corresponds to the information available to a policy at time step t, representing a partial view of the full environment state. Each hierarchy level constructs its own observation structure, reflecting its operational abstraction of the environment.

FlyAgent

The `FlyAgent` operates closest to the physical simulation and therefore relies on dense, spatially localized information. Each `FlyAgent` maintains a fixed *view range* defining its local perception. The view range r_{view} is a square window of size $(r_{\text{view}_x}+1)\times(r_{\text{view}_x}+1)$ cells centered on the agent's current position(see Figure 4.5). Within this window, the agent can access local information about the environment.

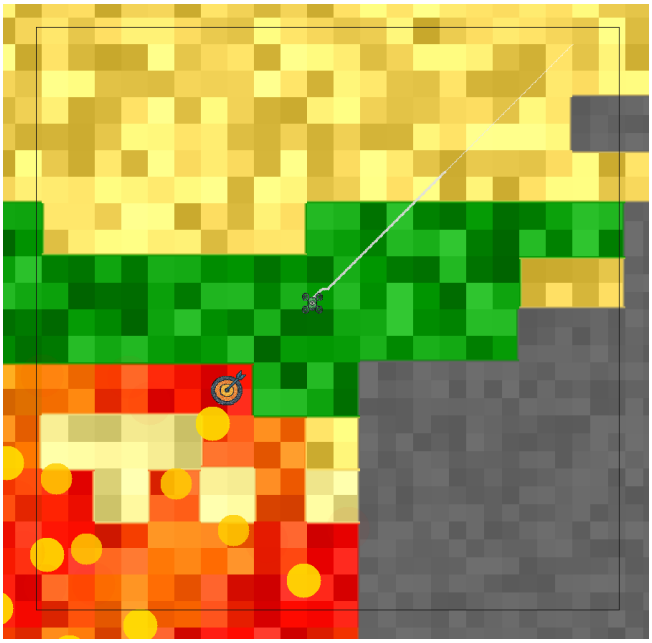


Figure 4.5: View range of a `FlyAgent` indicated by the small black lines.

A normalization parameter s_{norm} is set equal to the agent's view range r_{view}. This ties the internal scaling of all spatial features to the agent's sensory horizon. As a result, spatial relations such as goal distances, velocity vectors, and inter-agent separations are expressed relative to the agent's observable area. This *local normalization* ensures consistent feature magnitudes across maps of varying size and maintains scale invariance between agents with different sensor ranges. The state space of the `FlyAgent` then is:

- **ID:** A unique identifier within the agent's hierarchy subtype.
- **Velocity vector:** Per-axis velocity (v_x, v_y) scaled by the agent's maximum speed and the local normalization parameter:

  $$\tilde{\mathbf{v}} = \left(\frac{v_x}{v_x^{\max} s_{\text{norm}}},\ \frac{v_y}{v_y^{\max} s_{\text{norm}}} \right).$$

  The resulting vector is clipped to $[-1, 1]$. This represents the agent's motion in locally normalized coordinates.
- **Relative goal vector:** Goal displacement from the current grid position,

  $$\Delta\mathbf{p} = \left(\frac{g_x - p_x}{s_{\text{norm}}},\ \frac{g_y - p_y}{s_{\text{norm}}} \right),$$

  encoding both the direction and relative distance of the goal within the agent's perceptual range, it is also clipped to $[-1, 1]$.
- **Distance-to-goal magnitude:** Euclidean norm of the normalized goal displacement,

  $$d = \min\left(\sqrt{\left(\frac{g_x - p_x}{s_{\text{norm}}} \right)^2 + \left(\frac{g_y - p_y}{s_{\text{norm}}} \right)^2},\ 1.0 \right),$$

  providing a scalar measure of goal proximity complementary to the directional encoding.
- **Signed heading alignment to goal:** Represents the angular relationship between the agent's movement direction and the direction toward its goal. The goal vector is defined as $\mathbf{g} = (g_x - p_x,\ g_y - p_y)$ and the velocity vector as $\mathbf{v} = (v_x, v_y)$. After normalization to unit length, the relative orientation is expressed as

  $$\cos\theta = \hat{\mathbf{v}} \cdot \hat{\mathbf{g}}, \qquad \sin\theta = \hat{\mathbf{v}} \cdot (-g_y,\ g_x),$$

  both values clipped to $[-1, 1]$. This two-dimensional representation h_θ allows the policy to infer whether the agent is turning toward or away from its goal. This serves as a *global indicator* of the agent's overall movement intent, even when no local map information is available.
- **Speed magnitude:** Total velocity magnitude normalized to the agent's maximum speed:

  $$s = \sqrt{\left(\frac{v_x}{v_x^{\max}} \right)^2 + \left(\frac{v_y}{v_y^{\max}} \right)^2}.$$

  Like the signed heading alignment, the speed magnitude acts as a global variable that remains informative in the absence of local spatial inputs.

- **Distances and relative velocities to other agents:** For each neighboring agent within the local view range, a four-dimensional feature vector $\tilde{a}$ is recorded consisting of the relative position $(\Delta x, \Delta y)$ and the neighbor's normalized velocity (v_x, v_y). All components are scaled by the agent's s_{norm}, ensuring that inter-agent relations are expressed in locally normalized units. Agents outside the perceptual window contribute zero vectors, preserving a fixed feature dimension.

It should be noted, that not all these features are enabled at all times and their enabling is dependent on the selected setting, which influences the network architecture(see Section 4.2.5).

PlannerAgent

The `PlannerAgent` constructs a higher-level observation by aggregating global information across all subordinate agents and active fire locations.

- **Team state:** Includes the positions($\tilde{pos}$), current goal positions($\tilde{goal}$) and IDs of all subordinate `FlyAgents`.
- **Fire state:** Represents the spatial distribution of active fires as a set of (x, y) coordinates as $\tilde{fire}$. During centralized training, these positions are obtained directly from the simulation grid, providing the `PlannerAgent` with complete global awareness. In contrast, during decentralized execution, the fire locations are derived from the accumulated observations of the `ExploreAgent` and its subordinates, together with the partial detections reported by the `PlannerAgent`'s own `FlyAgents`. This transition from full to inferred information reflects the shift from centralized training to decentralized execution, enabling the agent to reason about the global fire landscape under realistic, partially observed conditions.

Unused and Experimental State Representations

During development, several alternative state formulations were implemented and evaluated but were ultimately not integrated into the final experiments. These approaches primarily explored *image-based feature representations* to provide convolutional encoders with spatial context at both local and global scales. The main experimental representations included the following:

- **Local image features:** Each `FlyAgent` could generate two local raster views within its perception radius: a *terrain view* encoding cell types and a *fire view* marking active fires. Both maps covered a $(r_{\text{view}} + 1) \times (r_{\text{view}} + 1)$ area centered on the agent and were intended as spatial inputs to convolutional layers.
- **Global map features:** Agents maintained several global rasters updated over time:
  - **TotalDroneView:** A binary occupancy grid with the agents current position, as view range coverage over the whole map.
  - **StepExploreMap:** A temporary per-agent map that tracked newly observed cells and visitation counts until the agent reached its goal.

  - **ExploreMapTotal:** A cumulative exploration map formed by merging the individual StepExploreMaps of all agents.
  - **FireMap:** An internal record of detected fires, updated whenever a fire was observed within an agent's local fire view.

- **Interpolation and normalization:** To ensure compatibility with convolutional networks, all image-like maps were interpolated to a common resolution using bilinear interpolation.
- **Derived statistical features:** Several scalar quantities were extracted from the above maps, such as the number of discovered fires, the percentage of explored cells relative to the total map area, and the ratio of newly explored to revisited cells.
- **Alternative coordinate encodings:** Additional normalization schemes were investigated, including expressing positions relative to the map center or in the coordinate frame of the exploration map itself.

Temporal Stacking

All agents employ a temporal structure based on *frame stacking*. Instead of relying solely on the current observation, each state representation consists of a short sequence of consecutive observations $\{o_t, \ldots, o_{t-k}\}$, where k is the stacking parameter.

4.2.3 Actions

There are two different action spaces in the hierarchical control structure. The `FlyAgent` performs primitive, continuous actions that directly affect its motion in the environment, while the higher-level `ExploreAgent` and `PlannerAgent` select spatial goal positions that guide subordinate agents toward specific regions or objectives on the map.

FlyAgent. The `FlyAgent` outputs a two-dimensional continuous action vector

$$[a_x, a_y] \in (-1, 1),$$

which governs its motion in the two-dimensional plane of the simulation. Two control modes are supported:

1. **Acceleration Control:** The network outputs are interpreted as normalized acceleration commands. Each component is scaled by the agent's maximum acceleration $a_{\max}$ (treated as a fraction of the maximum attainable speed $v_{\max}$, typically $a_{\max} \approx 0.5\, v_{\max}\, \mathrm{s}^{-1}$) and integrated over the simulation timestep Δt:

$$v_{t+1} = v_t + [a_x, a_y] \cdot a_{\max} \cdot \Delta t.$$

2. **Direct Velocity Control:** The network outputs are interpreted directly as normalized velocity components and scaled to the agent's maximum speed:

$$v_{t+1} = [a_x, a_y] \cdot v_{\max}.$$

The resulting velocity components are clamped to

$$v_i \in [-v_{\max}, v_{\max}],$$

ensuring that the agent's motion remains within its physical constraints. Optionally, the raw network outputs can be discretized into 41 uniform bins prior to scaling:

$$a_d = \text{round}\left(\frac{a}{0.05}\right) \times 0.05,$$

which effectively smooths small fluctuations in the network output and reduces jitter during continuous control.

ExploreAgent and PlannerAgent (High-Level Control). The `ExploreAgent` and `PlannerAgent` operate on a higher level of abstraction, producing goal assignments rather than direct motion commands. Their action is represented as a set of two-dimensional goal positions

$$\mathbf{g} = \{\, g_i = [x_i, y_i] \mid i = 1, \ldots, N_{\text{fly}} \,\},$$

where each g_i defines the target position assigned to a subordinate `FlyAgent`. Thus, a single high-level action corresponds to a coordinated set of spatial objectives for the entire swarm.

Actor Architecture

Each hierarchical agent is associated with an *Actor* architecture that maps its current observation to an appropriate action representation. While the overall structure follows a shared interface, the specific actor formulation depends on both the agent's hierarchical level and the reinforcement learning algorithm employed.

PlannerAgent (Categorical Actor). The `PlannerAgent` operates on a discrete spatial action space and therefore employs a categorical actor design. Given the current system state s_t, the actor network outputs a probability distribution over a finite set of candidate goal positions

$$G = \{g_1, g_2, \ldots, g_M\},$$

defined across the map for each subordinate `FlyAgent`. For each agent i, a categorical distribution is constructed as

$$a_i \sim \text{Categorical}(\pi_\theta(\cdot \mid s_t)),$$

and the corresponding spatial goal is selected from the candidate set:

$$g_i = G[a_i], \quad g_i \in \mathbb{R}^2.$$

This results in a collective goal vector

$$\mathbf{g} = \{\, g_i \mid i = 1, \ldots, N_{\text{fly}} \,\},$$

which defines a goal configuration for all controlled `FlyAgents`.

FlyAgent (Continuous Actor). The `FlyAgent` operates in a continuous action space and supports three algorithm-specific actor variants, IQL, PPO, and TD3, which mainly differ in how stochasticity and exploration are treated:

- **IQL (Stochastic Actor):** The actor outputs continuous mean actions $\mu_\theta(s_t)$, which are augmented by Gaussian exploration noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ during data collection:
$$a_t = \tanh(\mu_\theta(s_t) + \epsilon), \qquad \sigma \approx 0.05.$$
Actions are clamped to $(-1, 1)$, which is valid since the log-probabilities are later computed from the stored, noise-free actions.
- **PPO (Stochastic Actor):** The actor outputs both a mean and variance, $\mu_\theta(s_t)$ and $\sigma_\theta(s_t)$, which define a Gaussian or Tanh-squashed Gaussian distribution:
$$a_t \sim \mathcal{N}\Big(\mu_\theta(s_t), \sigma^2_\theta(s_t)\Big).$$
The corresponding log-probabilities $\log \pi_\theta(a_t \mid s_t)$ are used for policy optimization and advantage estimation. The Tanh-squashed variant ensures that actions remain within $(-1, 1)$, preserving numerical stability and consistent log-probability evaluation. In contrast, directly clipping Gaussian-sampled actions can distort the probability density near the boundaries, leading to invalid log-probabilities and unstable gradient updates. Alternatively, one could clip the actions only within the simulation environment rather than in the policy itself, however, this approach could cause excessive activation of boundary actions.
- **TD3 (Deterministic Actor):** The actor produces a deterministic action $a_t = \mu_\theta(s_t)$, with uncorrelated Gaussian noise added for exploration:
$$a_t = \text{clip}\Big(\mu_\theta(s_t) + \mathcal{N}(0, \sigma^2_{\text{explore}}), -1, 1\Big).$$

4.2.4 Rewards

The reward structure defines the optimization objectives for each agent hierarchy and serves as the principal learning signal during policy training (see Section 2.2.1). Since the agents operate at different levels of abstraction, their rewards are formulated with distinct scopes.

FlyAgent

The `FlyAgent` reward structure is defined in four configuration modes that reflect different training settings(see Section 4.2.7):

$$\begin{aligned}
R_{\text{simple}} &= R_{\text{goal}} + R_{\text{timeout}} + R_{\text{boundary}} + R_{\text{progress}}, \\
R_{\text{simple_collision}} &= R_{\text{goal}} + R_{\text{timeout}} + R_{\text{boundary}} + R_{\text{coll}} + R_{\text{progress}} + R_{\text{prox}}, \\
R_{\text{complex}} &= R_{\text{goal}} + R_{\text{timeout}} + R_{\text{boundary}} + R_{\text{ext}} + R_{\text{progress}}, \\
R_{\text{complex_collision}} &= R_{\text{goal}} + R_{\text{timeout}} + R_{\text{boundary}} + R_{\text{ext}} + R_{\text{coll}} + R_{\text{progress}} + R_{\text{prox}}.
\end{aligned}$$

The individual components are defined as follows.

$$R_{\text{goal}} = \mathbf{1}_{\text{objective reached}} \cdot \beta_{\text{goal}},$$

Goal completion reward. A terminal bonus applied when the global firefighting objective is achieved.

$$R_{\text{timeout}} = \mathbf{1}_{\text{timeout}} \cdot \beta_{\text{timeout}},$$

Timeout penalty. Applied when the episode terminates without reaching the goal due to exceeding the maximum number of environment steps.

$$R_{\text{boundary}} = \mathbf{1}_{\text{out of bounds}} \cdot \beta_{\text{boundary}},$$

Boundary violation penalty. Penalizes agents that leave the map area, discouraging trajectories that cross environment boundaries.

$$R_{\text{ext}} = \mathbf{1}_{\text{fire extinguished}} \cdot \beta_{\text{ext}},$$

Fire-extinguish reward. A positive bonus granted when the agent successfully extinguishes a burning cell.

$$R_{\text{coll}} = \mathbf{1}_{\text{collision}} \cdot \beta_{\text{coll}},$$

Collision penalty. Applied when the agent collides with another drone. This should discourages unsafe flight behavior during cooperative operation.

$$R_{\text{progress}} = \beta_{\text{progress}} \cdot (d_{t-1} - d_t),$$

Distance-improvement reward. A dense shaping term proportional to the reduction in goal distance between consecutive steps. Positive values reward forward progress, while regressions yield negative feedback.

$$R_{\text{prox}} = \beta_{\text{prox}} \sum_{i \in \mathcal{N}} \Big[\, |\Delta x_i| < \tau \text{ or } |\Delta y_i| < \tau \,\Big] \cdot f(|\Delta x_i|, |\Delta y_i|),$$

Proximity penalty. A continuous safety term that penalizes the agent for being within a small threshold τ of other drones. It is meant to further reduce collisions.

Each coefficient β_i represents a tunable reward weight.

PlannerAgent

The total planner reward R_{planner} is composed of additive components:

$$R_{\text{planner}} = R_{\text{goal}} + R_{\text{fast}} + R_{\text{timeout}} + R_{\text{burned}} + R_{\text{ground}} + R_{\text{same}} + R_{\text{ext}}.$$

with

$$R_{\text{goal}} = \mathbf{1}_{\text{objective reached}} \cdot \alpha_{\text{goal}},$$

Goal completion reward. A terminal bonus granted once the overall firefighting objective has been achieved. It serves as the primary positive signal driving successful coordination between all subordinate agents.

$$R_{\text{fast}} = \mathbf{1}_{\text{objective reached}} \cdot \alpha_{\text{fast}} \cdot \frac{\text{steps}_{\text{rem}}}{\text{steps}_{\text{total}}},$$

Time-efficiency bonus. Provides an additional shaping term for completing the mission quickly. The earlier the objective is achieved, the greater the reward contribution.

$$R_{\text{timeout}} = \mathbf{1}_{\text{timeout}} \cdot \alpha_{\text{timeout}},$$

Timeout penalty. Applied when the episode terminates without reaching the goal due to exceeding the maximum number of environment steps.

$$R_{\text{burned}} = \mathbf{1}_{\text{burned too much}} \cdot \alpha_{\text{burned}},$$

Over-burn penalty. Triggered if a large portion of the map burns beyond a threshold before mission success, discouraging delayed or inefficient planning.

$$R_{\text{ground}} = \mathbf{1}_{\text{goal=groundstation}} \cdot \alpha_{\text{ground}},$$

Ineffective-goal penalty. Applied when a subordinate `FlyAgent` is assigned to the ground-station position while fires are still active, representing an ineffective goal assignment.

$$R_{\text{same}} = \alpha_{\text{same}} \cdot \max(0,\, N_{\text{goals}} - N_{\text{unique}}),$$

Redundancy penalty. Reduces the reward when multiple `FlyAgents` pursue identical goal positions, encouraging distributed task allocation.

$$R_{\text{ext}} = \alpha_{\text{ext}} \cdot N_{\text{extinguished}}.$$

Fire-extinguish reward. Adds a positive contribution proportional to the total number of extinguished fires, reinforcing global progress toward the main objective.

Each coefficient α_i corresponds to a configurable weight in the planner's parameter set.

To stabilize training and reduce variance, reward signals in ROSHAN are normalized using a running mean and standard deviation estimator. It must be noted that rewards are only scaled this way and never shifted.

Intrinsic Reward

To encourage exploration beyond externally defined rewards, an intrinsic motivation signal based on Random Network Distillation (RND) [23] was implemented. The RND mechanism provides a novelty-driven intrinsic reward by measuring the prediction error between two networks that process the same observation input.

Architecture. The RNDModel consists of two networks with identical architectures, a fixed, randomly initialized *target network* and a trainable *predictor network*. Both share the same input encoding module. The target network's parameters remain frozen throughout training, serving as a static nonlinear projection of the observation space.

Intrinsic reward computation. For each observation o_t, the target network produces a latent feature representation $f_{\text{tgt}}(o_t)$, while the predictor network estimates $f_{\text{pred}}(o_t)$ by minimizing the mean-squared error between the two outputs:

$$L_{\text{RND}} = \left\| f_{\text{pred}}(o_t) - f_{\text{tgt}}(o_t) \right\|^2.$$

The intrinsic reward is derived directly from this prediction error,

$$r_t^{\text{int}} = \mathbb{E}\Big[\left(f_{\text{pred}}(o_t) - f_{\text{tgt}}(o_t) \right)^2 \Big],$$

which quantifies the novelty of the observation. States that are poorly predicted (i.e., rarely visited) yield higher intrinsic rewards, motivating the agent to explore them more frequently.

The RND mechanism was initially integrated into the early version of the `ExploreAgent` to promote active discovery of unexplored map regions.

4.2.5 Network Architecture

The neural network architecture of all agents in ROSHAN follows a modular design built from three primary components. The *Inputspace Encoders*, the *Actor–Critic modules*, and algorithm-specific network heads. This modularity ensures that network configurations can be easily exchanged between PPO, IQL, and TD3 while preserving a consistent input–output interface. All networks use ReLU activations and Xavier initialization.

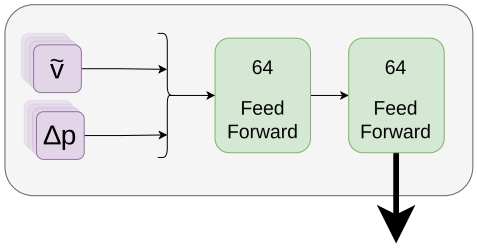


Figure 4.6: Simple Inputspace for `FlyAgent` without collision. Purple boxes describe the observation input described in Section 4.2.3, while **green** boxes are fully-connected layers.

Inputspace Encoders

Each agent receives spatial and contextual information from the simulation environment, which is first processed by an *Inputspace Encoder* to produce a compact latent representation. Three encoder variants exist, depending on the agent type and collision setting:

- **Simple Encoder (Figure 4.6):** Used by the `FlyAgent` when collisions are disabled. It encodes local perception such as position, velocity, and goal direction.
- **Collision Encoder (Figure 4.8):** Augments the simple encoder with a neighborhood encoding module that models spatial relations to nearby agents for collision avoidance.
- **Planner Encoder (Figure 4.7):** Integrates information from subordinate agents using a cross-attention mechanism to generate categorical logits for spatial goal selection.

Each encoder outputs a flattened latent vector that serves as the shared feature representation for both actor and critic networks.

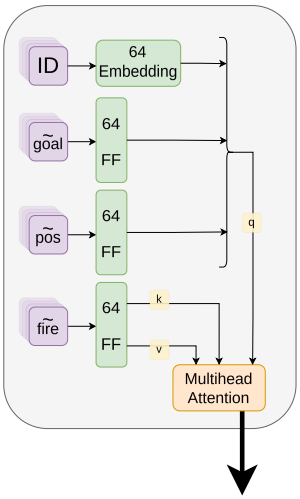


Figure 4.7: The encoder embeds each drone's position, goal position, and identifier into a shared latent space, while global fire targets are encoded separately. A multi-head cross-attention module relates drone embeddings (queries) to fire embeddings (keys and values), producing contextualized representations that indicate how strongly each drone should attend to each fire or goal location.

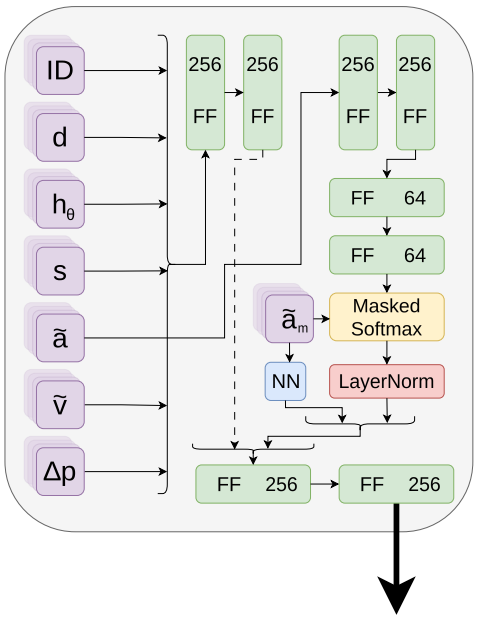


Figure 4.8: The encoder processes the agent's observation features (**purple**) described in Section 4.2.2. Information about nearby agents is aggregated using an attention-based pooling mechanism that handles variable neighbor counts through masking (**yellow**). The self representation and the aggregated neighbor embedding are concatenated and passed through a fusion network that produces a compact latent representation for each timestep. In the illustration, **green** boxes denote fully connected transformation layers, and the **blue** box marks the additional *NoNeighbor Flag*. The mask $\tilde{a_m}$ indicates the neighbor visibility mask applied during attention computation.

Actor–Critic Modules

The agents follow an *Actor–Critic* design (Figure 4.9) with either shared or independent encoders. For PPO and IQL, actor and critic networks may share the same encoder to reduce redundancy and improve feature consistency, while TD3 always has separate encoders for its dual critics to prevent correlated value estimates.

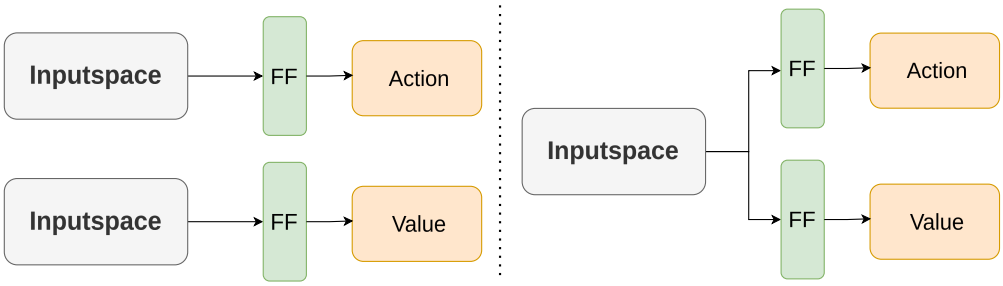


Figure 4.9: Example of two different Actor Critic Architectures. **Left:** Two separate encoding modules, one for the Actor and one for the Critic. **Right:** Actor and Critic share the same encoder module.

Actor Networks. The actor design depends on the algorithm and the agent hierarchy: the `PlannerAgent` uses a categorical head for discrete goal selection, while the `FlyAgent` employs continuous actors that output Gaussian or Tanh-squashed Gaussian distributions for PPO and IQL, or deterministic actions for TD3. Action sampling and stochasticity follow the formulations described in section 4.2.3.

Critic and Value Networks. The critics estimate the state–action value $Q(s, a)$ through two independent MLPs (Q_1, Q_2) with 256-unit hidden layers. For IQL and TD3, these critics are trained using clipped double-Q learning, while IQL additionally maintains a separate value function $V(s)$ with an identical architecture but without the action input.

4.2.6 Deterministic Goal Selector

To initialize exploration behavior without requiring a learned policy, a deterministic goal selection routine was implemented. It generates predefined waypoint patterns that guide multiple subordinate `FlyAgents` of an `ExploreAgent` across the entire map in a systematic coverage pattern. The algorithm divides the map into evenly sized vertical or horizontal stripes depending on its aspect ratio, assigning one stripe to each agent. Within each stripe, waypoints are generated in a meandering, back-and-forth pattern that ensures complete spatial coverage relative to the agents view range. This pattern can be interpreted as a deterministic *meandering flight path*, where each sweep direction alternates per row (or column) to minimize overlap and blind spots. Each agent receives its own sequence of waypoints, stored as a queue of (x, y) positions that are consumed cyclically as goal targets. Once a local goal is reached, the next waypoint in the sequence becomes the active objective. This produces coordinated exploration across the entire map.

4.2.7 Policy Configurations and Collision Handling

The `FlyAgent` can operate under two policy configurations, *simple* and *complex*, and can optionally include explicit collision handling between agents. These configurations affect both the termination conditions of an episode and the active reward components used during training.

Collision handling. When collision handling is enabled, physical contact between two agents of the same hierarchy type triggers a terminal condition, marking the episode as a failure. This is distinct from boundary exits, which represent collisions with the map limits and are handled separately. Enabling collisions affects not only the environment dynamics but also the observation space and network architecture, since additional proximity and collision features are integrated into the agent's network structure.

Simple policy. The simple policy was originally introduced during early development when the `FlyAgent` only needed to learn to navigate toward its assigned goal position. In this setting, an episode is considered successful as soon as the first agent reaches its target, emphasizing pure motion control and goal-seeking behavior without explicit interaction with other agents.

Complex policy. While the simple policy was sufficient for basic navigation training, it became inadequate once inter-agent collisions were introduced, as avoiding other agents and maintaining safe trajectories required richer observations. The complex policy extends the simple configuration by coupling navigation with cooperative task completion. An episode is only considered successful once all active fires on the map have been extinguished. Fire-Goal assignment is done with a simple heuristic.

Greedy Heuristic Fire–goal assignment follows a simple *greedy heuristic*. Whenever an agent reaches and extinguishes its target, it is reassigned to the nearest unassigned fire cell based on Euclidean distance. Each fire can only be allocated to a single agent at a time. During training, agents may be assigned to all fires on the map, whether discovered or not. During evaluation, however, it is possible that only fires observed by any agent(`FlyAgents` and `ExploreFlyAgents`) are eligible targets; if none are available, the agent returns to the ground station. Note that this assignment only is used in the `FlyAgent` and never in the `PlannerAgent`.

4.3 Implementation

This section outlines the software stack and configuration system of the ROSHAN framework. The complete source code is available at `https://github.com/RoblabWh/ROSHAN`. In total, the project consists of approximately 8200 lines of C++ code and 4500 lines of Python code.

4.3.1 Software Stack

This subsection describes the libraries used in the development of the ROSHAN simulation. Development and testing was conducted on Ubuntu 22.04.

C++ and SDL2. ROSHAN is primarily implemented in C++, chosen for its high performance and precise control over memory and computation, which are the key requirements for large-scale, compute-intensive simulations. The visualization of the cellular automaton, fire dynamics, and agent behaviors is handled through the Simple DirectMedia Layer 2 (SDL2) library. SDL2 is employed for its efficiency in managing graphics and multimedia resources, enabling rendering of the processes within the simulation.

Dear ImGui. For interactive user control and simulation monitoring, ROSHAN incorporates the Dear ImGui library. It provides an intuitive and responsive graphical user interface that allows users to adjust simulation parameters, visualize statistics, and interact dynamically with the system during runtime.

GDAL and OpenStreetMap. To embed realistic geographic environments, ROSHAN integrates the CLC database through the Geospatial Data Abstraction Library (GDAL). GDAL converts and handles geospatial data formats, ensuring accurate landscape representation. Additionally, OpenStreetMap (OSM) data is accessed over a Node.js backend server, allowing users to select specific real-world regions as input for their simulations.

Python and PyTorch. The reinforcement learning components of ROSHAN are implemented in Python, which serves as the primary interface for running and controlling the entire simulation. Through custom Python bindings, the C++ simulation core is seamlessly integrated into Python, allowing direct access to all simulation functions without the need for separate executables or interprocess communication. This design enables experiments, training loops, and evaluations to be executed entirely from within Python, while retaining the computational efficiency of the C++ backend. For the neural networks themselves, ROSHAN uses PyTorch, a flexible and widely adopted deep learning framework. PyTorch is used to design, train, and evaluate the neural networks that govern the behavior of all agents in the hierarchical setup.

4.3.2 Configuration File

The simulation parameters of ROSHAN are defined through a single configuration file, which specifies nearly all aspects of the environment, learning algorithms, and agent behavior. On loading, the configuration initializes the corresponding variables within the simulation, ensuring consistent experimental conditions across runs. Each configuration file is stored alongside its trained model, network configurations, log files and evaluation for full reproducibility. This section provides an overview of the most relevant configuration groups and their purposes.

General Settings

The framework can operate in either graphical or headless mode. While the graphical mode provides real-time visualization and interaction, the headless mode executes the simulation without rendering and is therefore substantially faster, making it the preferred choice for large-scale training and evaluation runs. Reproducibility of the environment dynamics is controlled with a seed. This seed only affects the simulation itself, as PyTorch-related operations are intentionally left unseeded to preserve stochastic diversity in learning. Checkpoint resumption is fully supported, restoring network architectures, their corresponding weights, optimizer and scheduler states, as well as TensorBoard metrics and internal log files. This allows seamless continuation of interrupted training sessions without loss of experimental context.

Automated sequential training runs can be enabled. Then, the system executes multiple consecutive training sessions, each running for a fixed number of training steps. After each training phase, an evaluation phase of length is performed.

When Optuna for automated hyperparameter optimization is used, the system performs a defined number of optimization trials, each performing an independent training run with a unique set of sampled hyperparameters. A number of start up trials specifies how many initial random trials are completed before switching to the optimization algorithm, while a number of warmup steps controls the number of training iterations required before evaluation begins within each trial. Optionally, underperforming trials can be terminated early through Optuna's pruning mechanism, which halts runs that fall below a specified performance of previous trials. The objective to be optimized can target either overall episodic reward or an objective value.

Algorithm Configuration

All reinforcement learning algorithms supported by the ROSHAN framework, including PPO, IQL, and TD3 (see Sections 2.3.1–2.3.3) are subject to parameterization.

Encoder sharing between actor and critic networks can be activated for the algorithms that support it. Stochastic policies can optionally use a *TanhNormal* distribution, which naturally bounds actions within the valid range of $[-1, 1]$. When disabled, a standard Gaussian distribution is used, and actions are manually clipped (see Section 4.2.3).

PPO. The PPO configuration defines all hyperparameters relevant for on-policy training. Core parameters include *learning rate*, *batch size*, and *trajectory horizon length* and the number of optimization *epochs* per update. Regularization and loss balancing are controlled by the entropy and value loss coefficient. Discounting and advantage estimation are handled with *gamma* and *_lambda* and policy update clipping is applied through *eps_clip*.

For more stable policy updates, the configuration supports early stopping based on the approximate Kullback–Leibler Divergence (KL Divergence) between the old and new policy. When it is active, an update is halted once the divergence exceeds a defined

threshold, preventing excessively large policy shifts. Two KL Divergence estimators are available which were suggested by John Schulman[50]:

$$k1 = old\_logprobs - logprobs$$

$$k2 = (exp(old\_logprobs - logprobs) - 1) - (old\_logprobs - logprobs)$$

Another exploration strategy is available through a manual decay of the policy's log standard deviation. Instead of learning a fixed state-independent variance, the standard deviation can be decayed gradually according to a specified decay rate.

IQL. The IQL configuration enables both offline and hybrid online fine-tuning workflows. Training must begin from a pre-existing replay buffer and proceeds in two phases, a fixed number of offline updates, followed by optional online updates during active environment interaction. IQL hyperparameters include the *learning rate*, *batch size*, *discount factor*, and *expectile coefficient*, which determines how strongly underperforming samples influence the critic update. The target network is updated using soft updates with coefficient *tau*. During online fine-tuning, gradient updates occur every *policy_freq* environment steps, allowing control over update density relative to data collection speed. A temperature parameter governs the strength of advantage weighting in the actor update, balancing exploitation and conservatism.

TD3. Hyperparameters include the replay buffer capacity and a minimum buffer fill threshold before training begins. Policy and critic networks are updated at distinct frequencies, with *policy_freq* defining how often the actor is optimized relative to the critic. *Exploration noise* is added to policy outputs during data collection, while *policy_noise* and *noise_clip* govern target policy smoothing and clipping for stability. Target networks are updated using the soft-update rate *tau*.

Fire Model Parameters

Rendering and physical simulation parameters of the fire model are also subject to parameterization. Rendering parameters allow for different visual representations, while simulation parameters control the CA behaviour that is described in Section 4.1.2.

Environment and Agent Configuration

Fire initialization specifies how fires are ignited at the beginning of each episode. Parameters include initial fire percentage, number of ignition clusters, and spatial spread factors. These values affect only the initial fire setup and do not influence the dynamic fire propagation governed by the CA model. A more detailed description of this option is described in Section 5.1.

Agent initialization defines how individual drones are positioned within the environment at the beginning of each episode. A portion of agents can be deployed from a Groundstation, while the remaining agents are distributed randomly across the grid. During setup, each agent is assigned an initial goal, with a configurable fraction of

`FlyAgents` directed toward active fire locations or towards corner positions. A detailed description of the agent initialization process is provided in Section 5.1.

Physical and behavioral properties are also subject to parameterization. Each agent type is configured with its own number of instances, time horizon, observation range, and maximum speed, while all agents, regardless of their hierarchy, share the same physical size. This allows agents to operate with different temporal stacking regarding their observation space, different view ranges and to operate at different speed. Reward coefficients mentioned in Section 4.2.4 can also be set here.

5 Evaluation

This chapter presents the experimental evaluation of the hierarchical reinforcement learning system implemented in ROSHAN. Experiments are structured according to the two hierarchical layers of control, the `FlyAgent` as the low-level policy and the `PlannerAgent` as the high-level goal coordinator, each of which is trained and evaluated independently. The chapter also describes the simulation environment, hyperparameter optimization, and evaluation metrics used throughout all experiments.

5.1 Environment Configuration

Unless stated otherwise, all experiments in ROSHAN are conducted under the default configuration described in this section and each experiment is repeated with five independent random seeds to ensure statistical robustness. These settings define the temporal resolution of the simulation, the duration and termination criteria of each episode, the initialization of fires, and the agent configurations used for both hierarchical levels.

General Simulation Parameters. Each simulation step advances the environment by a fixed control interval of $\Delta t = 0.1\,\mathrm{s}$, during which fire dynamics, particle emission, agent observations, and actions are updated. To provide temporal context for the neural networks, all observations are frame-stacked over three consecutive timesteps.

Termination. Episodes terminate either when all fires are extinguished or when a dynamically computed step limit $T_{\max}$ is reached. Unlike a fixed cutoff, $T_{\max}$ in ROSHAN is derived from the physical map dimensions, agent speed, and several scaling factors that approximate non-optimal flight paths, turning maneuvers, and extinguishing time. This formulation ensures that each agent hierarchy receives a proportional episode duration relative to its operational complexity.

Let the diagonal of the map be

$$D = L_{\text{cell}}\sqrt{N_x^2 + N_y^2},$$

where L_{cell} is the cell edge length and N_x, N_y are the grid dimensions. Dividing D by the agent's maximum velocity $v_{\max}$ yields a baseline traversal time, which is inflated by a turn-efficiency factor $k_{\text{turn}} = 1.5$ and a slack coefficient $s = 2.0$ to account for curved trajectories, exploration, and idling:

$$T_{\text{phys}} = \frac{k_{\text{turn}} D}{v_{\max}\,\Delta t}, \qquad T_{\text{slack}} = s\,T_{\text{phys}}.$$

Depending on the active hierarchy, additional scaling terms are applied to compute the final episode horizon $T_{\max}$:

- **FlyAgents.** For low-level controllers that don't use the complex policy described in Section 4.2.7 $T_{\max} = T_{\text{slack}}$. Otherwise, the episode length scales with traversal time and increases with map size and the number of active fires:

$$T_{\max} = T_{\text{slack}} \cdot \beta_1 \frac{\sqrt{A}\sqrt{F}}{v_{\max}},$$

  where $A = N_x N_y$ is the map area in cells, F is the number of initially ignited fires, and β_1 is a scaling coefficient. This formulation increases $T_{\max}$ for larger environments or higher fire densities, ensuring sufficient time for convergence.

- **PlannerAgents.** For high-level coordination, the horizon additionally accounts for the total travel and extinguishing effort required by all subordinate agents:

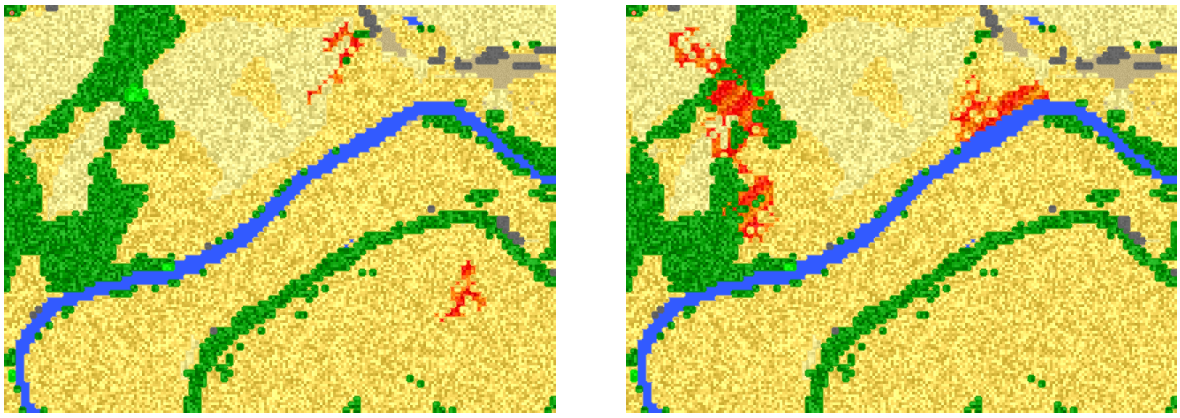
$$T_{\max} = \max\left(T_{\text{slack}}, \frac{1}{\Delta t}\left(\frac{\beta_2\sqrt{A}\sqrt{F}}{v_{\max}} + F\, t_{\text{fire}}\right)\right),$$

  where β_2 is a spatial scaling coefficient and t_{fire} denotes the average extinguishing time per fire. The maximum operator ensures that $T_{\max}$ never falls below the slack-adjusted traversal time, preventing premature termination on small or low-density maps.

Fire Initialization. At the beginning of each episode, a small portion of the environment is ignited. Rather than enforcing a fixed number of burning cells, the system interprets the configured fire percentage as an upper limit on the total number of possible ignitions. The actual number of ignited cells depends on the spatial layout of the map and the stochastic spread dynamics during initialization. Fires are generated in clusters to emulate varying ignition patterns. Each cluster begins from a randomly selected, flammable seed cell and expands outward through its neighboring cells according to a probabilistic spread rule. For every neighboring cell, a random draw determines whether ignition occurs, influenced by a base spread probability and a noise term that perturbs this probability to introduce spatial irregularity.

By default, 1% of all cells are considered as potential ignition candidates, distributed across two distinct clusters. Figure 5.1 shows two representative initialization configurations at different ignition densities.

FireSPIN Model Parameters. The fire dynamics are simulated using the fire model described in Section 4.1.2. Unless otherwise stated, all experiments use the default parameterization summarized in Table 5.1.



(a) Clustered fire initialization with 1% ignition density and two clusters.

(b) Clustered fire initialization with 10% ignition density and two clusters.

Figure 5.1: Comparison of clustered fire initialization patterns in ROSHAN. Red cells represent ignited areas.

Table 5.1: Default FireSPIN parameters used in the experiments. Values correspond to those introduced in Section 4.1.2.

Symbol	Description	Default Value	Unit
Convection Particle Parameters			
Y_{st}	Initial particle intensity	1.0	–
Y_{lim}	Ignition threshold intensity	0.20	–
F_l	Position scaling factor	0.15	–
C_0	Turbulent kinetic constant	1.98	–
τ_{mem}	Particle decay timescale	10	s
L_t	Integral length scale	80	m
Radiation Particle Parameters			
$Y_{\text{st,rad}}$	Initial radiation intensity	1.0	–
$Y_{\text{lim,rad}}$	Radiation ignition threshold	0.165	–
Wind Field Parameters			
U_w	Wind speed at 10 m height	10.0	m/s
α_w	Wind direction (random if $= -1$)	-1.0	deg
a	Wind scaling component	0.314	–

Agent Configuration. All experiments involving physical agents are conducted with a default team of four `FlyAgents` during both training and evaluation. Each agent operates with a maximum velocity of 10 m/s and a local observation radius of 10 cells and collision enabled. The `FlyAgent` policy network is shared among all agents, this allows training with any number of agents while maintaining flexibility at inference time. A network trained on n agents can therefore be deployed with any number of agents without architectural modification.

During hierarchical training, the `PlannerAgent` supervises a team of four `FlyAgents`

that act as fixed low-level controllers and extinguishers. Unlike the `FlyAgent` policy, the `PlannerAgent` network structure is bound to a fixed number of subordinate agents due to its input dimensionality. For evaluation, an expanded configuration of twelve `ExploreAgents`, each with a view range of 20 cells and a maximum speed of 20 m/s, is employed to provide state observations for high-level coordination.

All physical agents are initially deployed from a designated *Groundstation*. The Groundstation is represented by a fixed cell within the grid map, which is randomly assigned to one of the four map corners at the beginning of each episode, offset by one cell from the boundary. This ensures consistent takeoff conditions while introducing minor spatial variation across runs. Each agent's precise starting position within the Groundstation is determined by a circular offset pattern, distributing multiple agents evenly around the cell center to avoid immediate overlaps at deployment. A close-up view of four `FlyAgents` positioned at their respective Groundstation offsets is shown in Figure 5.2.

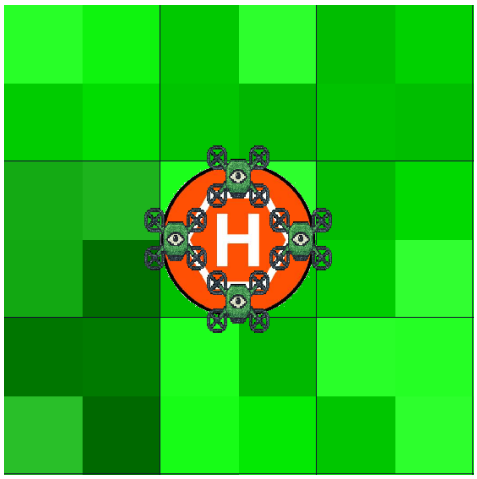


Figure 5.2: Close-up view of four `FlyAgents` initialized at the Groundstation. Agents are evenly distributed in a circular pattern around the cell center to avoid initial overlap.

5.2 Evaluation Metrics

Performance is measured using a set of task-specific metrics that reflect both local and global objectives within the hierarchical control structure:

- **TTE:** Number of simulation steps required to extinguish all active fires.
- **Success Rate:** Fraction of episodes completed successfully without triggering a failure condition.

The relevance of each metric depends on the hierarchy level under evaluation.

The `FlyAgent` is not directly responsible for extinguishing fires, so the objective is strictly local, reaching the assigned goal location, thus only the **Success Rate** is considered a meaningful indicator of performance.

In contrast, the `PlannerAgent` operates at the global level and is evaluated by all three metrics. Its goal is to coordinate multiple `FlyAgents` such that all fires are extinguished efficiently. Accordingly, **TTE** and **Success Rate** serve both as quantitative indicators. Additionally, the `PlannerAgent` is compared against the **greedy heuristic baseline** described in Section 4.2.7.

For all experiments, episodes that end unsuccessfully are classified according to their **Failure Reason**, which provides additional diagnostic insight.

5.3 Training Hierarchical Components

Each hierarchy level in ROSHAN is trained independently to allow for isolated evaluation and incremental refinement. The `FlyAgent` constitutes the low-level controller, responsible for navigation and extinguishing fires within a continuous action space. The `PlannerAgent` forms the high-level coordinator, assigning spatial goals to multiple subordinate agents. The training results of the `FlyAgent` serve as the methodological foundation for all subsequent hierarchical experiments.

5.3.1 FlyAgent Experiments

This section presents a series of experiments conducted on the `FlyAgent`. The experimental progression follows an incremental development strategy. It begins with a simple baseline configuration under collision-free conditions and then switches to inter-agent collision avoidance. Insights and stable design choices identified in earlier experiments are retained in subsequent ones where feasible, ensuring that improvements accumulate in a controlled and interpretable manner.

General Setup. Unless otherwise stated, all experiments in this section are trained for 300 training steps with 10 random seeds. Each policy is evaluated over 200 independent episodes and on a map of size $350m \times 360m$. Seeds for `FlyAgent` evaluation are intentionally left random to mimic minor differences in the environment. Experiments use the environmental setup described in Section 5.1. The standard hyperparameter configuration for the evaluated algorithms can be seen in Table 5.2, likewise, the standard reward coefficients used are shown in Table 5.3

E1 – PPO vs. TD3

Description. This experiment evaluates whether PPO or TD3 performs better in the simplest `FlyAgent` setup, with both algorithms trained on the same amount of environment interaction to ensure a fair comparison.

Parameter	PPO	TD3	IQL
Shared Parameters			
Learning rate (lr)	1.575×10^{-4}	1.0×10^{-4}	1.575×10^{-4}
Batch size	300	512	250
Horizon	1200	—	—
Discount (γ)	0.9635	0.99	0.99
Replay memory size	—	1.0×10^{6}	1.0×10^{6}
τ (soft update)	—	0.005	0.005
Encoder sharing	false	false	true
Tanh-Distri	false	—	true
k_{epochs}	3	—	—
λ (GAE)	0.9	—	—
Value loss coef	0.5	—	—
Entropy coef	0.01	—	—
Min memory size	—	30000	—
Policy update freq	—	2	—
Policy noise	—	0.2	—
Exploration noise	—	0.15	—
Noise clip	—	0.5	—
Temperature	—	—	3.0
Expectile	—	—	0.7
Offline updates	—	—	10
Online updates	—	—	0

Table 5.2: Comparison of Hyperparameters across PPO, TD3, and IQL

β_{goal}	β_{boundary}	β_{ext}	β_{timeout}	β_{coll}	β_{progress}	β_{prox}
$+1.0$	-1.0	$+0.01$	-1.0	-1.0	$+0.1$	-0.05

Table 5.3: Reward coefficients for the `FlyAgent`.

Setup. PPO and TD3 use the hyperparameter configurations shown in Table 5.2 and share the same reward coefficients listed in Table 5.3. For this experiment collision of the agents is turned off and both algorithms use the simple encoder described in Section 4.2.5. Both algorithms are trained using the R_{simple} reward signal described in Section 4.2.4. PPO collects data in fixed-length rollouts. One PPO update corresponds to 1200 observations, which are collected with 4 agents. For this experiment PPO is trained for 225 training steps, this results in approximately $225 \times 1200 = 27{,}000$ transitions. In contrast, TD3 does not rely on fixed rollouts and instead performs updates continuously at each environment step. To match the total amount of environment interaction used by PPO, TD3 is trained for 60,000 environment steps with 4 agents, yielding the same total of roughly 270,000 transitions (after accounting for a warm-up phase of 30,000 transitions to fill the replay buffer). Since TD3 is off-policy, its performance depends on the *update-to-data* (UTD) ratio, the number of gradient updates performed per collected transition. With 4 agents producing 4 transitions per environment step, performing only a single update per step would reduce the UTD to 0.25. To maintain the standard UTD

ratio of approximately 1, TD3 therefore performs four gradient updates per environment step.

Results and Discussion. The performance difference between PPO and TD3 is substantial. As shown in Figure 5.3, PPO reliably converges to high success, achieving a mean success rate of 91.15% with a median of 100% across seeds. In contrast, TD3 exhibits near-complete failure under the same training budget (Figure 5.4). The mean success rate remains below 5%, and most episodes terminate via boundary violations. The high boundary-exit rate reflects unstable behavior and a failure to acquire even basic goal-directed navigation. While it is possible that TD3 could eventually converge with significantly more training, its sample efficiency in this setting is markedly poor. The computational cost is another practical disadvantage, under identical hardware conditions, the PPO experiment completes in **approximately 32 minutes**, whereas TD3 requires **over 24 hours** to process the same number of transitions. As network architectures and number of transitions scale in later experiments, this performance gap would only widen. Given its substantially higher sample efficiency, stable convergence behavior, and dramatically lower runtime, PPO is selected as the primary algorithm for subsequent experiments.

Figure 5.3: E1 PPO. Success: 91.15% ± 17.46% (median 100.0%), **Boundary Exit:** 0.20% ± 0.40% (median 0.0%), **Timeout:** 8.65% ± 17.31% (median 0.0%).

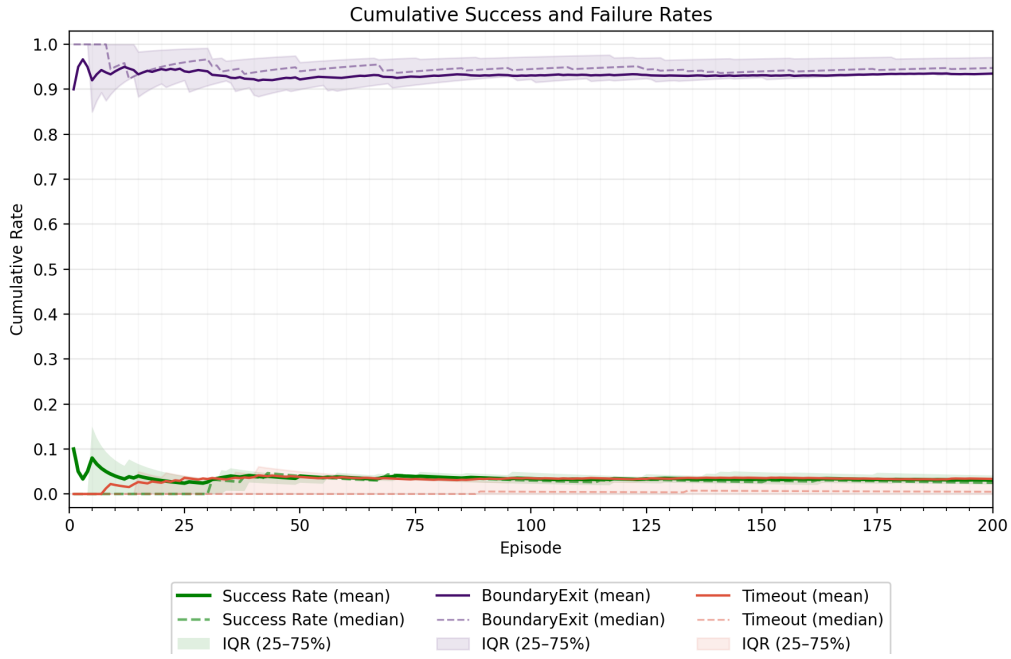


Figure 5.4: E1 TD3. Success: 3.10% ± 1.48% (median 2.5%), **Boundary Exit:** 93.50% ± 5.85% (median 94.75%), **Timeout:** 3.35% ± 6.10% (median 0.5%).

E2 – Encoder Sharing Enabled

Description. The primary question explored is whether sharing the inputspace encoder between the policy and value networks is beneficial for performance.

Setup. This experiment differs from E1 in three ways, only PPO is used for training, training is extended to 300 episodes, and encoder sharing between actor and critic networks is enabled, following the architecture described in Section 4.2.5.

Results and Discussion. Figures 5.5 and 5.6 compare the performance with and without encoder sharing. Without sharing (Fig. 5.5), PPO achieves an average success rate of 83.2% with wide performance variance across seeds, indicating that learning is sensitive to initialization. Timeout failures also remain noticeable, suggesting slower progress toward goals for some runs. When encoder sharing is enabled (Fig. 5.6), performance improves across all metrics. The mean success rate increases to 89.1%, with reduced variation across seeds. Timeout rates are similarly reduced, showing that agents reach goals more consistently and efficiently. The improvement in sample efficiency is particularly visible in the steeper early learning curve, indicating faster convergence. This improvement is also reflected in the distribution of outcomes across seeds. In both configurations, the median success rate reaches nearly 100%, yet the mean remains noticeably lower. This discrepancy arises from a small number of runs that underperform severely, disproportionately reducing the mean. However, in the shared-encoder configuration, the interquartile range is almost indistinguishable in Figure 5.6, indicating that most seeds achieve consistently high success. In contrast, without sharing, variability is substantially larger and failure cases occur more frequently. The improvement in robustness is achieved without increasing model capacity, making encoder sharing a strictly more stable and efficient architectural choice in this setting and it is therefore used in all subsequent experiments.

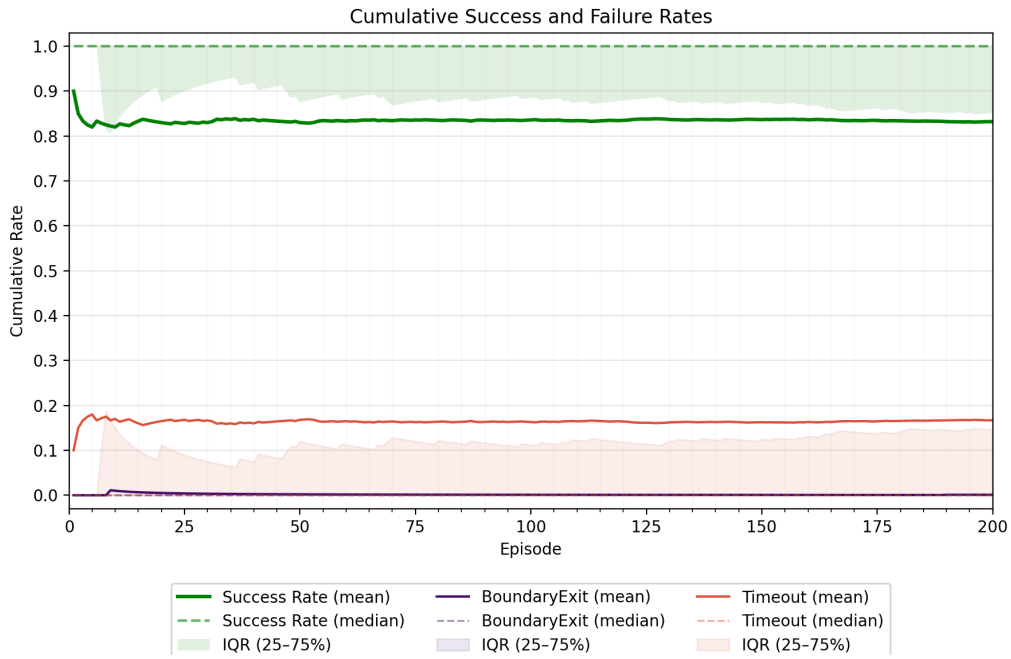


Figure 5.5: E2 Encoder Sharing off. Success: 83.20% ± 29.84% (median 100.00%), **Boundary Exit:** 0.10%±0.3% (median 0.0%), **Timeout:** 16.7%±29.89% (median 0.0%).

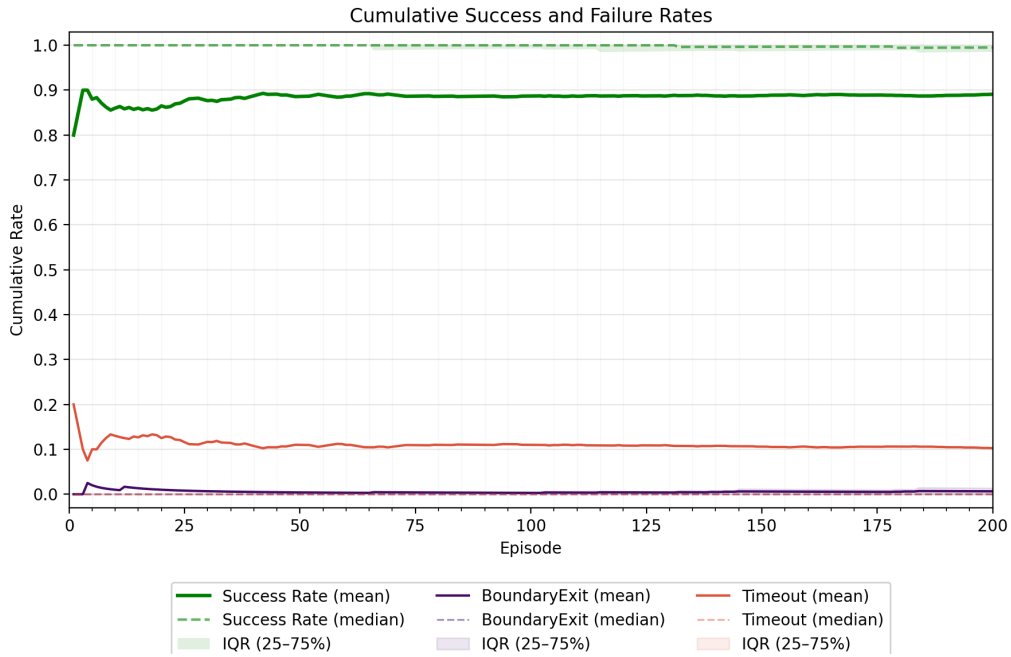


Figure 5.6: E2 Encoder Sharing on. Success: 89.10% ± 21.44% (median 99.50%), **Boundary Exit:** 0.65% ± 0.87% (median 0.0%), **Timeout:** 10.25% ± 21.01% (median 0.0%).

E3 – Collisions Enabled and Tanh-Distribution

Description. This experiment extends the previous setup by reintroducing inter-agent collision dynamics, which increases the difficulty of coordinated navigation and goal-reaching. In addition to evaluating performance under collisions, this experiment also investigates the effect of the action distribution used by PPO. Earlier experiments (and the original OpenAI PPO implementation) employ an unbounded Gaussian policy and apply action clipping after sampling. Here, we compare this approach against a squashed Tanh-Gaussian distribution, where the policy samples in an unconstrained space and the Tanh transform maps actions smoothly into the allowed range. The goal is to assess whether the squashed policy improves learning stability and smoothness of control in the more challenging collision-enabled scenario.

Setup. Both configurations use the collision-aware input encoder described in Section 4.2.5 and enable the complex policy structure introduced in Section 4.2.7. The reward signal is updated to the collision-sensitive variant $R_{\text{complex\_collision}}$ as defined in Section 4.2.4. The two variants therefore only differ in the action distribution used.

Results and Discussion. The training horizon in this experiment remains relatively short (1200 transitions per update over only 300 training episodes), which limits how much stable coordination behavior can emerge. Under these conditions, enabling collisions substantially increases task difficulty, and both configurations achieve low performance (Figures 5.7 and 5.8). Median success rates drop to roughly 3%, with most episodes ending in either collision or timeout, indicating that agents struggle to jointly avoid each other while making progress toward goals. Comparing the two action distributions, the Tanh-squashed policy does not improve overall success rates. However, it does meaningfully change the failure profile. The Tanh-squashed policy reduces collisions on average (from 34.0% to 23.05%), but this is accompanied by higher timeout rates (from 49.15% to 64.85%), suggesting more conservative motion. Importantly, the variance across seeds is substantially lower in the Tanh-squashed case, performance clusters around its median, whereas the unclipped Gaussian policy exhibits much wider variability and several seeds diverge drastically. The contrasting behaviors highlight a trade-off, the unclipped Gaussian policy yields more decisive but unstable actions, while the Tanh-squashed policy yields stable but overly cautious behavior. It is not entirely clear at this point, wheather the Tanh-squashed policy provides a real benefit.

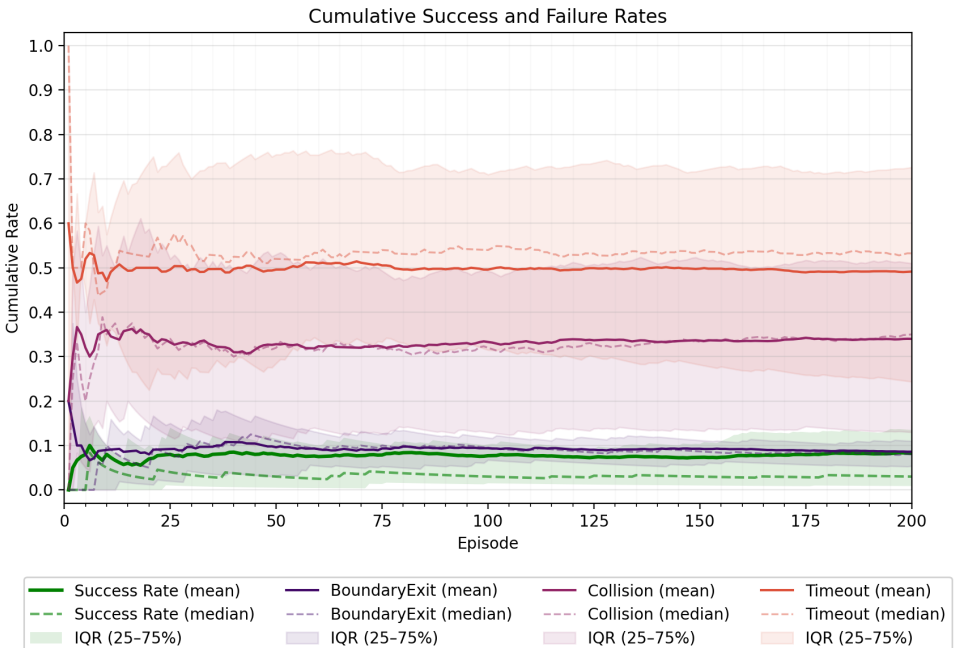


Figure 5.7: E3 (Collision enabled). Success: 8.25% ± 9.98% (median 3.0%), **Boundary Exit:** 8.6% ± 4.71% (median 8.00%), **Collision:** 34.00% ± 21.14% (median 35.00%), **Timeout:** 49.15% ± 26.44% (median 53.25%).

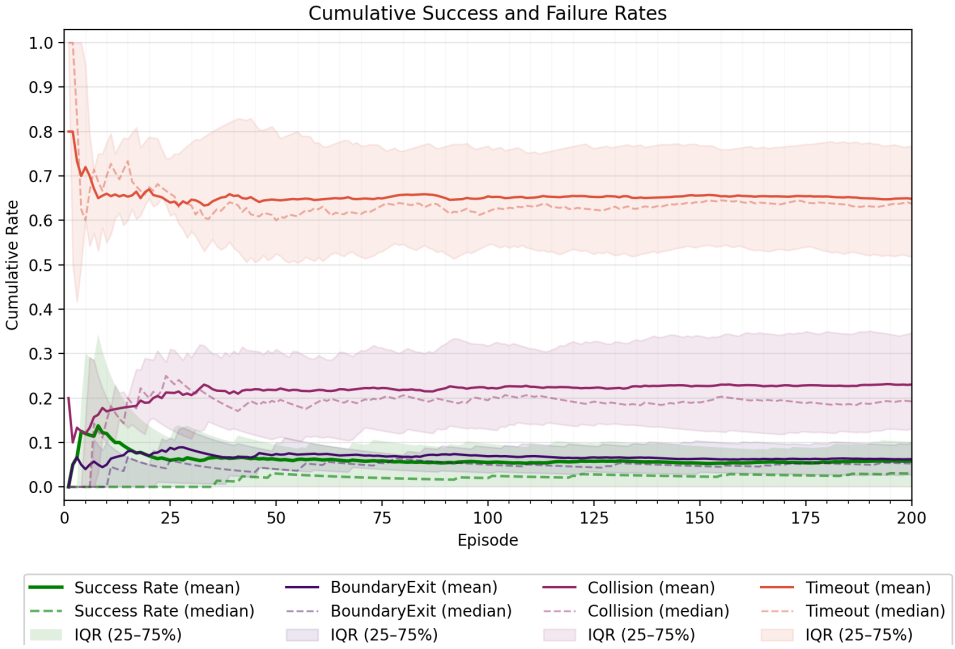


Figure 5.8: E3 (Collision enabled + Tanh). Success: 5.85% ± 6.99% (median 3.0%), **Boundary Exit:** 6.25% ± 4.38% (median 5.25%), **Collision:** 23.05% ± 11.21% (median 19.25%), **Timeout:** 64.85% ± 15.34% (median 63.75%).

E4 – Hyperparameter Tuning: Reward

Description. This experiment investigates the effect of reward shaping on the collision-enabled `FlyAgent`. A hyperparameter search over reward coefficients was performed using Optuna in order to identify a more informative reward signal. Additionally, the effect of the Tanh-squashed action distribution is revisited in this improved reward setting.

Setup. The hyperparameter search optimized the coefficients of the reward terms listed in Section 4.2.4. Each Optuna trial trained PPO with collision enabled for a fixed training horizon. The search spanned the ranges:

- $\beta_{\text{goal}} \in [0.2, 5]$
- $\beta_{\text{boundary}}, \beta_{\text{timeout}} \in [-5, -0.2]$
- $\beta_{\text{ext}} \in [10^{-3}, 0.2]$
- $\beta_{\text{progress}} \in [10^{-3}, 0.5]$ (log-scale)
- $\beta_{\text{prox}} \in [-0.5, -10^{-3}]$

A total of 111 trials were conducted. The best-performing trial initially appeared promising, but subsequent retraining showed substantial instability, consistent with prior observations that even 10-seed evaluations can still produce outliers. Therefore, the top 10 performing trials were re-evaluated, and the reward coefficients were averaged to obtain a more robust and representative reward signal. Two final experiments were then run using this aggregated reward signal, one with the standard clipped Gaussian policy and one with the Tanh-squashed Gaussian policy.

β_{goal}	β_{boundary}	β_{ext}	β_{timeout}	β_{coll}	β_{progress}	β_{prox}
$+0.230$	-4.737	$+0.0128$	-3.050	-4.737	$+0.278$	-0.189

Table 5.4: Reward coefficients from the highest-scoring Optuna trial (later identified as unstable).

β_{goal}	β_{boundary}	β_{ext}	β_{timeout}	β_{coll}	β_{progress}	β_{prox}
$+0.879$	-4.328	$+0.0072$	-3.732	-4.328	$+0.296$	-0.115

Table 5.5: Reward coefficients averaged over the ten best-performing stable Optuna trials.

Results and Discussion. The first evaluation (Figure 5.9), using the raw best-performing reward coefficients from the Optuna search, produces only a minor improvement over previous experiments. The median success rate remains below 10%, and a substantial proportion of episodes still fail through either collisions or timeouts. This confirms that the highest-scoring single Optuna trial was an outlier. When retrained, its performance does not generalize. This is consistent with earlier observations that

even 10-seed rollouts can produce occasional spurious high scores in this environment. Averaging the coefficients over the ten best stable trials leads to a markedly better reward signal (Figure 5.10). With the aggregated reward, median success increases to nearly 50%, and timeout rates are substantially reduced. While collisions remain common, the agents now learn to make consistent progress toward goals before coordination failures occur. This represents the first configuration in which PPO reliably learns non-trivial joint navigation behavior in the collision-enabled setting. Repeating the experiment with the Tanh-squashed action distribution further improves both success and stability (Figure 5.11). Success increases to 57.8% on average with a median of 55.75%, and timeout failures drop sharply to around 6%. Importantly, variance across seeds is substantially lower for the Tanh-squashed policy, consistent with the trend already observed in Experiment E3, the Tanh distribution consistently produces more stable learning outcomes, even when overall success varies across configurations. Interestingly, the failure behavior reverses relative to E3. With the original reward signal, the Tanh policy was too conservative, leading primarily to timeouts. After reward optimization, the Tanh policy no longer suppresses progress. Meanwhile, the unclipped Gaussian policy continues to produce more volatile behavior, occasionally successful, but with noticeably greater variability and more frequent catastrophic episodes. In this configuration, Tanh reduces variability, lowers timeout failures, and improves overall performance stability.

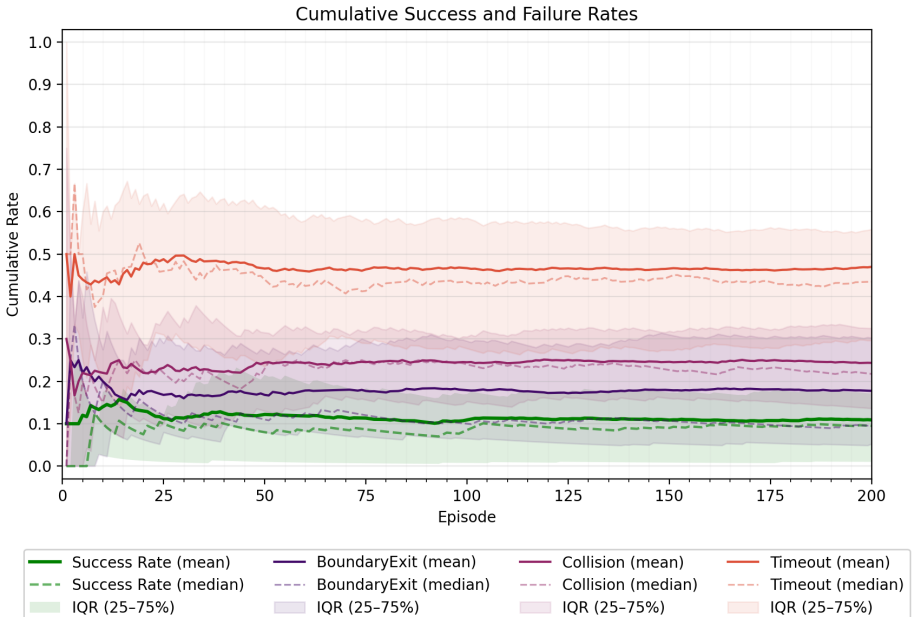


Figure 5.9: E4 (Best Reward). Success: 10.90%±10.38% (median 9.50%), **Boundary Exit:** 17.75% ± 16.02% (median 9.50%), **Collision:** 24.35% ± 15.46% (median 21.75%), **Timeout:** 47.00% ± 21.14% (median 43.50%).

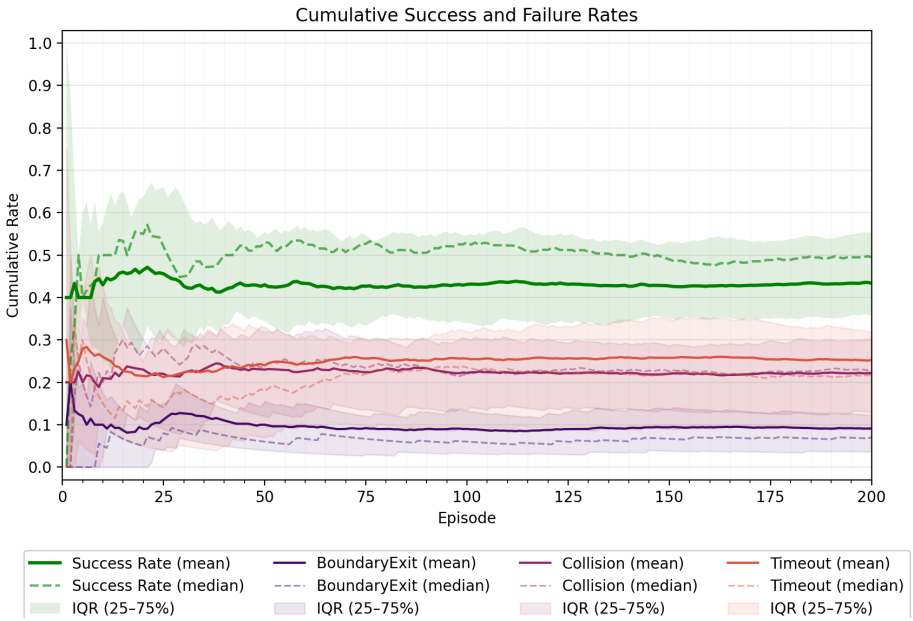


Figure 5.10: E4 (Mean Reward). Success: 43.40% ± 19.41% (median 49.50%), **Boundary Exit:** 9.10% ± 7.66% (median 7.00%), **Collision:** 22.20% ± 11.67% (median 22.75%), **Timeout:** 25.3% ± 20.33% (median 21.75%).

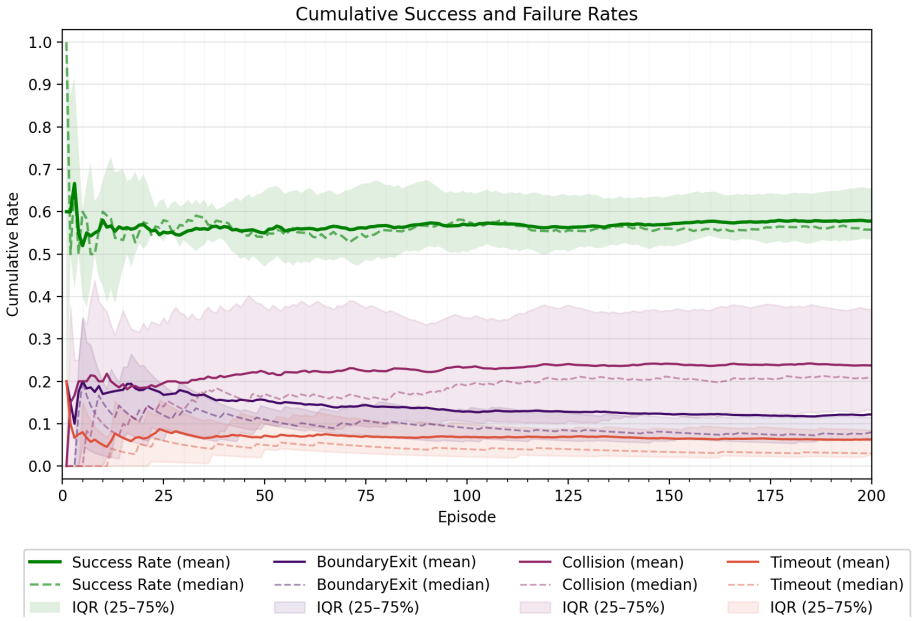


Figure 5.11: E4 (Mean + Tanh). Success: 57.80% ± 10.18% (median 55.75%), **Boundary Exit:** 12.2% ± 14.06% (median 8.00%), **Collision:** 23.75% ± 12.45% (median 20.75%), **Timeout:** 6.25% ± 6.17% (median 3.00%).

E5 – Medium Horizon Scaling + ClipKL

Description. The previous experiments demonstrated that meaningful coordination can emerge under collision dynamics once reward shaping and encoder sharing are introduced. However, training remained constrained by a relatively short horizon and modest batch size. In this experiment, the training horizon and batch size are both increased by an order of magnitude, aiming to provide a more stable gradient signal. Additionally, we test whether applying early stopping based on approximate KL Divergence prevents catastrophic policy shifts.

Setup. The training horizon is increased from 1200 to 12,000 transitions per PPO update, and the batch size is increased from 300 to 1000. All other settings match the best-performing configuration from Experiment E4, including the aggregated reward coefficients (Table 5.5), encoder sharing, collision encoder, and the Tanh-squashed action distribution. Two variants are evaluated, one with and one without early approximate KL Divergence based stopping, where the threshold is set at 0.03.

Results and Discussion. Increasing the training horizon alone leads to a substantial improvement in performance compared to all prior collision-enabled experiments. As shown in Figure 5.12, the agent achieves a mean success rate of 85.1%, with extremely low timeout rates and only moderate collision frequency. Importantly, success rates are consistent across seeds, and failure cases primarily arise from occasional coordination errors rather than training instability. This indicates that longer rollouts and larger batch sizes meaningfully improve both gradient quality and behavioral consistency. However, the TensorBoard traces in Figure 5.14 show that, in this configuration, occasional spikes in the approximate KL Divergence can still occur. These spikes are strongly correlated with sharp performance collapses, reflecting catastrophic policy regressions. Using approximate KL Divergence based early stopping mitigates this failure mode. With the early stopping threshold applied, the mean success rate increases to 90.05%, and collision frequency is reduced by nearly half (from 12.05% to 6.95%), as shown in Figure 5.13. Timeout failures are eliminated entirely. The TensorBoard curves in Figure 5.15 confirm that the approximate KL Divergence remains within a narrow operating range throughout training, preventing the destabilizing update steps responsible for the previous collapses. The reduced performance variance across seeds further indicates that KL Divergence based early stopping effectively stabilizes learning without suppressing policy improvement.

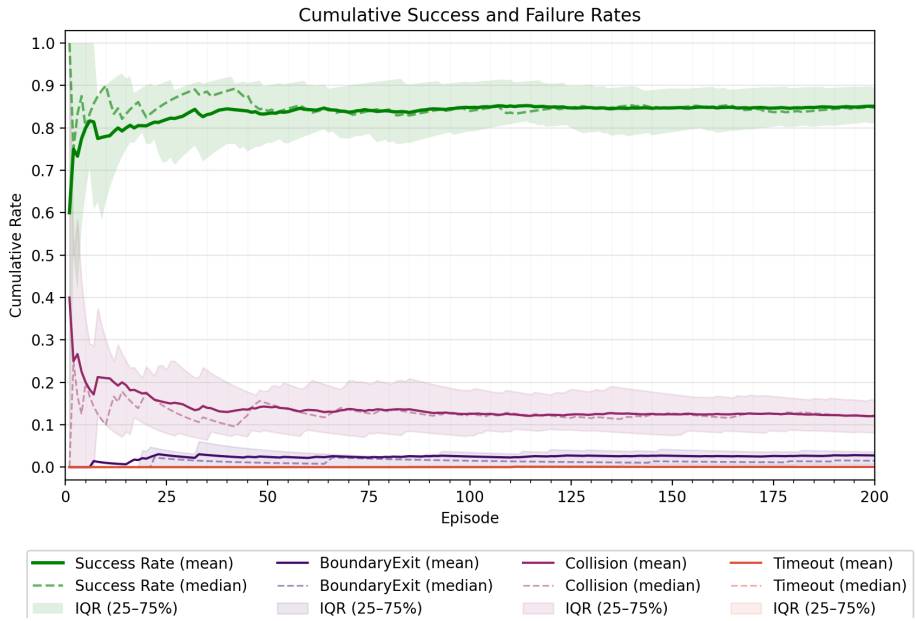


Figure 5.12: E5 (Medium Horizon). Success: 85.10% ± 4.78% (median 84.75%), **Boundary Exit:** 2.75% ± 3.15% (median 1.5%), **Collision:** 12.05% ± 5.24% (median 12.25%), **Timeout:** 0.05% ± 0.15% (median 0.0%).

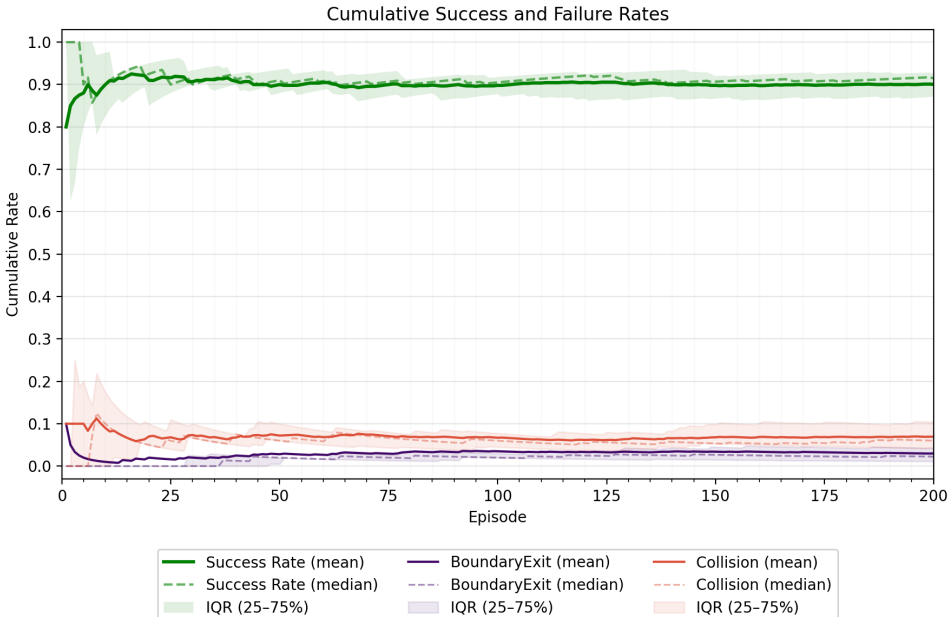


Figure 5.13: E5 (Medium Horizon + KL Cut). Success: 90.05%±3.14% (median 91.5%), **Boundary Exit:** 3.00% ± 2.51% (median 2.25%), **Collision:** 6.95% ± 3.26% (median 6.0%), **Timeout:** 0.00% ± 0.0% (median 0.0%).

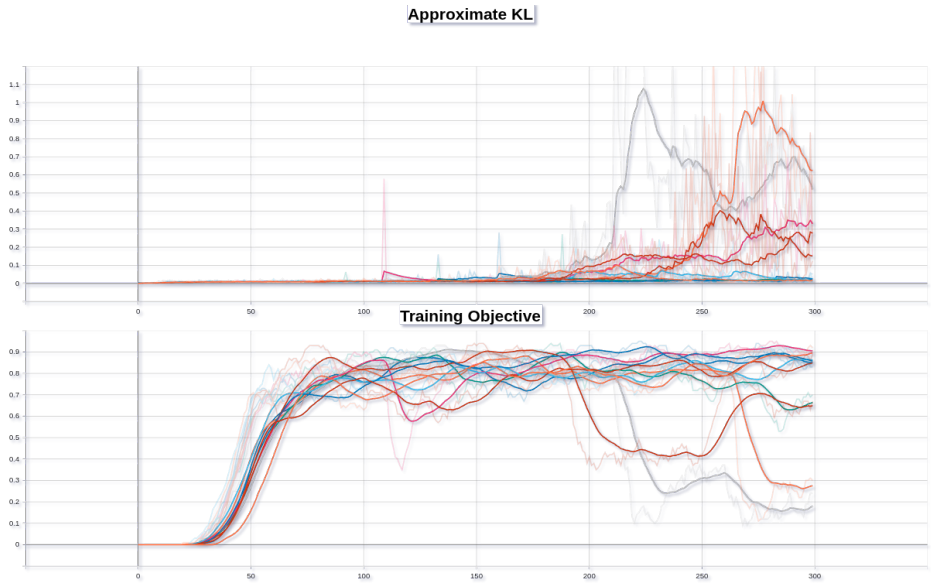


Figure 5.14: Smoothed TensorBoard curves for the Medium Horizon configuration without KL Divergence based early stopping. Each curve represents one training seed. Pronounced spikes in the approximate KL Divergence (top) are strongly correlated with abrupt collapses in task performance (bottom).

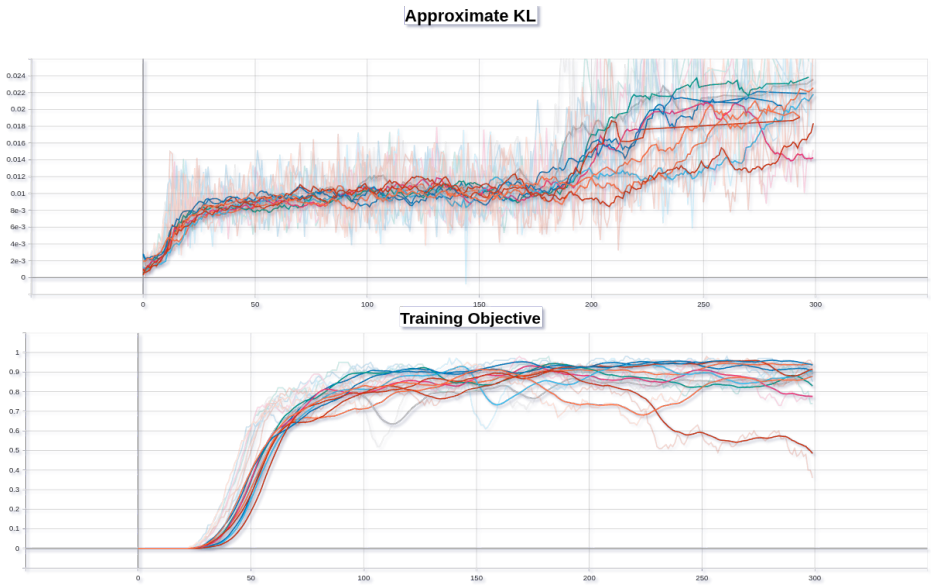


Figure 5.15: Smoothed TensorBoard curves for the configuration with KL Divergence based early stopping enabled. Each curve represents one training seed. The approximate KL Divergence (top) stays within the set threshold throughout training, and the task performance (bottom) remains mostly stable across runs.

E6 – Large Horizon Scaling + ClipKL

Description. Following the improvements observed in Experiment E5, this experiment further increases the training horizon and batch size to evaluate whether longer on-policy rollouts continue to enhance stability and coordinated navigation performance. The goal is to determine whether the benefits of extended horizon training scale to substantially longer trajectories and whether approximate KL Divergence based early stopping remains necessary or helpful under these conditions.

Setup. The training horizon is increased from 12,000 to 66,000 transitions per update, and the batch size is increased from 1000 to 2000 episodes. All other training parameters match the best-performing configuration from Experiment E5. Two configurations are compared, one with and one without early approximate KL Divergence based stopping, where the threshold is set at 0.03. Due to the substantially increased computational cost, the experiment is conducted with 5 seeds. Total training time for this configuration is approximately **23 hours** per condition, and therefore **around 46 hours** for the pair of experiments with and without approximate KL Divergence based early stopping.

Results and Discussion. Both configurations achieve similarly high performance at evaluation time (Figures 5.16 and 5.17), with mean success rates around 94% and zero timeout failures. Collision rates remain low and comparable across seeds, indicating that coordinated navigation strategies are now learned robustly at this training horizon. The performance difference between the two configurations is therefore not reflected in the final evaluation metrics, but rather in the dynamics of the training process. In the configuration without approximate KL Divergence based early stopping, the training traces (Figure 5.18) show spikes in approximate KL Divergence divergence. Unlike the medium-horizon case in Experiment E5, however, these spikes *do not* lead to catastrophic policy collapse. Instead, the policy typically recovers after a while, suggesting that the larger rollout size provides a stronger and more stable gradient signal that can withstand temporary distribution shifts. In contrast, the configuration with approximate KL Divergence early stopping exhibits no large upward spikes in KL Divergence divergence (Figure 5.19). However, a different failure pattern emerges, sharp decreases in the KL Divergence signal correlate with drops in the training objective. As a result, while the training curves appear more stable, this stability does not translate to improved evaluation performance, in fact, the two configurations are nearly indistinguishable in final outcome. These observations indicate that at large horizon sizes, the training signal becomes stable enough that approximate KL Divergence-based early stopping provides limited benefit. While it still prevents extreme update steps, the reduced number of gradient updates can offset its stability advantage. In this setting, long-horizon PPO appears robust, and KL Divergence based early stopping becomes neither harmful nor meaningfully beneficial.

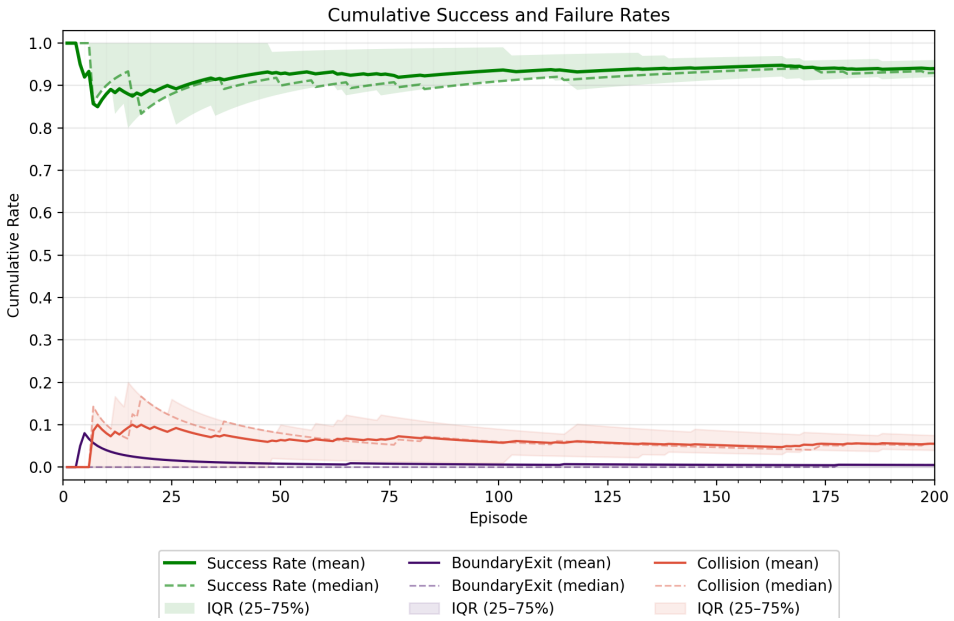


Figure 5.16: E6 (Large Horizon). Success: 94.00% ± 2.88% (median 93.00%), **Boundary Exit:** 0.5%±0.55% (median 0.5%), **Collision:** 5.5%±2.63% (median 5.5%), **Timeout:** 0.00% ± 0.00% (median 0.0%).

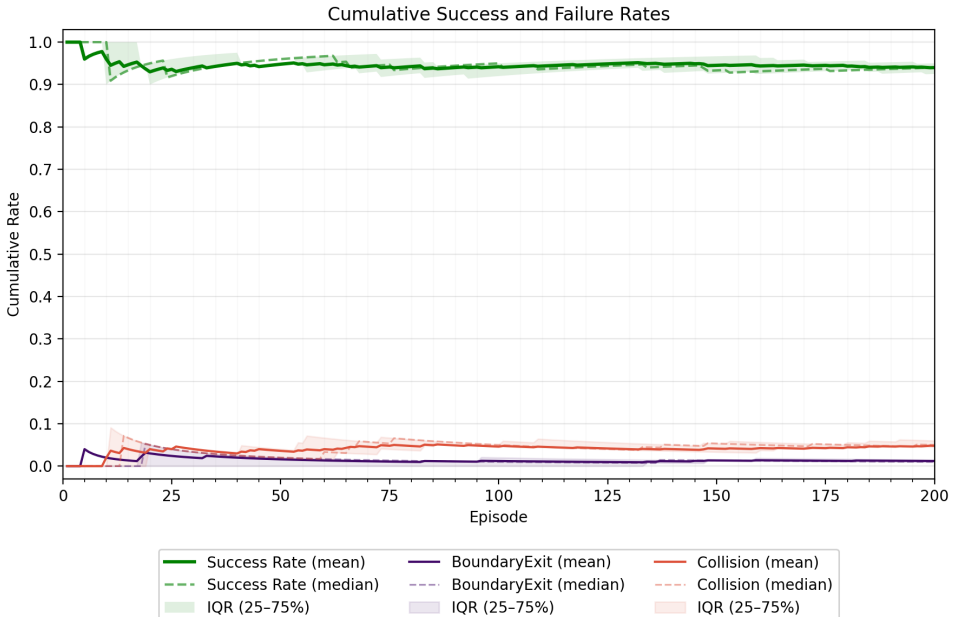


Figure 5.17: E6 (Large Horizon + KL Cut). Success: 94.00% ± 1.92% (median 94.00%), **Boundary Exit:** 1.2% ± 0.8% (median 1.0%), **Collision:** 4.8% ± 1.57% (median 5.0%), **Timeout:** 0.00% ± 0.00% (median 0.0%).

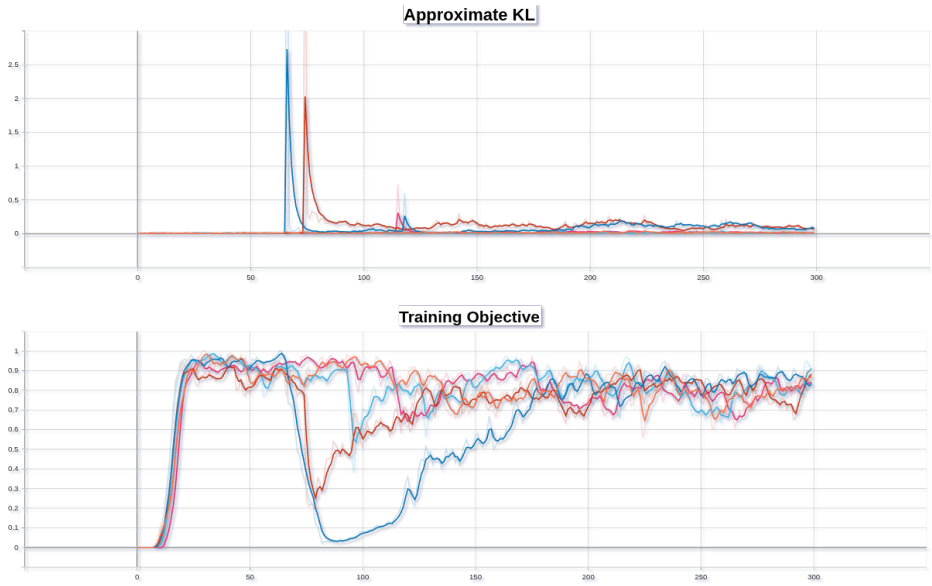


Figure 5.18: Smoothed TensorBoard curves for the Large Horizon configuration **without** KL Divergence based early stopping. Each curve represents one training seed. Approximate KL Divergence exhibits occasional upward spikes, but unlike the medium-horizon case, the policy consistently recovers and no catastrophic performance collapses occur.

Figure 5.19: Smoothed TensorBoard curves for the Large Horizon configuration **with** KL Divergence based early stopping. KL Divergence remains bounded above, but pronounced downward valleys are correlated with unlearning or temporary regressions in the training objective.

E7 – IQL Policy Extraction

Description. During early stages of development, overall performance in the collision-enabled setting was highly sensitive to initialization and reward tuning. At this point, IQL was explored as a potential way to refine a suboptimal but already reasonably performing policy. The idea was to treat the replay buffer of a trained `FlyAgent` as offline data and extract a refined policy via IQL and potentially improving it with online-finetuning. While this approach did not prove to be competitive in the present simulation setting, it was implemented and explored for a while and is therefore revisited in this experiment.

Setup. The replay buffer was collected from the best-performing configuration in Experiment E4 (Mean Aggregated Reward + Tanh-squashed distribution; see Figure 5.11). The buffer size was limited to 200,000 transitions due to memory constraints in the current simulation build. No additional online data collection or fine-tuning was performed in this experiment, IQL was used strictly for **offline policy extraction**. The hyperparameters listed in Table 5.2 were applied without modification. Training proceeded for 10 full offline passes over the dataset, with a batch size of 250, resulting in approximately 8,000 gradient updates (for comparison, the PPO policy used in E4 was trained with roughly 3,600 updates). Due to the increased runtime demands of IQL (similar to TD3), this experiment was performed using 5 seeds.

Results and Discussion. The extracted IQL policy achieves a mean success rate of 51.56% (Figure 5.20), which is moderately lower than the original PPO policy from which the dataset was collected (57.80%, Figure 5.11). Boundary exits remain comparable between the two policies, while collisions and timeouts are slightly higher under IQL. These differences are likely attributable to the relatively small replay buffer and the absence of any online refinement stage, which is a key component of the full IQL training procedure. Overall, the results demonstrate that IQL is capable of extracting a meaningful policy even from limited offline data in this setting. However, achieving further improvements would require online fine-tuning, which is substantially more expensive. In comparison, the long-horizon PPO configuration already required hours of wall-clock time per seed (Section 5.3.1). Given the high computational cost of online refinement and the rapidly evolving simulation environment, IQL was not considered a practical candidate for experimentation and development. Nevertheless, this experiment provides a useful reference point for offline reinforcement learning in coordinated drone fire suppression tasks and highlights the potential of offline pretraining when larger and more diverse replay buffers are available.

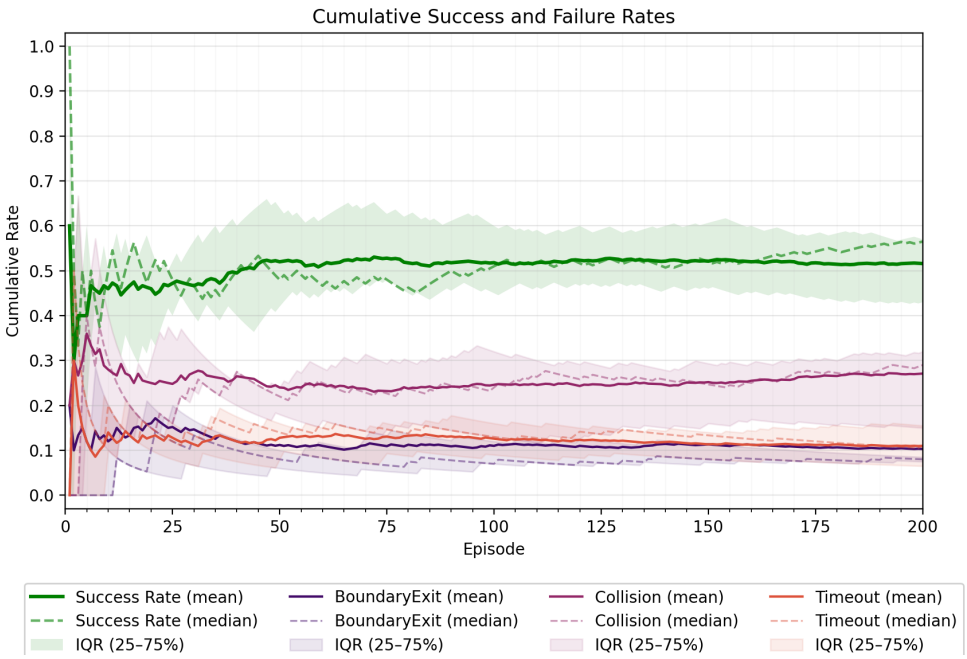


Figure 5.20: E7 (IQL Policy Extraction). Success: $51.56\% \pm 12.90\%$ (median 56.50%), **Boundary Exit:** $10.30\% \pm 5.12\%$ (median 8.00%), **Collision:** $27.10\% \pm 12.52\%$ (median 29.00%), **Timeout:** $11.00\% \pm 6.36\%$ (median 11.00%).

5.3.2 PlannerAgent Experiments

This section evaluates the `PlannerAgent`, which operates at a higher decision-making level by assigning fire targets to subordinate `FlyAgents`. As a reference, all learned policies are compared against the baseline greedy heuristic, whichs results can be observed in Table 5.6. Timeouts in the heuristic can be attributed to agents getting stuck in a state where each individual agent occupies the goal position of another agent.

Metric	Greedy Heuristic
TTE	1860 ± 1104 (1381)
Success	73%
Timeout	27%

Table 5.6: Performance of the baseline greedy heuristic. Entries for TTE report mean $\pm$ standard deviation (median) over evaluation episodes.

General Setup. Unless otherwise stated, all experiments are trained for 100 training steps using 3 random seeds on a $350,\mathrm{m} \times 360,\mathrm{m}$ map. In every experiment, 35% of the map is initially ignited in 2 ignition clusters. Performance is evaluated over 100 episodes on an identically seeded environment and compared directly against the baseline heuristic. Environment seeding ensures reproducibility, while each episode has its own distinct

fire configuration, the same episode will exhibit identical fire placement and propagation across different runs. All evaluations and the heuristic use the best-performing collision-enabled `FlyAgent` configuration obtained in Experiment E6, collision-related episode termination is disabled for both the heuristic and the `PlannerAgent`. Reward coefficients and training hyperparameters follow Table 5.8 and Table 5.7. The reward coefficients were selected from the best performing Optuna trial (53 trials in total). The `PlannerAgent` operates with a hierarchy interval of 20, meaning that it issues one high-level action for every 20 low-level environment steps executed by its subordinate agents.

α_{goal}	α_{burned}	α_{ground}	α_{timeout}	α_{same}	$\alpha_{\text{extinguished}}$	α_{fast}
$+4.656$	-0.886	-0.236	-0.435	-1.913	$+0.523$	$+0.523$

Table 5.7: Reward coefficients for the `FlyAgent`.

Parameter	
Learning rate (lr)	1.0×10^{-4}
Batch size	300
Horizon	1200
Discount (γ)	0.99
Encoder sharing	true
Tanh-Distri	false
k_{epochs}	3
λ (GAE)	0.98
Value loss coef	0.5
Entropy coef	0.0005

Table 5.8: Hyperparameters for PPO for the `PlannerAgent`

P1 - Baseline Planner Training and Reward Refinement

Description. The goal of this experiment is to establish a baseline for the `PlannerAgent` and to evaluate whether improved critic stability and reward shaping lead to more effective fire-goal allocation behavior.

Setup. The initial configuration uses the default hyperparameters (Table 5.8) and reward structure (Table 5.7). During training, this configuration produced unstable value estimates and high clipping ratios, indicating that the value function updates dominated the policy learning signal. To address this, the value loss coefficient was reduced to 0.001, which stabilized training and reduced the approximate KL Divergence and clipping ratio to reasonable levels. In a third configuration, the reward coefficients were replaced by the aggregated top-10 Optuna trial mean (Table 5.9), following the same methodology used in Experiment E4. This optimized reward was combined with the reduced value loss coefficient, as this combination proved beneficial in preliminary testing.

α_{goal}	α_{burned}	α_{ground}	α_{timeout}	α_{same}	$\alpha_{\text{extinguished}}$	α_{fast}
$+3.911$	-2.2	-0.293	-2.384	-1.552	$+0.605$	$+0.217$

Table 5.9: Aggregated reward coefficients from the top 10 Optuna trials.

Results and Discussion. Reducing the value loss coefficient improves performance compared to the default configuration (Table 5.10), increasing median success from 66% to 83% and lowering the timeout rate accordingly. However, TTE remains substantially higher than the heuristic baseline (Table 5.6), indicating that while the `PlannerAgent` becomes more reliable in reaching successful outcomes, it does not yet consistently plan efficient assignments. When using the aggregated reward signal from the Optuna study, performance improves further. The median success rate increases to 95% and timeouts drop to 5%, representing a meaningful gain in robustness across seeds. At the same time, the TTE objective improves only moderately and still remains notably higher than the heuristics median extinguishing time. In comparison to the greedy baseline, the final configuration provides more consistent and higher success rates, but does not yet match the baselines efficiency in minimizing episode duration. These results indicate that while reward shaping and critic stabilization enable the `PlannerAgent` to outperform static assignment heuristics in reliability, further work is needed to explicitly incentivize faster suppression strategies.

Metric	**Default**	**Value Coeff.**	**Optuna Tuned**
TTE	2839 ± 780 (3040)	2557 ± 825 (2500)	2379 ± 780 (2350)
Success	$65.6\% \pm 11.0\%$ (66.0%)	$74.0\% \pm 21.0\%$ (83.0%)	$89.7\% \pm 7.5\%$ (95.0%)
Timeout	$34.3\% \pm 11.0\%$ (34.0%)	$26.0\% \pm 21.0\%$ (17.0%)	$10.3\% \pm 7.5\%$ (5.0%)

Table 5.10: Comparison of `PlannerAgent` performance across training configurations. Each entry reports mean $\pm$ standard deviation (median) over evaluation episodes.

P2 – Planning Frequency (HRL Step Ratio)

Description. This experiment investigates how the planning frequency of the `PlannerAgent` influences performance. The aim is to determine whether more frequent re-planning improves responsiveness to changing fire conditions and agent coordination, or whether it instead introduces instability.

Setup. Planner training uses the same network architecture and optimization settings as before, with the value-loss coefficient fixed to 0.001 and the aggregated reward coefficients. The only parameter varied is the HRL decision interval. In this experiment, the interval is reduced from 20 to 10 and 5, defining how many low-level steps are executed before a new high-level command is issued.

Results and Discussion. Varying the HRL interval affects both the reliability of plan execution and the efficiency of extinguishing fires (Table 5.11). The 5-step configuration achieves lower success than the 10-step setup, but it still performs above the heuristic baseline in both success rate and timeout frequency. Notably, the median TTE is also lower than in the earlier Optuna-tuned 20-step configuration, indicating that more frequent re-planning can improve responsiveness and reduce overall mission duration. The 10-step configuration, however, offers the strongest overall performance. It achieves the highest mean success (90.7%), the lowest timeout rate, and a reduced TTE. This represents the first instance in which a learned `PlannerAgent` surpasses the baseline heuristic both in mean TTE and mean success.

An interesting and counterintuitive observation emerges from this experiment. In earlier development, agents occasionally oscillated between competing fire goals, moving toward one, then switching direction repeatedly. This behavior was initially attributed to overly frequent high-level re-planning, motivating the original choice of 20 HRL steps to stabilize decision-making. However, the results here show the opposite relationship, lower HRL steps improve overall performance.

Metric	5 HRL Steps	10 HRL Steps
TTE	2091 ± 899 (1850)	1822 ± 818 (1630)
Success	$82.0\% \pm 5.0\%$ (85.0%)	$90.7\% \pm 6.6\%$ (87.0%)
Timeout	$18.0\% \pm 5.0\%$ (15.0%)	$9.3\% \pm 6.6\%$ (13.0%)

Table 5.11: Comparison of lower HRL Step Ratio. Each entry reports mean ± standard deviation (median) over evaluation episodes.

P3 – Horizon Scaling

Description. This experiment evaluates how extending the PPO rollout horizon and batch size interacts with the `PlannerAgent`'s planning frequency. The aim is to determine whether larger training horizons improve the stability and effectiveness of high-level decision-making when the `PlannerAgent` re-plans more frequently.

Setup. Building on the configuration from Experiment P2, the HRL decision interval is varied across $3, 5, 10$ steps. For each planning frequency, three progressively increased training horizons are tested. The rollout horizon is increased from 2,400 to 4,800 and 9,600 transitions per update, with corresponding batch sizes of 600, 1,200, and 2,400 episodes per update, respectively. For the largest horizon condition, the *10 HRL steps* configuration is omitted, as it consistently underperformed in previous experiments and was therefore deemed uninformative to evaluate at greater computational cost.

Results and Discussion. Increasing the training horizon further demonstrates the influence of planning frequency on overall performance (Tables 5.12, 5.13, and 5.14). Across all horizon settings, the *5 HRL step* configuration consistently provides the most stable performance. It achieves **100% success** with **no timeouts** in multiple instances and maintains one of the lowest median TTE values observed. This configuration represents the **first case where a learned `PlannerAgent` reliably surpasses the greedy heuristic baseline in both success and mean/median TTE**. However, occasional underperforming seeds still appear, indicating that convergence remains sensitive to initialization.

Metric	3 HRL Steps	5 HRL Steps	10 HRL Steps
TTE	1562 ± 735 (1452)	1434 ± 525 (1393)	1911 ± 829 (1805)
Success	$93.0\% \pm 9.9\%$ (100.0%)	$100.0\% \pm 0.0\%$ (100.0%)	$90.7\% \pm 5.4\%$ (89.0%)
Timeout	$7.0\% \pm 9.9\%$ (0.0%)	$0.0\% \pm 0.0\%$ (0.0%)	$9.3\% \pm 5.4\%$ (11.0%)

Table 5.12: Comparison of HRL Step Ratios with horizon size 2400 and batch size 600. Each entry reports mean ± standard deviation (median) over evaluation episodes.

The *3 HRL step* configuration shows a similar trend. While its average performance is competitive, it also exhibits greater variance due to the presence of more extreme outliers. The best-performing 3-step run achieves a median TTE of 1243, compared to 1205 in the best-performing 5-step run, both surpassing the greedy heuristic. This suggests that both configurations *can* converge to effective behavior, but the 3-step variant is more prone to oscillatory goal-switching, which was first observed in earlier experiments. In contrast, the *10 HRL step* configuration, which previously performed competitively at smaller horizons, consistently underperforms at larger horizons. The reduced re-planning frequency appears insufficient in this environment, resulting in lower success and higher timeout rates. Given the small sample size per condition, this trend should be interpreted cautiously.

Metric	3 HRL Steps	5 HRL Steps	10 HRL Steps
TTE	1393 ± 556 (1299)	1411 ± 505 (1362)	1883 ± 749 (1705)
Success	$98.7\% \pm 1.9\%$ (100.0%)	$100.0\% \pm 0.0\%$ (100.0%)	$93.7\% \pm 8.3\%$ (99.0%)
Timeout	$1.3\% \pm 1.9\%$ (0.0%)	$0.0\% \pm 0.0\%$ (0.0%)	$6.3\% \pm 8.3\%$ (1.0%)

Table 5.13: Comparison of HRL Step Ratios with horizon size 4800 and batch size 1200. Each entry reports mean ± standard deviation (median) over evaluation episodes.

Overall, these results indicate that longer training horizons help reveal the structural advantages of more frequent high-level control. Among the tested configurations, the **5 HRL step setting offers the best stability–performance trade-off** under the current hyperparameters. Meanwhile, the *3 HRL step* configuration can achieve strong performance but seems to be more susceptible to outliers. Importantly, across all longer-horizon settings, the learned `PlannerAgent` now consistently **outperforms the greedy heuristic baseline** in success, median TTE, and timeout rate, demonstrating that effective high-level coordination can be learned when both the temporal resolution of planning and the training horizon are appropriately chosen.

Metric	3 HRL Steps	5 HRL Steps
TTE	1695 ± 687 (1560)	1311 ± 417 (1295)
Success	$95.7\%6.1 \pm \%$ (100.00%)	$100\% \pm 0.0\%$ (100.0%)
Timeout	$4.3\% \pm 6.1\%$ (0.0%)	$0.0\% \pm 0.0\%$ (0.0%)

Table 5.14: Comparison of HRL Step Ratios with horizon size 9600 and batch size 2400. Each entry reports mean ± standard deviation (median) over evaluation episodes.

6 Conclusion

6.1 Summary

This thesis introduced ROSHAN, a HRL framework for autonomous UAV-based wildfire suppression in a controllable CA environment. The system integrates three levels of control, the low-level `FlyAgent`, which performs continuous navigation and local fire extinguishing, the `ExploreAgent`, which maintains coverage of the environment and ensures that newly emerging or previously unseen fire locations enter the global decision space and the high-level `PlannerAgent`, which assigns suppression objectives and allocates `FlyAgents` across active fire regions. This hierarchical decomposition reduces the complexity of long-horizon coordination by separating short-horizon motion control from global allocation decisions. The `FlyAgent` was trained using PPO and evaluated across a sequence of increasingly complex experimental conditions. Early experiments compared PPO and TD3, showing that PPO achieved high success rates while TD3 performed poorly under comparable sample budgets. Subsequent experiments demonstrated that encoder sharing between actor and critic, reward shaping, and the use of Tanh-squashed action distributions improved learning stability. When collisions were introduced, performance initially degraded, but tuning of reward coefficients and the use of KL Divergence based update control restored stable convergence. Increasing the rollout horizon and batch size further improved training stability and resulted in reliable, consistent performance across seeds. These results show that while the `FlyAgent` is capable of learning robust navigation, its performance is highly sensitive to hyperparameter selection, particularly reward shaping, action distribution, and update horizon. The `PlannerAgent` was evaluated against a deterministic greedy assignment heuristic that served as the baseline for coordinated suppression. Under the tested conditions, the `PlannerAgent` surpassed this heuristic when trained with appropriately large rollout horizons and planning frequencies of 5 HRL steps. In these configurations, it achieved higher success rates, fewer timeouts, and lower median TTE than the heuristic. However, performance was again sensitive to hyperparameter settings. Planning frequency, rollout horizon, and value-loss weighting were found to strongly affect convergence and reliability. Configurations that were not well aligned with the temporal dynamics of fire spread resulted in unstable or inconsistent performance across seeds, while well-tuned configurations outperformed the baseline. Overall, the hierarchical approach proved effective within the scope of the tested environment. ROSHAN demonstrated that a trained high-level policy can outperform the heuristic when the temporal abstraction and learning parameters are appropriately matched to the environment. At the same time, the experiments showed that both levels of the hierarchy exhibit notable sensitivity to hyperparameter choice. This indicates that the limiting factor is not the capacity of the policies, but rather the careful calibration of learning dynamics between the hierarchical levels and algorithmic hyperparameter settings. While generalization beyond the tested conditions remains open, the results

provide a concrete indication that HRL is a viable direction for autonomous multi-agent wildfire suppression, with stability and hyperparameter robustness being the primary areas requiring further study.

6.2 Lessons Learned

One of the most substantial challenges in this work was the design of effective state representations for the individual agents. The `FlyAgent` underwent several iterations in this regard. Early versions relied on global information, such as goal distances or map-wide fire states, which resulted in poor learning efficiency and unstable policy behavior. The representation that ultimately led to reliable convergence expressed all positional information relative to the agent itself and normalized spatial inputs to its observation radius. This not only improved training stability but also allowed the policy to generalize across different map sizes, since no absolute map-scale information is assumed. The development of the `ExploreAgent` similarly proved more complicated than expected. Initial attempts aimed to train the agent using reinforcement learning, which had state representations involving large-scale global map features and visual encodings. These approaches either failed to converge or resulted in behavior that did not meaningfully improve global awareness. In contrast, a simpler deterministic exploration strategy provided a reliable basis for ensuring that unobserved regions were periodically revisited. This result shows that not all components of a hierarchical system require learned behavior, in some cases, simple, hand-designed logic can be sufficient. Another lesson learned during development was the value of building automated experimentation and evaluation pipelines early in the process. Hyperparameter tuning, systematic evaluation across random seeds, and automated logging were initially added only after substantial manual experimentation had already taken place. Once introduced, these tools accelerated progress and allowed more confident interpretation of results. In particular, automated hyperparameter search played a big role in identifying stable configurations and would likely have reduced overall development time had it been used from the start.

6.3 Future Directions

Several directions for continued development follow naturally from this work. First, the system would benefit from more extensive and systematic hyperparameter optimization. While the present results show that effective policies can be learned, both agent layers were found to be sensitive to rollout horizon length, value-loss weighting, planning frequency, reward structure and algorithmic specific hyperparameters. Another structured search could refine these parameters further. Alongside this, a more comprehensive exploration of network architectures and model capacities, combined with larger rollout horizons, may improve training stability and yield policies that generalize more reliably across seeds. Second, the high-level decision-making of the `PlannerAgent` provides a clear opportunity for further exploration. In particular, alternative state-space representations could be investigated to improve robustness. Adjusting how global fire

information, map structure, and agent status are encoded may lead to planning strategies that are more adaptive or transferable to new environments, without necessarily increasing the complexity of the underlying control hierarchy. Third, extending the scenario to include real-world operational constraints represents an important step toward practical applicability. In realistic deployments, UAVs are subject to limited flight endurance due to battery capacity, as well as finite fire retardant resources when used for suppression. Introducing these constraints would require the `PlannerAgent` to reason about resource allocation over time, potentially transforming the problem into a joint planning-and-scheduling task. Such an extension would increase the complexity of the learning problem, but also bring the system closer to realistic wildfire response conditions.

Use of Artificial Intelligence Assistance

The writing of this thesis was prepared with support from ChatGPT, an AI language model developed by OpenAI. The tool was used to improve clarity, coherence, and linguistic formulation in the written text. The underlying research questions, conceptual developments, methodological decisions, analyses, and conclusions originate entirely from the author. All interpretative judgments, selection of evidence, academic arguments, and evaluative claims are the result of human reasoning. ChatGPT did not influence the direction of the research nor contribute original conceptual content.

The workflow involving ChatGPT followed a structured and iterative procedure. First, the content for each section was independently drafted in English, including conceptual explanations, theoretical context, and descriptions of experimental setups. ChatGPT was then prompted to reformulate or refine passages, typically with instructions regarding tone, level of formality, and formatting style suitable for academic writing and LaTeX typesetting.

The generated text was subsequently reviewed and revised through multiple refinement stages. Sentences that did not align with the intended meaning, emphasis, or argumentative structure were removed or rephrased manually. This refinement process was primarily subtractive. Rather than requesting corrections for specific expressions, unsuitable formulations were simply omitted and replaced during editing. Instances in which ChatGPT introduced unwarranted interpretation or evaluative commentary were also removed. In this role, ChatGPT functioned as an advanced linguistic editing tool, assisting in the restructuring of sentences and smoothing of transitions.

This disclosure is provided in the interest of transparency and scholarly integrity, acknowledging the use of AI technology while clearly distinguishing between human conceptual authorship and AI-assisted text refinement.

List of Figures

List of Tables

Bibliography

[1] M. W. Jones, J. T. Abatzoglou, S. Veraverbeke, N. Andela, G. Lasslop, M. Forkel, A. J. P. Smith, C. Burton, R. A. Betts, G. R. van der Werf, S. Sitch, J. G. Canadell, C. Santín, C. Kolden, S. H. Doerr, and C. Le Quéré, "Global and regional trends and drivers of fire under climate change," *Reviews of Geophysics*, vol. 60, no. 3, p. e2020RG000726, 2022, e2020RG000726 2020RG000726. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2020RG000726

[2] J. San-Miguel-Ayanz, J. M. Moreno, and A. Camia, "Analysis of large fires in european mediterranean landscapes: Lessons learned and perspectives," *Forest Ecology and Management*, vol. 294, pp. 11–22, 2013, the Mega-fire reality. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378112712006561

[3] L. Caon, V. R. Vallejo, C. J. Ritsema, and V. Geissen, "Effects of wildfire on soil nutrients in mediterranean ecosystems," *Earth-Science Reviews*, vol. 139, pp. 47–58, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0012825214001585

[4] D. L. Martell, "A review of recent forest and wildland fire management decision support systems research," *Current Forestry Reports*, vol. 1, no. 2, pp. 128–137, Jun 2015. [Online]. Available: https://doi.org/10.1007/s40725-015-0011-y

[5] D. Or, E. Furtak-Cole, M. Berli, R. Shillito, H. Ebrahimian, H. Vahdat-Aboueshagh, and S. A. McKenna, "Review of wildfire modeling considering effects on land surfaces," *Earth-Science Reviews*, vol. 245, p. 104569, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0012825223002581

[6] L. Bodrožić, J. Marasović, and D. Stipaničev, "Fire modeling in forest fire management," https://www.researchgate.net/publication/228925788_Fire_modeling_in_forest_fire_management, 2005, department for Modelling and Intelligent Systems, FESB – Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split, Croatia.

[7] "Development and Structure of the Canadian Forest Fire Behavior Prediction System," 1992. [Online]. Available: https://cfs.nrcan.gc.ca/publications?id=10068

[8] R. Rothermel, "A mathematical model for predicting fire spread in wildland fuels." *U.S. Department of Agriculture, Intermountain Forest and Range Experiment Station*, 1971.

[9] M. A. Finney, "FARSITE: Fire Area Simulator—Model Development and Evaluation," U.S. Department of Agriculture, Forest Service, Rocky Mountain Research Station, Ogden, UT, Research Paper RMRS-RP-4, 1998, revised 2004. 47 p. [Online]. Available: http://dx.doi.org/10.2737/RMRS-RP-4

[10] C. Tymstra, R. Bryce, B. Wotton, S. Taylor, and O. Armitage, "Development and structure of Prometheus: the Canadian Wildland Fire Growth Simulation Model," Natural Resources Canada, Canadian Forest Service, Northern Forestry Centre, Edmonton, Alberta, Information Report NOR-X-417, 2010.

[11] J. Coen, "Simulation of the Big Elk Fire using coupled atmosphere-fire modeling," *International Journal of Wildland Fire*, vol. 14, 01 2005.

[12] W. Mell, M. Jenkins, J. Gould, and P. Cheney, "A physics-based approach to modeling grassland fires," *International Journal of Wildland Fire - INT J WILDLAND FIRE*, vol. 16, 01 2007.

[13] J. J. Kari, *Basic Concepts of Cellular Automata.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-540-92910-9_1

[14] B. Drossel and F. Schwabl, "Self-organized critical forest-fire model," *Phys. Rev. Lett.*, vol. 69, pp. 1629–1632, Sep 1992. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.69.1629

[15] J. G. Freire and C. C. DaCamara, "Using cellular automata to simulate wildfire propagation and to assist in fire management," *Natural Hazards and Earth System Sciences*, vol. 19, no. 1, pp. 169–179, 2019. [Online]. Available: https://nhess.copernicus.org/articles/19/169/2019/

[16] E. Mastorakos, S. Gkantonas, G. Efstathiou, and A. Giusti, "A hybrid stochastic Lagrangian – cellular automata framework for modelling fire propagation in inhomogeneous terrains," *Proceedings of the Combustion Institute*, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1540748922002838

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[18] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, ser. Wiley Series in Probability and Statistics. Wiley, 1994. [Online]. Available: https://doi.org/10.1002/9780470316887

[19] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018. [Online]. Available: https://arxiv.org/abs/1506.02438

[20] J. Schmidhuber, "A possibility for implementing curiosity and boredom in model-building neural controllers," in *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats.* Cambridge, MA, USA: MIT Press, 1991, p. 222–227.

[21] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," 2017. [Online]. Available: https://arxiv.org/abs/1705.05363

[22] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," 2016. [Online]. Available: https://arxiv.org/abs/1606.01868

[23] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," 2018. [Online]. Available: https://arxiv.org/abs/1810.12894

[24] S. Fujimoto and S. S. Gu, "A minimalist approach to offline reinforcement learning," 2021. [Online]. Available: https://arxiv.org/abs/2106.06860

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[26] I. Kostrikov, A. Nair, and S. Levine, "Offline reinforcement learning with implicit q-learning," *CoRR*, vol. abs/2110.06169, 2021. [Online]. Available: https://arxiv.org/abs/2110.06169

[27] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv: Learning*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:16326763

[28] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018. [Online]. Available: https://arxiv.org/abs/1802.09477

[29] S. Pateria, B. Subagdja, A.-H. Tan, and C. Quek, "Hierarchical reinforcement learning: A comprehensive survey," *ACM Computing Surveys*, vol. 54, pp. 1–35, 06 2021.

[30] A. Levy, G. Konidaris, R. Platt, and K. Saenko, "Learning multi-level hierarchies with hindsight," 2019. [Online]. Available: https://arxiv.org/abs/1712.00948

[31] O. Nachum, S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," 2018. [Online]. Available: https://arxiv.org/abs/1805.08296

[32] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," 2016. [Online]. Available: https://arxiv.org/abs/1604.06057

[33] R. Fox, S. Krishnan, I. Stoica, and K. Goldberg, "Multi-level discovery of deep options," 2017. [Online]. Available: https://arxiv.org/abs/1703.08294

[34] A. Gupta, V. Kumar, C. Lynch, S. Levine, and K. Hausman, "Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning," 2019. [Online]. Available: https://arxiv.org/abs/1910.11956

[35] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1, pp. 181–211, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370299000521

[36] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Proceedings of the 6th International Conference on Neural Information Processing Systems*, ser. NIPS'92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, p. 271–278.

[37] Y. Jiang, S. Gu, K. Murphy, and C. Finn, "Language as an abstraction for hierarchical deep reinforcement learning," 2019. [Online]. Available: https://arxiv.org/abs/1906.07343

[38] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," 1999. [Online]. Available: https://arxiv.org/abs/cs/9905014

[39] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," 2016. [Online]. Available: https://arxiv.org/abs/1609.05140

[40] A. McGovern and A. G. Barto, "Automatic discovery of subgoals in reinforcement learning using diverse density," in *International Conference on Machine Learning*, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:1223826

[41] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor, "A deep hierarchical approach to lifelong learning in minecraft," 2016. [Online]. Available: https://arxiv.org/abs/1604.07255

[42] E. Ausonio, P. Bagnerini, and M. Ghio, "Drone swarms in fire suppression activities: A conceptual framework," *Drones*, vol. 5, no. 1, 2021. [Online]. Available: https://www.mdpi.com/2504-446X/5/1/17

[43] R. Hansen, "Estimating the amount of water required to extinguish wildfires under different conditions and in various fuel types," *International Journal of Wildland Fire*, vol. 21, pp. 525–536, 05 2012.

[44] R. N. Haksar and M. Schwager, "Distributed deep reinforcement learning for fighting forest fires with a network of aerial robots," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 1067–1074.

[45] R. S. Tanha, M. Hooshmand, and M. Afsharchi, "Uav-based firefighting by multi-agent reinforcement learning," in *2023 13th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2023, pp. 117–123.

[46] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," 2020. [Online]. Available: https://arxiv.org/abs/1706.02275

[47] G. Efstathiou, S. Gkantonas, A. Giusti, E. Mastorakos, C. M. Foale, and R. R. Foale, "Simulation of the December 2021 Marshall fire with a hybrid stochastic Lagrangian-cellular automata model," *Fire Safety Journal*, vol. 138, p. 103795, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0379711223000632

[48] "Copernicus Land Monitoring Service," https://land.copernicus.eu/en, accessed: 25.12.2023.

[49] "CLC+ Backbone," https://land.copernicus.eu/en/products/clc-backbone, accessed: 25.12.2023.

[50] J. Schulman. (2020, Jul) Approximating kl divergence. Blog post, accessed 2025-10-26. [Online]. Available: http://joschu.net/blog/kl-approx.html