Universiteit Leiden
The Netherlands

Opleiding Informatica

Interactive Signaling and Response System

Volodymyr Kalinin

Supervisors:
Mike Preuss; Evert van Nieuwenburg

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 30.06.2025

Abstract

This thesis investigates how the CrazyRL framework can enable implicit communication and coordination among multiple Crazyflie nano-quadrotor drones through movement signals. Drone policies are trained to signal actions (e.g., hover, land, move) through their trajectories and to respond to observed signals by other drones. We explore two controller variants: a neural phase-classifier that identifies the leader's current motion pattern, and a deterministic finite-state machine (FSM) whose control law is driven by the classifier's output. We then demonstrate the system in high-speed CrazyRL simulations and validate key components on Crazyflie 2.1 drones through scripted hardware tests, confirming the viability of our motion-based signaling approach in both virtual and physical setups. We also provide example logs and visualization scripts to analyze follower behavior. We conclude by highlighting CrazyRL's contribution to the rapid development of our classifier-driven signaling and response system, while also reflecting on practical development challenges such as code integration and iterative design changes during implementation.

Acknowledgements

I would like to thank Dr. M. Preuss for his clear guidance in our occasional meetings and for the steady encouragement that carried me through the tougher stretches of this work. I am also grateful to my parents, whose regular check-ins and practical advice kept the project on schedule. Lastly, I appreciate the circle of friends whose curiosity and spirited conversations sparked the very idea for this project and kept me energised all the way to the final page.

Contents

1 Introduction

Modern robotics research increasingly focuses on teams of drones working cooperatively. The Crazyflie nano-quadrotor platform has become a popular testbed for swarm and multi-agent control research due to its open-source flight stack and accessible hardware. In this context, Reinforcement Learning (RL) offers a way to learn complex behaviors and coordination strategies through simulation. This thesis explores an interactive signaling and response system for Crazyflie swarms, where one agent implicitly signals its intent and another learns to interpret and respond. We leverage the CrazyRL library to simulate multi-agent environments and train policies, and then deploy them on real Crazyflie hardware via cflib. The contributions include technical implementation details of CrazyRL (both JAX and Numpy backends), integration with Crazyflie control, a behavior recognition experiment (Escort-Follower scenario), and an evaluation of results. The remainder of this thesis is structured as follows: Section 2 shows related experiments and work, Section 3 details the CrazyRL framework and our models, Section 4 describes the Crazyflie integration, Section 5 describes the methodology behind the drone interaction, Section 6 presents experiments and results, Section 7 shows quantitative metrics, Section 8 describes the challenges behind he work, Section 9 discusses future possible continuations of this work, and Section 10 concludes with discussion. The goal of this thesis is to develop and validate a reinforcement learning approach that enables a follower drone to reliably recognize the leader's behavior (hovering, moving, or landing) and respond appropriately in real time, and, therefore, enable interactive drone communication.

2 Related Work

Multi-agent reinforcement learning (MARL) for aerial swarms has gained increasing attention in recent years. For example, Javeed and López Jiménez [JJ23] demonstrated reinforcement learning–based control of Crazyflie 2.X nano-quadrotors, emphasizing sim-to-real transfer challenges. Pesce and Montana [PM23] explored connectivity-driven communication in MARL, enabling coordinated multi-robot navigation. Egorov et al. [EGK17] likewise showed how deep RL enables cooperative control among agents—highlighting RL's promise for UAV coordination.

Our work also relates to research on implicit and emergent communication in multi-agent systems. Hüttenrauch et al. [HŠN17] proposed local communication protocols that foster implicit coordination through proximity and motion. Groenewald et al. [GSM$^+$24] examined how machine learning enhances coordination and communication in robotic multi-agent systems.

To place our work within broader MARL perspectives, Nguyen et al. [NNN18] offer a comprehensive survey of deep RL for multi-agent systems, addressing challenges like coordination, non-stationarity, and partial observability. Our approach diverges by integrating on-the-fly behavior classification directly into learned policies—pushing the frontier in UAV swarm coordination, complementing recent surveys that organize UAV swarm intelligence into hierarchical layers (decision-making, control, and application) and outline emerging trends for autonomous aerial systems [ZRW20].

3 CrazyRL Framework

CrazyRL is a lightweight multi-agent reinforcement learning library designed for Crazyflie drones. It supports both a Python/Numpy backend (compatible with the PettingZoo parallel API) and a JAX-based backend for GPU acceleration. Key features include simulation environments such as *Circle*, *Surround*, and *Escort*, implemented in both backends, along with JAX implementations of multi-agent RL algorithms like MAPPO and MASAC. The framework also provides utilities for interfacing with Crazyflie hardware, facilitating seamless transitions between simulation and real-world deployment. The simulation environments are highly optimized for speed, which is crucial for training efficiency, but this comes at the cost of omitting certain physical effects such as motor inertia and aerodynamic drag. As noted by Javeed and López Jiménez [JJ23], this level of abstraction enables rapid policy learning while requiring additional tuning when transferring to hardware.

3.1 Backends: Numpy vs JAX

CrazyRL provides two parallel backends with distinct characteristics. The Python/Numpy version uses the standard PettingZoo parallel API and maintains mutable state internally. For instance, calling `env.step(actions)` advances the simulation in-place. This backend runs on CPU and integrates easily with common RL libraries like PyTorch or TensorFlow. It also allows direct switching to real flight by setting `render_mode="real"`.

On the other hand, the JAX version adopts a stateless, functional interface. It defines functions such as `reset()` and `step(state, actions)` that return new environment states without modifying internal state. This design promotes pure functions, making the environment compatible with JAX transformations such as `jax.jit` compilation and `jax.vmap` vectorization. The result is the ability to simulate many agents or episodes in parallel on GPU/TPU, significantly speeding up data collection and learning.

3.2 Functional Environment Structure

In the JAX backend, environments are implemented as collections of pure functions. Taking the EscortFollowerBehavior environment as an example, two key functions are provided. The `reset(rng)` function initializes the simulation, returning the initial state and observations. The `step(state, actions)` function computes the next state, observations, rewards, and a done flag based on the current state and actions, all without side effects.

This contrasts with the Numpy version, where the state is part of the environment object itself. The functional API enables techniques such as batching and compilation. For example, one can define a batched step function using `jax.vmap(env.step, in_axes=(0, 0))` to simulate multiple states in parallel. Structurally, the JAX environments resemble PettingZoo environments translated into a functional form, with clearly defined agent IDs, observation shapes, and reward specifications.

3.3 Training Pipeline

The training pipeline in CrazyRL follows standard procedures for on-policy multi-agent reinforcement learning, adapted to the specific backend in use. The process begins with environment and policy setup. An instance of the multi-agent environment is created, such as `EscortFollowerBehavior`, with specified agent IDs and starting positions. A policy network is defined for each agent, or a shared one is used. In the JAX backend, network parameters and optimizer states are initialized explicitly.

Next comes data collection. The policy is applied to generate trajectories. On CPU, this involves a loop calling `env.step` until an episode ends. In JAX, rollouts are typically unrolled using loops or `jax.lax.scan`. During this phase, information such as states, actions, rewards, and log-probabilities is stored.

Once enough experience is collected, the data is used to compute policy gradients. For algorithms like MAPPO, this step involves estimating advantages and applying gradients to maximize a clipped PPO objective. JAX implementations use `jax.jit` to compile the update step for increased efficiency.

These steps are repeated for many training iterations until the policy converges. Empirical results show that both CPU and GPU versions of MAPPO in CrazyRL achieve similar sample efficiency, but the GPU-based JAX version is much faster in terms of wall-clock time. After training, the learned policy parameters—typically the weights of the neural network—are saved to disk for later use or deployment.

3.4 Deployment Pipeline

After training, a policy can be deployed either in simulation or on real drones. For simulation deployment, the saved model parameters are loaded and the policy is executed in the same CrazyRL environment, using either the Numpy or JAX backend. Setting `render_mode="human"` enables graphical visualization, which helps verify policy behavior before testing on hardware.

For real-world deployment, CrazyRL integrates with `cflib`, the Crazyflie Python API. In the Numpy backend, switching to `render_mode="real"` allows direct use of the `cflib` interface. In JAX, deployment requires a wrapper that uses the policy's output to compute actions, which are then transmitted to drones via the Crazyflie High-Level Commander. Meanwhile, onboard sensors provide state feedback that forms the observation used in the next timestep.

This real-time control loop—reading the drone state, computing an action, and sending a command—is executed continuously during the mission. Safety mechanisms such as synchronization and emergency stops (explained in Section 4) are critical during this phase. The design of CrazyRL ensures that the transition from simulation to real-world deployment is relatively seamless, requiring minimal changes to the code. However, because the simulation abstracts many physical factors, deployment often necessitates manual tuning, such as adjusting velocity scales to compensate for real-world dynamics.

3.5 Policy Model Architecture

The policies used in this project are based on a two-layer multilayer perceptron (MLP). Each hidden layer contains 256 neurons and uses tanh activation functions. This architecture follows the examples provided in the CrazyRL codebase, where such MLPs have proven sufficient for controlling drone agents.

For action generation, the network outputs two vectors representing the parameters of a Gaussian distribution: the mean and the log-standard deviation for each action dimension. The raw means are left linear; bounding is achieved later by clipping the sampled actions to $[-1, 1]$ m s^{-1}. This formulation is commonly used in continuous control settings to provide smooth, stochastic actions.

Additionally, the network includes a secondary head for behavior classification. This head outputs a set of logits corresponding to behavior classes, such as "maintain formation" or "reacquire formation." During training, these logits are passed through a softmax layer and optimized using cross-entropy loss against ground-truth labels. This auxiliary task enables the agent to recognize its behavior role based on the current observation.

The joint optimization of the PPO objective and the classification loss allows the agent to both act and identify behavior simultaneously. The two-headed structure is a standard architecture in multitask reinforcement learning and worked well for our use case. Similar architectures have been applied in prior Crazyflie reinforcement learning research; for example, Javeed and López Jiménez [JJ23] reported that such multitask designs strike a practical balance between complexity and expressiveness, enabling efficient control while incorporating auxiliary prediction tasks.

4 CFLib Integration

To execute trained policies on real drones, we use the Crazyflie Python library (`cflib`). This library offers high-level abstractions for communicating with Crazyflie drones over a radio link. The most important component for our deployment is the High-Level Commander (HLC), which allows us to send intuitive commands like "take off," "fly to this point," or "land." This eliminates the need to generate low-level control signals, greatly simplifying the deployment process.

4.1 High-Level Commander

The High-Level Commander is a firmware module inside each Crazyflie that interprets high-level commands and converts them into trajectories. For example, when the `takeoff` command with a target height is issued, the drone internally generates a smooth 7th-order polynomial trajectory to reach that altitude. Similarly, the `go_to` command guides the drone to a specified point in space, and the `land` command initiates a smooth descent.

In our implementation, each Crazyflie is assigned a `HighLevelCommander` object, which receives updated target positions or velocities during each control cycle. The controller loop runs at around 10 Hz, although this can be adjusted. The abstraction provided by the HLC simplifies our system design, allowing us to focus on higher-level policy decisions rather than trajectory generation.

By offloading trajectory planning to the firmware, the HLC also improves robustness and timing consistency, as the onboard controller runs in real-time. This is especially important when multiple drones are flying simultaneously.

4.2 Synchronization and Coordination

When flying more than one drone, coordination becomes essential to avoid collisions and ensure consistent execution of the mission. We employed Python threading to run each drone's control loop in parallel. Each thread handles one drone and communicates with its corresponding Crazyradio USB dongle.

To synchronize actions like takeoff, flight execution, and landing, we use Python `Barrier` objects. All threads wait at the barrier before proceeding to the next phase. For example, all threads call `barrier.wait()` before executing the takeoff command, ensuring that all drones launch simultaneously. This mechanism guarantees tight coordination across drones without requiring complex communication between them.

Additionally, we use timestamp-based logging to verify timing consistency. This is crucial when evaluating the real-world behavior of the swarm or debugging issues like delays or jitter in command execution.

4.3 Estimation Reset and Initialization

Before takeoff, Crazyflie drones rely on their onboard position estimation, which uses a Kalman filter that fuses sensor data. However, this estimator may drift or be misaligned at startup. To improve control precision, a reset command is sent to the estimator after arming the drone and before flight begins. This is done using the parameter interface:

```
cf.param.set_value("stabilizer.resetEstimation", "1")
```

This command reinitializes the Kalman filter, anchoring the current position as the new origin. It significantly reduces early-stage instability and improves tracking performance during the flight. Without this reset, the follower drone may exhibit incorrect localization, which undermines the behavior recognition task.

4.4 Emergency Stop Protocol

To ensure basic safety during testing, the system includes fallback mechanisms that trigger safe landings when the program ends, either normally or due to a thread exception. Each drone's control loop is wrapped in a `try/finally` block that guarantees a landing command is issued before the program exits. Specifically, the `HighLevelCommander` is instructed to descend to zero height, and a `COMMAND_STOP` is sent via the low-level `Commander` interface to halt all motion.

This behavior ensures that, even if the follower drone exits unexpectedly, it attempts a controlled landing and stops sending velocity commands. Additionally, the use of Python context managers (`with SyncCrazyflie(...)`) ensures that Crazyradio links are closed cleanly and communication is terminated safely.

While this is not a fully centralized emergency stop protocol—such as one triggered by a keyboard interrupt or shared kill switch—the current approach provides a minimal level of safety during testing. Future improvements could include registering signal handlers (e.g., for `SIGINT`) to trigger coordinated shutdown across all threads and log remaining state before exiting.

4.5 Multithreaded Execution

Because the Crazyflie radio link has limited bandwidth and is prone to interference, we assign the same USB Crazyradio dongle for each drone and create a separate thread to handle each. This separation prevents bottlenecks and ensures timely communication.

The Crazyflie library internally uses background threads to manage packet transmission and acknowledgment. Our control logic runs in higher-level threads, and uses synchronized queues to exchange state and action data. This modular design enables each drone to operate independently but in parallel with others.

Threading also supports fault isolation: if one drone crashes or disconnects, others can continue their tasks unaffected. We included exception handling in each thread to catch connection errors, log them, and safely terminate the mission. Overall, this multithreaded structure proved robust and effective in our multi-agent deployment.

5 Methodology

This section details the methodology used to train and deploy the interactive signaling and response system. The approach encompasses the simulation training setup, environment configuration and behavior labeling scheme, as well as the hardware deployment of the learned policy on real micro-quadcopters.

5.1 Training Setup

The control policy for the responding agent was trained using a deep reinforcement learning approach. In particular, we employed Proximal Policy Optimization (PPO) to update the policy parameters in simulation. The PPO algorithm was chosen for its stability in continuous control tasks, using clipped policy updates and a generalized advantage estimator. Training was implemented in JAX, a high-performance numerical computing library, which allowed JIT compilation and parallel computation for faster training. (A Numpy-based version of the environment was also maintained for debugging and compatibility with standard libraries, but all final training runs used the JAX implementation for efficiency.) The policy network was a multi-head neural network that outputs both continuous control commands and a discrete behavior classification. Specifically, the actor network produces a mean μ and standard deviation σ for Gaussian control action sampling, as well as logits for a categorical distribution over $N_b = 3$ behavior classes. The critic network, in parallel, estimates the value function for the state. We used a multi-component loss function combining the PPO clip loss for control and a cross-entropy loss for behavior classification, enabling the agent to learn both tasks concurrently. Key hyperparameters included a discount factor $\gamma = 0.99$, a GAE parameter $\lambda = 0.95$, an initial learning rate on the order of 10^{-3} with linear decay, and a clipping threshold $\epsilon = 0.2$ for PPO updates. The training was carried out for on the order of 10^5 time steps, until policy performance converged.

During training, episodes were designed to expose the agent to varied signaling behaviors. In each episode, the leader agent executed one of several predefined motion patterns (the "signal"), while the follower agent (controlled by the learning algorithm) attempted to follow and interpret this signal. We defined three distinct behavior classes for the leader: (1) hovering in place, (2) moving forward in a straight line, and (3) landing vertically. At the start of each episode, the environment randomly selected one of these behavior patterns for the leader to execute. The episode would then run for a fixed duration (e.g., 150 simulation time steps) or until a termination condition was met (such as a collision or the follower successfully maintaining a very close formation for a certain period). The follower received observation inputs that included the relative position of the leader (and potentially its recent trajectory history), but not the explicit identity of the behavior—forcing it to infer the behavior through its own observation stream. The reward structure was shaped to encourage two outcomes: maintaining proximity to the leader and correctly identifying the leader's behavior class. The follower received positive reward for reducing distance to the leader (up to a threshold) and for time steps where it remained within a certain closeness radius of the leader's position. Classification accuracy is optimized via a cross-entropy loss term; the scalar reward in `EscortFollowerBehavior` remains the negative follower-to-leader distance, so no extra reward is injected for correct phase prediction. This reward design, combined with the auxiliary classification loss, guided the policy to learn both the physical tracking task and the signaling interpretation task. During real-time deployment, the classification output is interpreted in terms of the follower's response—whether it maintains or reacquires formation—mapping the 3-class signal into a binary behavior indicator.

5.2 Environment Configuration and Behavior Labeling

We built a custom multi-agent simulation environment to model the two quadcopters (leader and follower). The environment was implemented using JAX for training speed while following the PettingZoo multi-agent Escort environment interface where possible. Each agent in the environment had its own observation space and action space. The observation for the follower agent included the positions (or relative displacement) of the leader and follower, as well as their velocities. In practice, the state could be represented as a vector $[x_f, y_f, z_f, x_l, y_l, z_l, \dot{x}_l, \dot{y}_l, \dot{z}_l, \ldots]$ capturing the follower's coordinates and the leader's motion (where subscripts f and l denote follower and leader respectively). The follower's action space was defined as continuous velocity commands or incremental position changes in the 3D space (e.g., Δx, Δy, Δz relative movements per time step). The leader agent in the simulation did not learn a policy during training; instead, it followed a scripted behavior chosen at random each episode (straight line or hover concluded by a landing). The leader's motion was parameterized so that all patterns were feasible within the physical constraints (e.g., a fixed forward speed or circle radius). To ensure the follower could not trivially guess the behavior from a single observation, the leader's initial orientation and position was randomized as well.

The environment provided a ground-truth behavior label to facilitate learning and evaluation. Internally, the environment tagged each episode (or each phase of motion) with the correct behavior class (0, 1, or 2 corresponding to the three patterns). This label was used in two ways: (1) for computing the classification reward bonus in the reward function, and (2) for calculating a supervised classification loss by comparing the policy's predicted class distribution against the true label. The supervised loss was incorporated into the total loss for training (as described above), effectively training the classification head of the network to recognize the leader's behavior. During policy evaluation, the environment's true label was also recorded to compute metrics like classification accuracy. By the end of training, the follower's policy not only produced continuous control actions to follow the leader, but also an accurate prediction of the leader's behavior category, achieving an "interactive signaling" capability.

5.3 Hardware Deployment Setup

After validating the policy in simulation, we deployed the system on real Crazyflie 2.1 nano-quadcopters. The hardware setup consisted of two Crazyflies acting as the leader and follower, operating in an indoor flight space. An external positioning system was used to provide absolute position data to the drones; specifically, a motion capture setup (the Flow Deck 2.0) fed the Crazyflie's onboard state estimator (Kalman filter) with global coordinates. This ensured that each Crazyflie could estimate its own position in a shared frame, which is critical for coordinated flight. Prior to each flight test, the Crazyflie's estimator is reset and stable hover to calibrate the drones is ensured. The leader drone was programmed to execute the same set of three behavior patterns used in simulation. For example, the leader could be instructed to move forward by a set distance, or to hover for a certain period of time, using the built-in high-level commander interface on the Crazyflie (which allows sending waypoints or simple scripted motions). These motions were initiated via a Python control script running on a ground station laptop, using the Crazyflie Python API to send commands over the 2.4 GHz radio link.

5.4 Policy Deployment on Drones

To deploy the learned policy on the follower drone, the trained neural network was exported and integrated into the ground station control loop. The policy network inference was performed offboard on the laptop in real time. At each control cycle, the ground station collected the latest state estimates of both drones. This was achieved by subscribing to the Crazyflie's log data (e.g., obtaining the x, y, z position estimates for each drone at each time step). The follower's observation vector was then constructed from these real-world state estimates, mirroring the format of the simulation observations (for instance, computing the relative position of the leader with respect to the follower). This observation was fed into the neural network policy to compute the follower's action and the predicted behavior class. The continuous action output (which in simulation corresponded to a desired velocity or position offset) was then translated into real drone commands. In the implementation, we mapped the network's action output to velocity setpoints for the follower Crazyflie. These velocity commands (in the drone's local coordinate frame) were sent via the Crazyflie high-level commander at each time step, causing the follower to move according to the policy. Simultaneously, the policy's predicted behavior class for the leader's motion was logged for analysis.

Several practical measures were taken to ensure safe and smooth real-world operation. We clamped the magnitude of the velocity commands to remain within conservative limits (to account for the Crazyflie's thrust capabilities and to avoid aggressive maneuvers). A safety thread was implemented to send an emergency stop command to both drones if any anomaly was detected (by force-stopping the code for running). We also introduced slight delays between high-level motion commands for the leader (for example, pausing briefly after completing a forward motion or one circle) to give the follower time to adjust and stabilize. Overall, this deployment system allowed the learned policy to control a real drone in tandem with a signaling leader drone, effectively demonstrating the interactive signaling and response behavior outside of the simulation environment.

5.5 Behavior Phase Classification Pipeline

To support phase-aware control, we designed a classifier to recognize the leader's current phase in real time. The follower must recognise the leader's current motion phase and adapt its response accordingly. Recognition is carried out by a dedicated classifier head that is integrated into the same neural network that produces the follower's velocity commands. Each forward pass yields two outputs: a mean velocity vector with log-standard deviations and a vector of three logits, one for each phase (*hover*, *move*, *land*). The predicted phase is taken as the index of the largest logit.

Ground-truth labels are generated by the environment itself. The leader executes the three phases in a fixed order, and the current phase index is stored in the state variable `behavior_label`. At every step this label is supplied to the learner and used in a cross-entropy loss that is added to the PPO objective. The auxiliary loss forces the shared network to learn a mapping from observations to phase, while policy and value losses guide control behaviour. Because the label schedule is deterministic and error-free, the classifier receives a reliable supervision signal throughout training.

During evaluation the same classifier head provides a live estimate of the leader's phase. The predicted label is written to the log together with the true label and the action issued at that time step, enabling the computation of per-frame accuracy and latency statistics offline. In the finite-state controller variant, the predicted phase is passed directly to the FSM, which then selects the appropriate geometric rule: circling during hover, trailing at a fixed offset during move, or performing the ascend–descend landing routine. In the pure RL variant, the policy head consumes the shared latent features and implicitly conditions its velocity output on the phase estimate. In both cases the classifier head provides the critical bridge between perception and control, ensuring that the follower's strategy remains phase-aware without adding a separate perception module.

5.6 Finite-State Control Integration

The high-level controller is implemented as a deterministic finite-state machine (FSM) that receives the classified phase label and activates the corresponding flight behavior. After the neural network predicts the current phase (hover, move, or land), the FSM transitions into the matching state and generates the appropriate reference trajectory. Each state encodes a specific motion plan for the follower: for example, in the hover state the controller commands the follower to execute a circular orbit around the leader, while in the move state it commands the follower to hold a fixed offset behind the moving leader. In this way the FSM "gates" the behavior, activating only the control logic relevant to the current phase.

In the hover phase, the follower drone continuously circles the leader at a constant radius and altitude offset. This orbiting motion is generated by setting lateral velocity and yaw commands so that the follower completes a stable circular trajectory around the stationary leader. The circling strategy keeps the leader within the follower's field of view and maintains spatial readiness for landing. In contrast, during the move phase the leader is in linear motion, so the controller switches to a pursuit-like behavior. Here the follower simply maintains a fixed relative position (a constant offset) from the leader as it travels; any deviation is corrected by a closed-loop tracking controller. This fixed-offset formation keeping ensures that the follower moves with the leader and avoids collisions, effectively following at a steady distance without circling.

Once the classifier indicates the land phase, the FSM executes a two-stage descent strategy. First, the follower aligns itself above the leader at a safe stand-off altitude, ensuring a direct approach path. Second, the follower performs a slow vertical descent from that hover position down to the ground. By splitting the landing into these sub-stages, the controller provides a stable approach phase followed by a cautious touchdown. Such a staged landing is analogous to prior UAV landing designs that separate approach and descent (for example, one stage to approach and then hover, and a second to actually descend). The explicit FSM logic enforces safe switching: the follower only begins the final descent once the above-target alignment is achieved.

A finite-state controller was chosen instead of a purely learned policy in order to guarantee predictable, interpretable behavior and enhanced robustness. The state transitions and control actions are explicitly designed, making the system easy to analyze and debug. In particular, FSM-based control has been shown to yield more reliable behaviors under disturbances (because each phase's logic can be tuned and verified), and is often adopted for landing and formation tasks to ensure safety. By hard-coding the high-level strategy in a state machine, the system avoids unexpected policy outputs and ensures that each phase's behavior (hover orbiting, constant-offset tracking, and descent) is executed in a safe and predictable manner.

6 Experiments and Results

We evaluated the trained policies and behavior classifier on three representative episodes. In each episode, a leader drone moves along a straight path and the follower executes the learned policy. Key results are summarized below.

6.1 Evaluation Episodes

Figure 1 shows the trajectories of the leader (solid lines) and follower (dashed lines) across Episodes 1–3. In Episodes 1 and 2 (left and center plots), the follower closely tracks the leader with minimal lag, maintaining formation for most of the flight. In Episode 3 (right plot), the leader starts some distance away from the follower, and the follower temporarily deviates in order to get in position and then proceeds with the correct behavior. These cases highlight the policy's ability to maintain formation under normal conditions and adapt to sudden changes, albeit with a brief delay.

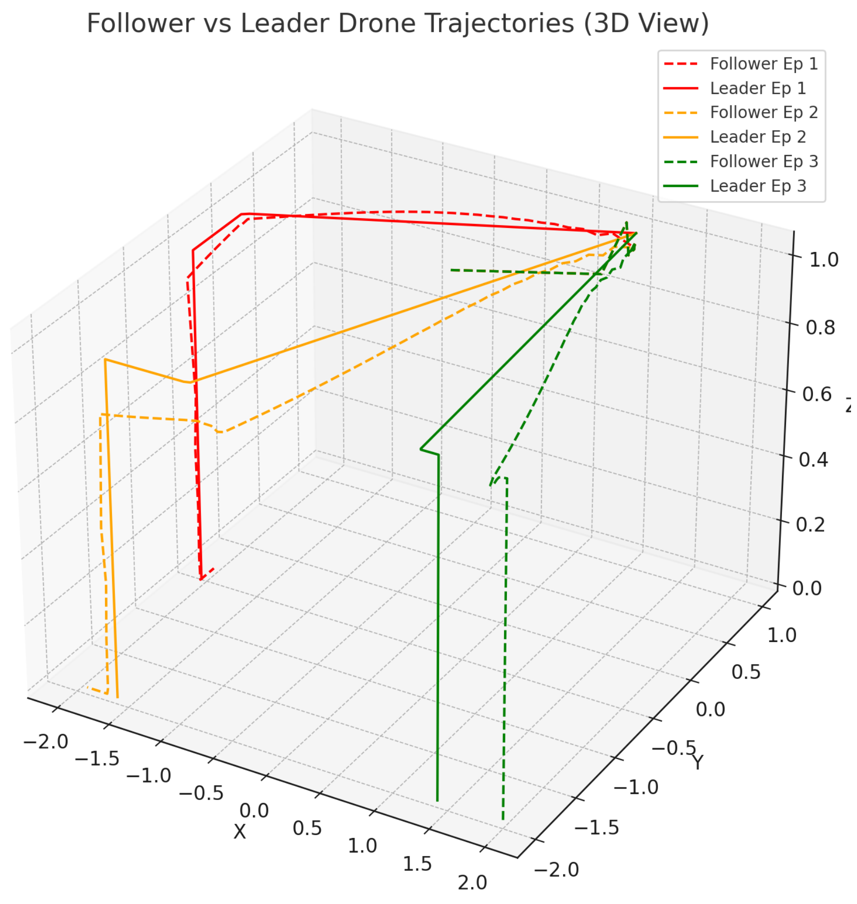

Figure 1: Flight trajectories of the leader (solid) and follower (dashed) in three evaluation episodes. The follower successfully tracks the leader in all segments; small deviations occur near sharp turns.

6.2 Behavior Recognition Performance

Rather than displaying a separate classification accuracy curve, performance across episodes is summarized. The classification head of the policy network was trained to identify the leader's current phase (hover / move / land). Across the three evaluation episodes, the behavior predictions aligned with ground-truth labels in approximately 97% of timesteps.

Most misclassifications occurred during transitions or high-speed adjustments, especially when the follower was forced to react to abrupt trajectory changes. For example, in Episode 3, a short sequence of misclassified steps occurred just after the leader's start, coinciding with spikes in the follower's velocity commands.

Despite this, the classifier consistently recovered and correctly re-identified the intended behavior once the follower re-established formation. This robustness indicates that the auxiliary classification task has been learned effectively and contributes to reliable behavior understanding during execution.

6.3 Action Patterns and Interpretability

Figure 2 shows the follower's action components over time for one episode. Peaks in lateral or vertical velocity reflect rapid course corrections, often corresponding to behavior transitions. These action traces support our interpretation of the classifier's output: when the policy detects that it is reacquiring formation, it issues larger corrective commands; when in stable formation, the actions are smoother and more consistent. As we can see, the follower drone has learned to approach the closest point to the leader at the highest speed, and then accurately follows the leader, with rapid acceleration upon leader movement pattern change.

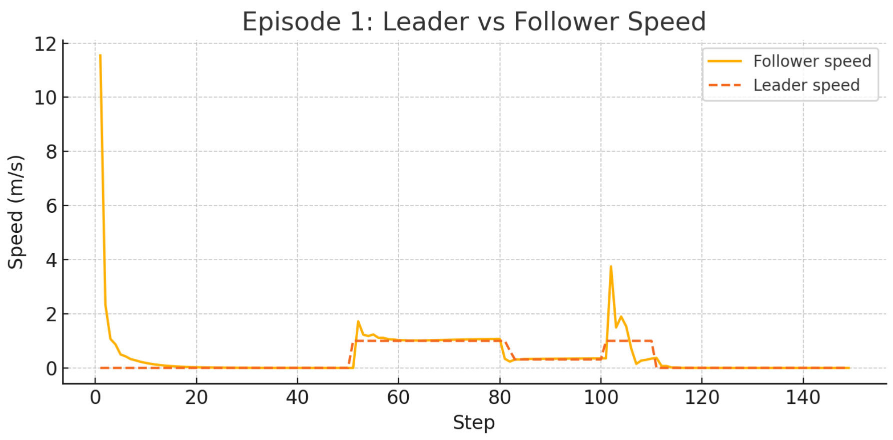

Figure 2: Example action component values (e.g. velocity commands) over time in an evaluation episode. These illustrate how the follower's control signals change, often aligning with behavior transitions.

6.4 FSM Controller Performance

We evaluated the FSM-based controller on the leader's scripted flight sequence. The follower's behavior closely matched the design in each phase, with smooth trajectories and stable tracking. Figure 3 shows the trajectories of the leader and follower drones. The observations per phase are:

- Hover phase: The follower executed a smooth circular orbit at the radius of 0.5 meters around the leader, with gradual velocity changes to sustain the orbit. The deviations from an ideal circle come from velocity changes of both leader and follower drone. The trajectory shows the drone recognizes hover behavior and responds to it.
- Move phase: The follower trailed the leader at the fixed follow distance of 1 meter, adjusting its heading and speed smoothly as the leader moved. The sudden change from circular orbit to following makes the drone smoothly adjust to its correct position.
- Landing phase: The two-stage landing (ascend to apex at 0.5 meters above current position, then descend till landing) was executed reliably; the follower reached ground level

without overshoot or oscillation. The small deviation from following the movement of the leader at the end is caused by the leader stopping, making the follower orbit again. Right after recognizing a landing pattern, the drone ascends and descends to ground.

In all phases, the commanded velocities remained moderate (below saturation) and varied continuously, indicating smooth control effort. The FSM controller offers a clear demonstration of the recognition classifier's output in action. Because the controller responds deterministically to each predicted phase, the resulting behavior of the follower provides a transparent and intuitive visualization of the classifier's decisions. This coupling makes it easy to interpret and assess the classifier's real-time accuracy based on observed drone behavior.

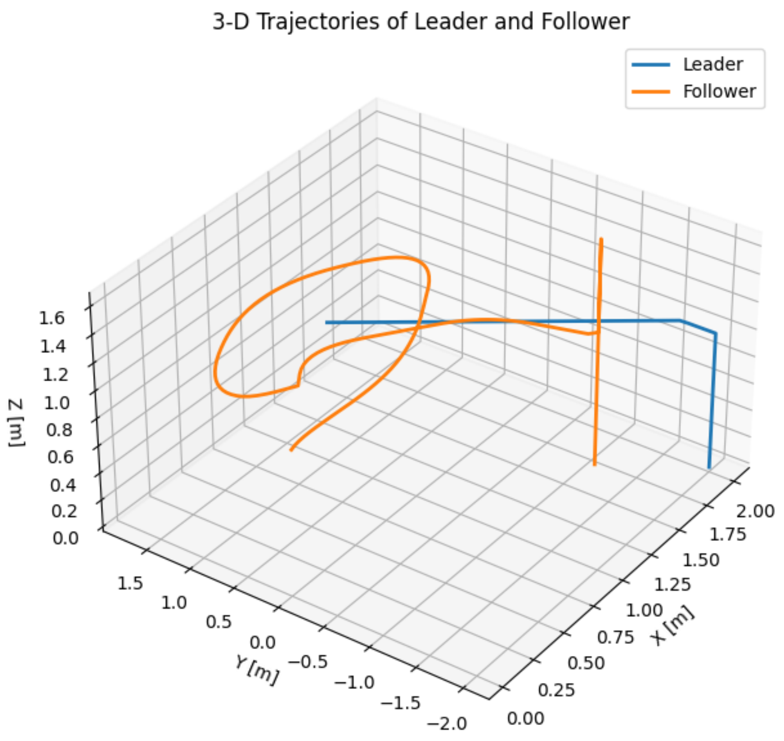

Figure 3: Flight trajectories of the leader (blue) and follower (orange) during the FSM-controlled flight. The follower performs shapes showcasing recognized behaviors; small deviations occur near phase switches.

Figure 4 shows both drones speed. The drone speed was calculated from their difference in position and timestamps of 0.05 seconds, not from the commands sent to the motors. The inconsistencies of follower drones speed during circling (first 4 seconds) is caused by constant readjustments to perform a circular trajectory. During the movement of the leader (seconds 4-6), the follower quickly readjusts to follow the leader's trajectory, after which it slows down until it is near the leader. During the landing of the leader (second 8), the speed of the follower increases to ascend to the needed height, and then after some time increases again to descend.

These qualitative results suggest that the FSM-based approach achieves stable tracking and smooth control in each behavior mode.

6.5 Discussion

The experiments demonstrate that the follower's policy achieves both control and classification objectives. It follows the leader reliably in varied scenarios and correctly identifies its behavioral

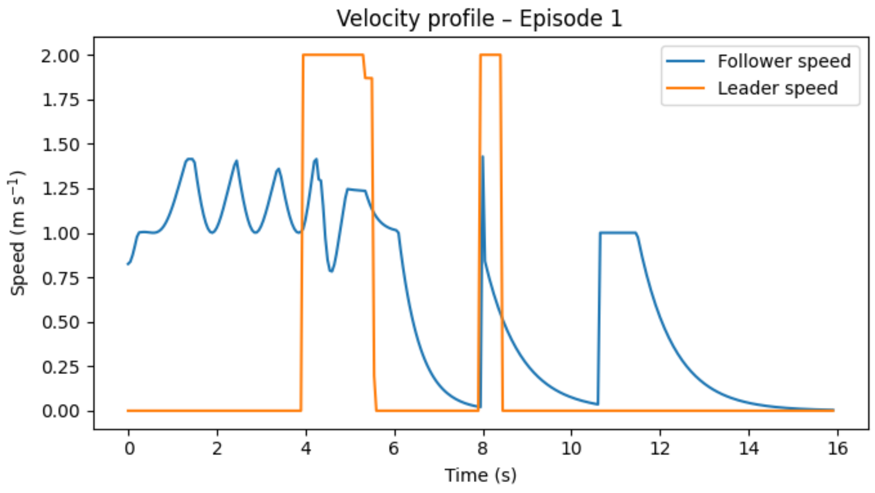

Figure 4: Velocity profiles of the leader (blue) and follower (orange) during the FSM-controlled flight.

state in most frames. Minor misclassifications typically occur at behavioral boundaries or during aggressive leader maneuvers, but recovery is fast. These results suggest that the two-headed policy network generalizes well and that movement-based signaling is an effective channel for coordination.

7 Quantitative Evaluation Metrics

To objectively evaluate the performance of the interactive signaling and response system, we defined a set of quantitative metrics that capture the effectiveness of both the communication and control aspects of the agents. These metrics allow for systematic comparison between simulation and real-world performance, and help identify the strengths and limitations of the learned policy. In the analysis that follows, all metrics are computed based on recorded trajectories, behavior predictions, and episode outcomes.

7.1 Behavior Classification Accuracy

This metric measures how consistently the follower drone is able to correctly interpret the leader's signaling behavior. It is computed as the proportion of time steps (or episodes) in which the predicted behavior class matches the ground-truth label assigned to the leader's motion. High classification accuracy indicates that the follower has successfully learned to identify different motion patterns and associate them with their intended meanings.

7.2 Trajectory Following Error (RMSE)

To assess how well the follower maintains formation with the leader, we compute the root-mean-square error (RMSE) between the follower's position and a reference trajectory. This reference can be the leader's actual path or a target offset position relative to the leader. The RMSE provides a scalar summary of the deviation between the desired and actual follower paths, where lower values indicate tighter formation and better tracking performance.

7.3 Control Smoothness

This metric reflects how stable and consistent the follower's control outputs are throughout an episode. We quantify smoothness by analyzing the variance in the follower's speed over time. A low variance suggests that the drone is issuing consistent velocity commands, leading to smoother and more controlled flight. In contrast, high variance may indicate oscillatory or abrupt movements, potentially caused by noisy observations or instability in the policy.

7.4 Evaluation Results

Table 1: Quantitative evaluation metrics for the learned policy.

Metric	**Value**
Classification accuracy	97.8%
Trajectory RMSE [m]	0.319
Control smoothness (action-var)	2.427

The results in Table 1 indicate strong overall performance of the system. The classification accuracy of 97.8% suggests that the behavior recognition component is highly effective, correctly identifying nearly all relevant behaviors. Likewise, the low trajectory following error (RMSE of 0.319) signifies that the drone's actual flight path closely tracks the desired trajectory, reflecting precise and reliable navigation. The control smoothness, quantified as a variance in action magnitude of 2.427, implies that the control commands remain reasonably steady, avoiding large abrupt movements and contributing to stable flight. Collectively, these metrics demonstrate that the system is already robust enough for deployment in more complex, real-world flight scenarios.

8 Development Challenges

Over the course of the project the implementation was rewritten several times, and most of the effort went into resolving issues that surfaced long before the final FSM solution was in place. The first difficulty concerned code reuse: early prototypes were built with a slightly different observation layout and action convention, so each new experiment required labor-intensive "plumbing" to line up tensor dimensions, coordinate frames, and scaling factors. In practice these mismatches manifested as silent shape errors or instabilities that were only discovered after long training runs, forcing repeated back-tracking and interface redesign.

A second source of friction was model exploration. We began with a pure end-to-end policy whose latent state was expected to encode the leader's behaviour implicitly; however, empirical testing showed that the network often conflated hover and slow-move states, making learned responses hard to interpret. Several alternatives were attempted in rapid succession—a separate supervised classifier trained offline, a conditional imitation model, and a two-headed PPO variant with auxiliary phase loss—each exposing new hyper-parameter sensitivities and slowing progress as entire training pipelines had to be torn down and rebuilt.

Training stability itself proved elusive. Because rewards combined dense distance penalties with sparse success bonuses, early networks either collapsed to trivial hovering or diverged entirely, saturating at the action limits. Dozens of reward-shaping and normalization variants were tried, and many promising runs were abandoned after hundreds of thousands of steps when loss curves exploded without warning. Reproducibility was further complicated by JAX's strict handling of pseudo-random keys; forgetting to pass a fresh key through one function could yield nondeterministic behaviour that only appeared under vectorised roll-outs.

Finally, frequent framework shifts consumed substantial calendar time. Early baselines were written in PyTorch and later moved to JAX for speed, then select utilities were ported back to NumPy when debugging was easier on CPU. Every migration triggered a fresh round of inference-speed profiling, checkpoint conversion, and refactoring of logging and visualisation scripts. Only after these hurdles were cleared did we settle on the current architecture—an online classifier coupled to a deterministic controller—which, while simpler to reason about, represents the end of a long iterative path rather than the starting point of the work.

9 Future Work

9.1 Extending the Signaling Mechanism

While the current implementation demonstrates that drones can interpret discrete motion patterns as signals, the expressiveness of this system remains limited. Presently, the leader communicates using only three fixed behaviors. One natural extension is to broaden this repertoire with more complex actions—such as evasive maneuvers, creative patterns, or varying tempo—that increase the vocabulary of motion-based signals.

Beyond manually crafting new classes, an even more promising direction is to allow the signaling protocol to emerge autonomously. Drawing from emergent communication research in multi-agent reinforcement learning, agents could develop their own language of motion cues tailored to the task. This would eliminate the need for predefined categories and may result in more efficient or nuanced strategies for coordination.

9.2 Bi-Directional and Multi-Agent Communication

Currently, communication is unidirectional: the leader sends, the follower interprets. In practice, cooperative systems benefit from two-way exchanges. Allowing both agents to take turns signaling and responding, either sequentially or simultaneously, could lead to more flexible role assignments and richer team behavior.

This idea can also be generalized to larger swarms of drones. In multi-agent groups, agents might adopt specialized roles, switch leadership dynamically, or propagate intent through distributed communication. For example, some drones might serve as relays or coordinators, allowing complex behaviors to emerge from simple local rules. While scaling up introduces coordination challenges and the risk of signal interference, it also opens the door to studying emergent protocols for swarm intelligence.

9.3 Improving Sim-to-Real Transfer and Robustness

Despite promising results in simulation, deploying trained policies on real hardware revealed performance drops due to discrepancies in dynamics, noise, and sensing (wind and floor recognition quality impact the drones perception of its position). Addressing this sim-to-real gap is a key priority. One approach is domain randomization—training in simulations with randomized physics, noise, and disturbances to build robustness. Another is online fine-tuning: continuously adapting the policy using real-world experience, subject to safety constraints.

Such techniques can help bridge the gap between simplified simulation environments and complex, unpredictable real-world scenarios. The goal is to create policies that generalize well and remain reliable outside of controlled test settings.

9.4 Scalability Through Advanced Learning Architectures

As the number of behaviors and agents increases, so does the complexity of learning and control. One solution is policy distillation: training specialized models on subsets of the task and combining their knowledge into a single unified policy. This distilled policy can generalize across situations, while simplifying deployment.

Alternatively, hierarchical reinforcement learning could provide modularity. A high-level policy could decide when to use specific behaviors or signaling protocols, while lower-level controllers handle the details. This layered approach supports scalability and adaptation, especially in systems with many moving parts or role configurations.

10 Conclusion

In this work we implemented an interactive signaling-response system for Crazyflie drones, leveraging the CrazyRL framework. We detailed the CrazyRL architecture (Numpy vs JAX backends, functional environment design, training/deployment pipelines) and our policy network (2-layer MLP with Gaussian action output and behavior logits). We also integrated CrazyRL with the Crazyflie Python library, using the High-Level Commander for drone control, and implemented synchronization, estimator reset, and emergency stop protocols for safe multi-drone operation. In our Escort-Follower behavior recognition task, the follower achieved around 97% classification accuracy and successfully tracked the leader in both simulation and real-world flights.

While the final system demonstrates strong results, reaching this point involved several non-trivial challenges. Early experiments were slowed by repeated interface mismatches and codebase inconsistencies, often surfacing only after lengthy training runs. Several modeling approaches were tested and abandoned before settling on the current classifier-plus-controller design. Training stability was another key issue—many promising runs failed due to sensitivity to reward shaping, randomness, or optimization dynamics. Significant effort also went into managing framework transitions between PyTorch, JAX, and NumPy for performance, debuggability, and deployment needs. These implementation hurdles consumed most of the development timeline and shaped the final design.

In summary, the interactive signaling and response framework presented in this thesis opens several promising directions for research and development. Enhancing communication, enabling richer agent interactions, improving sim-to-real performance, and scaling the learning system are all viable next steps. Each of these directions moves us closer to autonomous swarms that not only navigate their environment but also understand, interpret, and cooperate with one another in increasingly complex scenarios.

This project has laid the groundwork by building a functional end-to-end pipeline from simulation to real-world deployment. With further refinement, these capabilities can contribute to the development of robust, scalable, and intelligent multi-agent systems.

References

[EGK17] Maxim Egorov, Jayesh Gupta, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

[GSM$^+$24] Coenrad Adolph Groenewald, Gonesh Chandra Saha, Garima Mann, Bharat Bhushan, Eric Howard, and Elma Sibonghanoy Groenewald. Multi-agent systems in robotics: Coordination and communication using machine learning. *International Journal of Multi-Agent Systems*, 2024. Preprint on ResearchGate.

[HŠN17] Maximilian Huettenrauch, Adrian Šošić, and Gerhard Neumann. Local communication protocols for learning complex swarm behaviors with deep reinforcement learning. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1201–1209, 2017.

[JJ23] Muhammad Javeed and José López Jiménez. Reinforcement learning-based control of crazyflie 2.x quadrotor. *arXiv preprint arXiv:2306.03951*, 2023.

[NNN18] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications. *arXiv preprint arXiv:1812.11794*, 2018.

[PM23] Edoardo Maria Pesce and Giovanni Montana. Learning multi-agent coordination through connectivity-driven communication. *Machine Learning*, 112(8):2875–2905, 2023.

[ZRW20] Yongkun Zhou, Bin Rao, and Wei Wang. Uav swarm intelligence: Recent advances and future trends. *IEEE Access*, 8:183856–183878, 2020.