Alma Mater Studiorum - University of Bologna

SCHOOL OF ENGINEERING

DEPARTMENT OF
ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING
"Guglielmo Marconi"
DEI

**MASTER DEGREE IN
AUTOMATION ENGINEERING**

MASTER THESIS
in
Cyber-physical Systems Programming

Learning-Based Control for Nano-Drones Flight: From Simulation to Reality

Candidate:
Mattia Guazzaloca

Supervisor:
Prof. Andrea Acquaviva

Co-supervisor:
Sebastiano Mengozzi, MSc

Academic Year 2024/2025
Session II

Abstract

Recent advances in deep reinforcement learning have opened new possibilities for autonomous aerial robots, particularly nano-drones, which are among the most agile yet difficult platforms to control. Their small size and limited hardware resources make classical control design challenging, while learning-based approaches promise greater adaptability but often face difficulties when transferring from simulation to reality.

This thesis explores reinforcement learning for low-level motor control on the Crazyflie 2.1 nano-quadrotor. A complete experimental pipeline was developed, combining massively parallel simulation with the Proximal Policy Optimization (PPO) algorithm, multi-seed evaluation, and deployment on the real platform using only onboard sensing.

The learned controller successfully transferred to real flight, achieving stable hovering and accurate trajectory tracking. Performance was comparable to state-of-the-art solutions, confirming that key design elements such as rotor-delay modeling and action history are critical for bridging the simulation-to-reality gap. An ablation study further demonstrated that removing these elements severely compromises stability, underscoring their importance for reliable deployment.

Overall, the results show that reinforcement learning can be applied effectively to nano-drones for direct motor control, providing a reproducible pipeline that bridges simulation and reality. At the same time, the study highlights current limitations in training stability and robustness, and points toward future research aimed at improving efficiency and extending learned controllers to more complex flight scenarios.

Contents

Introduction

Over the past decade, advances in multiple areas of science and engineering have radically transformed the possibilities for intelligent machines. Progress in materials and battery technology has made it possible to design mobile platforms that are lighter, stronger, and longer lasting than ever before. Improvements in sensing hardware have provided robots with increasingly accurate and diverse ways of perceiving their environment, from high-resolution cameras and inertial units to compact depth and range finders. At the same time, embedded processors have grown in efficiency, enabling substantial onboard computation even under tight power constraints. Parallel to these hardware advances, artificial intelligence has undergone its own revolution: learning-based methods now allow machines to extract structure and patterns from raw data, adapt through experience, and perform tasks that were once exclusive domain of human intelligence (e.g., writing, translation, and conversation). Taken together, these trends are pushing robotics out of the laboratory and into everyday life, where autonomous systems are expected to support, extend, and sometimes replace human activity in domains such as manufacturing, logistics and healthcare. Yet for all this progress, a central challenge remains unsolved: how to design algorithms that allow machines not only to act, but to do so reliably and adaptively in the unpredictable environments of the real world.

Within this landscape, aerial robotics—and in particular quadrotors—stand out as one of the most challenging and promising fields of application. Thanks to their mechanical design, they can accelerate rapidly, change direction almost instantaneously [1], and navigate confined or cluttered spaces inaccessible to other vehicles. This agility makes them suitable for tasks such as search and rescue, inspection, mapping, and entertainment [2]. Yet the very properties that make quadrotors versatile also make them exceptionally difficult to control. They are inherently unstable, highly nonlinear systems [3], must constantly counteract gravity to stay aloft, while their lightweight structure limits onboard computational power and their small batteries restrict flight time. These constraints make quadrotors demanding benchmark platforms for testing robotic autonomy and advanced control algorithms under extreme dynamic conditions.

Classical approaches to quadrotor autonomy are typically organized into modular pipelines that divide the task of flight into distinct components [4]. The perception block processes data from onboard sensors to estimate the state of the vehicle and its envi-

ronment. The planning block then uses this information to generate feasible trajectories that satisfy mission objectives while avoiding obstacles and respecting the physical limits of the platform. Finally, the control block ensures that the vehicle follows the desired trajectory by adjusting motor commands in real time. This separation of responsibilities has clear advantages: each module can be designed, analyzed, and tuned independently, and decades of research in control theory provide guarantees of stability and performance when the underlying models are accurate.

However, this modular structure also has important limitations. Each stage in the pipeline introduces its own latency, and while these delays may be acceptable for tasks such as structured indoor flight or slow surveying, they become critical in highly dynamic environments. For instance, a quadrotor navigating through a dense forest must react to unexpected obstacles and wind gusts in fractions of a second. In such settings, the cumulative delay from perception, planning, and control may prevent timely responses, leading to collisions or loss of stability. In addition, modular designs often rely on simplified mathematical models to keep each block tractable, but these assumptions may break down in complex or unstructured environments, reducing robustness and adaptability. These drawbacks have motivated a growing body of research aimed at replacing parts of the traditional pipeline with unified approaches that reduce latency and relax modeling assumptions.

Deep Reinforcement Learning (DRL) offers a radically different paradigm by training policies that learn directly from interaction with the environment how to map sensor observations to motor commands. Instead of relying on separate perception, planning, and control modules, a single neural policy can be optimized end-to-end, reducing latency and automatically adapting to the full nonlinear dynamics of the system. Recent studies have shown that this approach can achieve remarkable results, even on real hardware, with autonomous quadrotors demonstrating agile maneuvers that rival or surpass human pilots in competitive racing scenarios [5]. Despite these successes, reliable deployment outside of carefully designed experiments remains challenging. Policies trained in simulation often fail to generalize to real-world conditions due to discrepancies in dynamics, delays, or sensor noise—a difficulty widely known as the Sim2Real gap [6]. Furthermore, most of the successful demonstrations have relied on larger aerial platforms equipped with powerful onboard computers or companion processors. Extending these methods to resource-constrained nano-drones, which operate with limited computation, sensing, and flight time, remains an open challenge and a key motivation for this thesis.

This thesis contributes to this line of research by investigating the transfer of end-to-end reinforcement learning controllers from simulation to real hardware, focusing on the Crazyflie 2.1 nano-quadrotor. The objective is to train a policy that maps proprioceptive observations directly to motor commands, bypassing traditional control layers. Building upon the foundational work of Eschmann et al. in Learning to Fly in Seconds [7], this

study explores how such end-to-end controllers can be trained, deployed, and validated on a resource-constrained aerial platform, with the broader goal of advancing robust and accessible methods for learning-based control.

The remainder of this thesis is organized as follows. The first chapter introduces the theoretical foundations of quadrotor dynamics and reinforcement learning, providing the background needed to understand the proposed approach. This is followed by a review of prior work on modular and learning-based control strategies for quadrotors, highlighting the gradual shift toward end-to-end low-level control. The methodology chapter presents the proposed framework in detail, from task definition and simulation environment to training setup and deployment strategy. The subsequent chapter reports the experimental results obtained both in simulation and on real hardware. Finally, the thesis concludes with a summary of the main findings and an outlook on possible directions for future research.

Chapter 1

Background

This chapter provides the necessary background for the work presented in this thesis. Section 1.1 presents the fundamentals of reinforcement learning, explaining its formulation as Markov decision processes and the notions of policies and value functions. Section 1.2 then introduces the basics of deep learning, describing how neural networks can serve as powerful function approximators, while Section 1.3 shows how their integration with reinforcement learning has led to deep reinforcement learning, with a focus on actor–critic methods such as Proximal Policy Optimization and Twin Delayed Deep Deterministic Policy Gradient. Section 1.4 reviews the principles of drone dynamics and control, discussing both classical and learning-based approaches, and finally Section 1.5 provides an overview of simulation environments for reinforcement learning, which are essential for large-scale training and sim-to-real transfer.

1.1 Reinforcement Learning

In the broader field of machine learning, two classical paradigms are supervised and unsupervised learning. In supervised learning, a model is trained using labeled examples, where the correct input–output pairs are explicitly provided. Unsupervised learning, on the other hand, aims to discover patterns or hidden structures in unlabeled data without relying on explicit feedback. Reinforcement Learning (RL) differs from both of these: instead of being presented with a fixed dataset, an agent must learn directly from its own experience by interacting with an environment. Feedback comes in the form of evaluative signals, or rewards, that indicate how desirable the outcomes of its actions were.

This trial-and-error process is reminiscent of how animals or humans learn through experience: actions that lead to favorable results tend to be repeated, while those that lead to poor outcomes are gradually discarded [8]. At the core of RL lies a continuous loop of interaction between an agent, representing the decision maker, and an environment, representing the system with which the agent interacts. At each discrete time step t, the agent observes the current state (s_t) of the environment, chooses and executes an

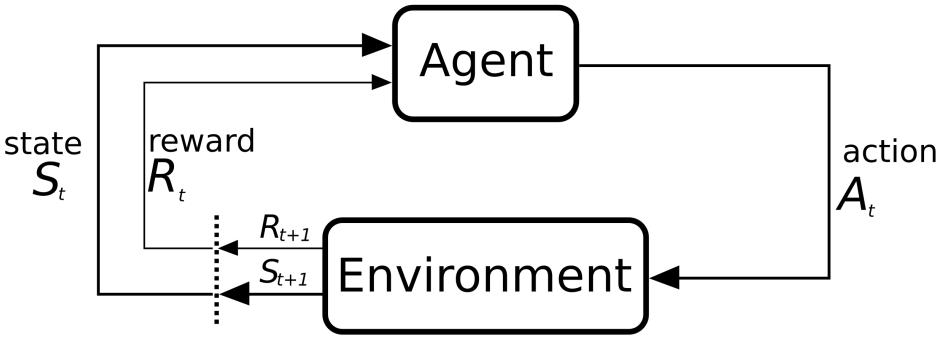


Figure 1.1: Agent–environment interaction in reinforcement learning. [8]

action (a_t), after which the environment responds by transitioning to a new state (s_{t+1}) and emitting a scalar reward (r_t) that serves as feedback about the desirability of the action's outcome. This cyclical process, often referred to as the agent-environment loop, is illustrated in Figure 1.1. Over many repetitions of this loop, the agent adjusts its behavior in order to maximize the sum of rewards it obtains over time.

The possible states the environment can take are collected in the state space, while the possible actions the agent can perform belong to the action space. These spaces can be discrete or continuous. In a simple setting such as a gridworld, states may correspond to positions on the grid, and actions to a small set of legal moves, for example move left or go ahead by one step. In robotics, however, the spaces are typically continuous: states may include real-valued sensor readings such as positions, velocities, or orientations, while actions correspond to real-valued control commands. Other examples include steering and throttle in autonomous driving or torque control in robotic manipulation.

The agent's decision-making mechanism is defined by its policy, a function that maps states to actions. The goal of RL is to discover or optimize such a policy so that the long-term accumulation of rewards is as high as possible. The interaction unfolds across a finite time horizon, called an episode, which consists of a sequence of experiences, or trajectory, represented as $\tau = [(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots]$. An episode can terminate for different reasons: the task may be completed successfully, it may fail with a negative outcome (for example, a crash or violation of safety limits), or it may simply reach a predefined time horizon. At the end of each episode, the agent uses the collected experience to improve its policy.

This abstraction is deliberately general and can describe a wide range of applications, from board games and recommendation systems to robotics [9]. In the specific case of aerial vehicles, the state may include the drone's position, orientation, and velocity; the actions correspond to thrust or torque commands applied to the motors; and the reward function encodes objectives such as stable hovering, trajectory tracking, or energy

efficiency.

Formally, this type of sequential decision-making problem is modeled using a Markov Decision Process (MDP), which provides the mathematical foundation for most reinforcement learning algorithms.

1.1.1 Markov Decision Processes

The intuitive description of the agent–environment loop can be given a precise mathematical formulation using the framework of Markov Decision Processes (MDPs) [10]. An MDP provides a way to model sequential decision-making in uncertain environments, and is formally defined by a tuple

$$(\mathcal{S}, \mathcal{A}, P, r, \gamma),$$

where:

- $\mathcal{S}$ is the state space, the set of all possible situations the agent may encounter;
- $\mathcal{A}$ is the action space, the set of decisions available to the agent;
- $P(s'|s, a)$ is the state transition function, describing how the environment evolves from state s to state s' when action a is taken;
- $r(s, a)$ is the reward function, assigning a scalar signal that evaluates the desirability of taking action a in state s;
- $\gamma \in [0, 1)$ is the discount factor, which controls how future rewards are weighted relative to immediate ones.

In its most general form, the next state could depend on the entire history of interactions:

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

This formulation quickly becomes intractable, since the number of dependencies grows with time. To address this, the Markov property is assumed: the next state depends only on the current state and action,

$$s_{t+1} \sim P(s_{t+1}|s_t, a_t).$$

Although this assumption may appear restrictive, in practice the state can often be defined to include all information required to predict the future, thereby making the process effectively Markovian.

The transition function P then serves as a model of the environment's dynamics. In deterministic systems, a given state–action pair always produces the same successor state. In stochastic systems, by contrast, the outcome is uncertain, and the same action in the

same state may lead the environment to transition to different future states with certain probabilities. This stochasticity is crucial in many real-world scenarios. For example, a drone that issues identical thrust commands at two different times may experience slightly different accelerations due to wind gusts, sensor noise, or hardware imperfections.

The reward function r complements the dynamics by providing immediate evaluative feedback. In either case, the agent does not know the reward function in advance and only receives the scalar values it generates during interaction. Reward design is therefore critical, since it encodes the task's objectives and strongly shapes the behavior that will emerge. In aerial robotics, for instance, rewards may penalize deviation from a reference trajectory, encourage smooth and energy-efficient flight, or assign large negative values to unsafe maneuvers such as collisions.

To reason about the long-term consequences of actions, reinforcement learning introduces the concept of the return $R(\tau)$, with τ the trajectory over an episode that ends at $t = T$:

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^T r^T = \sum_{t=0}^{T} \gamma^t r_t \quad \text{with} \quad \gamma \in [0, 1]$$

The discount factor γ ensures that this sum remains finite and encodes the trade-off between valuing immediate and distant outcomes. Lower values of γ instruct the agent to maximize short-term rewards, while higher values encourage it to prioritize long-term rewards.

The agent's ultimate goal is to maximize the objective J defined as the expectation of the returns over many trajectories sampled from a policy π:

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

This compact formulation provides the bridge between the abstract MDP model and the concrete learning algorithms used in reinforcement learning.

Finally, while the Markov property makes the problem mathematically tractable, it is not always satisfied in practice. In many real-world domains the agent does not observe the true underlying state but only partial and noisy information. Such cases are modeled as Partially Observable Markov Decision Processes (POMDPs) [11], which explicitly account for uncertainty. For aerial robots, this is the typical situation: onboard sensors such as IMUs, barometers, or cameras provide only partial glimpses of the true state, making estimation and control inherently more challenging.

1.1.2 Policies and Value Functions

Within the MDP framework, the behavior of the agent is determined by its policy. A policy, denoted $\pi(a|s)$, specifies the probability of selecting action a when the agent is

in state s. Policies can be either deterministic, mapping each state to a single action, or stochastic, assigning probabilities to different possible actions. Stochasticity plays an important role in reinforcement learning, as it directly supports the balance between exploitation and exploration. Exploitation refers to leveraging current knowledge to select actions that are known to yield high rewards, while exploration refers to trying uncertain actions in order to discover strategies that may lead to improved performance. An agent that only exploits risks converging prematurely to suboptimal behavior, whereas an agent that explores excessively may act inefficiently and fail to accumulate meaningful rewards.

This dilemma is particularly acute in robotics. For aerial vehicles, excessive exploration may result in unsafe maneuvers and crashes, while a conservative strategy that exploits too early might succeed only at basic hovering and never discover more advanced flight behaviors. Striking the right balance is therefore crucial, and mechanisms such as injecting randomness into the action selection process or designing curiosity-driven reward signals are often employed to encourage safe but effective exploration.

To assess and improve policies, reinforcement learning relies on *value functions*, which estimate the expected return associated with specific states or state–action pairs. The state-value function $V^\pi(s)$ measures the expected discounted return when starting from state s and subsequently following policy π:

$$V^\pi(s) = \mathbb{E}_{s_0=s,\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right].$$

The action-value function (or Q-function) $Q^\pi(s,a)$ instead measures the expected return when the agent takes action a in state s and then continues according to policy π:

$$Q^\pi(s,a) = \mathbb{E}_{s_0=s,a_0=a,\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right].$$

These two functions are closely related. The state-value function can be expressed as the expectation of the action-value function over the policy's distribution of actions,

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)],$$

while the action-value function can be written in terms of the immediate reward and the value of the next state,

$$Q^\pi(s,a) = \mathbb{E}_\pi\big[r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\big].$$

Together, value functions form the backbone of many reinforcement learning methods. By providing structured estimates of the expected return, they enable systematic evaluation and improvement of policies at the level of individual states and actions.

1.2 Deep Learning

Artificial Intelligence (AI) aims at solving tasks that are intuitive for humans yet difficult to describe formally with explicit rules. A powerful approach is to let machines learn from data. Deep Learning (DL) addresses this by organizing computation into multiple layers, so that simple features are composed into progressively more complex abstractions. In this sense, deep networks can be viewed as learning a hierarchy of concepts, where each level builds upon the previous one. This hierarchical representation underpins the success of deep learning in domains such as computer vision, natural language processing, and control [12].

The following subsections first introduce the basic building blocks of neural networks and their architectures, and then describe how these models are trained to realize useful functions in practice.

1.2.1 Deep Neural Networks

The idea of artificial neurons has been present in the scientific literature since the mid-twentieth century, but it was Rosenblatt's perceptron [13] in the 1950s that first demonstrated how networks of such units could be trained on data. Interest declined in subsequent decades due to theoretical limitations and limited computational resources, until the rediscovery of efficient training algorithms such as backpropagation in the 1980s [14]. The true breakthrough, however, came in the 2000s, when large datasets and hardware accelerators enabled the training of much deeper networks. This gave rise to what is now called deep learning [12], which has transformed fields ranging from computer vision to robotics.

At their core, neural networks can be understood as parametric function approximators $f(x;\theta)$, mapping an input x to an output y through parameters θ (weights and biases). Although their name is inspired by biology, where neurons integrate multiple inputs and produce an output signal, the analogy is largely superficial. In artificial networks, a neuron computes a weighted sum of its inputs, adds a bias term, and applies a nonlinear activation function:

$$z = \sum_i w_i x_i + b, \qquad y = \phi(z).$$

The nonlinearity $\phi(\cdot)$ is essential: without it, the composition of multiple layers would collapse into a single linear function. With suitable nonlinear activations, multilayer networks are in fact universal function approximators, capable of representing any continuous function to arbitrary accuracy under mild conditions [15]. Common activation choices include sigmoid, hyperbolic tangent, and the Rectified Linear Unit (ReLU).

Neural networks are built by composing many such units into layers. The input layer receives the external data, with one unit for each input dimension. The output

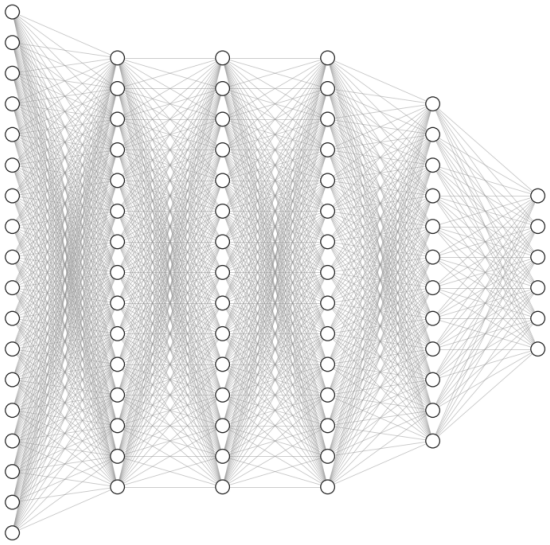


Figure 1.2: A fully connected feedforward neural network with four hidden layers. Each neuron in one layer connects to every neuron in the next layer.

layer produces predictions or decisions, with a size determined by the specific task (e.g., one output per class in classification, or a continuous value in regression or control). Between these, one can insert an arbitrary number of hidden layers, each transforming the representation before passing it forward. The number of hidden layers defines the depth of the network: deeper networks are said to be deep neural networks.

Depth plays a crucial role in practice. While even shallow networks are theoretically universal approximators, deep architectures can achieve complex mappings far more efficiently, often requiring exponentially fewer units. This efficiency comes from their ability to learn hierarchical representations: lower layers typically detect simple features, intermediate layers combine them into more complex patterns, and higher layers form abstract concepts. In image analysis, for instance, the first layers may learn to recognize edges, subsequent ones assemble edges into parts such as eyes or mouths, and the final layers integrate parts into whole faces. Figure 1.3 illustrates this process.

Beyond fully connected architectures, a wide range of specialized designs have been introduced to exploit the structure of data. Convolutional neural networks leverage spatial locality and weight sharing to process visual inputs efficiently; recurrent neural networks capture temporal dependencies in sequential data; and attention-based models such as Transformers [16] excel at modeling long-range dependencies. These architectures show how the same basic building blocks — neurons and layers — can be organized differently to match the structure of a task.

Neural networks are widely applied across all major machine learning paradigms, including supervised, unsupervised, and reinforcement learning. In this thesis, the focus is on their role in reinforcement learning, where they serve as powerful function approx-

Figure 1.3: Hierarchical features extracted by the model: (left) low-level Gabor-like filters, (middle) mid-level parts-based features, (right) high-level holistic faces. Inspired by the depiction in [17].

imators for policies and value functions, enabling agents to learn directly from high-dimensional data such as sensor measurements or visual observations.

1.2.2 Training Deep Neural Networks

Designing a neural network architecture specifies the class of functions it can represent, but it is the training process that determines which particular function is realized. Training consists of adjusting the network parameters so that its predictions approximate the desired outputs. This requires defining an error measure, computing how sensitive that error is to changes in the parameters, and then updating the parameters in a way that reduces it.

Formally, let $\hat{y} = f(x; \theta)$ be the prediction of the network for an input x with parameters θ, and let y denote the target. A loss function $L(y, \hat{y})$ quantifies the discrepancy between prediction and target. In supervised learning, this might be the mean squared error for regression or cross-entropy for classification. In reinforcement learning, by contrast, the loss is derived from policy or value objectives that encourage actions leading to higher cumulative rewards. In both cases, the total loss often combines a primary term with additional regularization terms that penalize overly complex solutions, thereby encouraging better generalization.

Training unfolds as an iterative cycle. In the forward pass, data propagate through the network to produce predictions. The loss is then evaluated to measure the error. In the backward pass, the backpropagation algorithm [14] computes the gradient of the loss with respect to each parameter by systematically applying the chain rule of calculus. This requires that the activation functions be differentiable, so that error signals can be propagated through the nonlinearities. As a simple illustration, the gradient of the loss with respect to a weight w_{ji} connecting neuron i to neuron j is proportional to the input

y_i times the error signal at unit j:

$$\frac{\partial L}{\partial w_{ji}} = \delta_j \, y_i.$$

Once gradients are available, parameters are updated according to a rule based on gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(y, \hat{y}(\theta)),$$

where α is the learning rate controlling the step size. Repeating this cycle over many iterations gradually reduces the loss as the network adapts to the data.

The choice of optimization algorithm strongly influences training efficiency and stability. The simplest method, stochastic gradient descent (SGD), updates parameters using small random batches of data. Variants such as momentum improve efficiency by accumulating information from past gradients, helping the optimizer escape flat or noisy regions. More advanced adaptive methods, such as Adam [18], RMSProp [19], or Adagrad [20], adjust learning rates individually for each parameter based on gradient history. These algorithms often accelerate convergence and reduce sensitivity to hyperparameter choices.

A central challenge in training deep networks is generalization: performing well not just on the training data but also on unseen examples. A model can even memorize the entire training dataset, achieving very low error on those same examples but performing poorly on new, unseen data — a phenomenon known as overfitting. Several strategies are used to improve a network's ability to generalize. Some modify how data are presented to the model, for example by augmenting the dataset with variations or noise. Others constrain the model's complexity, such as adding penalties on large weights. Architectural choices can also stabilize learning. These techniques all aim to prevent overfitting and help the model capture general patterns rather than memorizing the training data.

Successful training also depends on carefully chosen hyperparameters such as the learning rate. While often treated as heuristics, these design choices are often critical for effective training.

These principles apply to training deep networks in general, regardless of whether the learning signal comes from labeled datasets or from interaction with an environment. In reinforcement learning, the same backpropagation and gradient-based optimization methods are used; the main difference is that the loss is derived from rewards rather than ground-truth labels.

1.3 Deep Reinforcement Learning

Reinforcement learning provides a principled mathematical framework for sequential decision making, while deep learning offers powerful tools for representing complex functions.

Their combination, known as deep reinforcement learning (DRL), has enabled agents to solve problems that were previously intractable. Landmark results such as playing Atari directly from pixels [21] and mastering the game of Go [22] demonstrated the potential of DRL and brought the field to wide attention.

Classical reinforcement learning methods relied on tabular representations of policies and value functions, or on linear approximations combined with carefully engineered features. These approaches were effective in small, discrete environments but could not scale to domains with continuous, high-dimensional states and actions. Neural networks address this limitation by serving as flexible function approximators. A policy $\pi(a|s)$ can be represented by a network that maps states to action distributions, while value functions $V^\pi(s)$ or $Q^\pi(s, a)$ can be approximated by networks that estimate their values. Crucially, networks allow generalization across states, enabling agents to behave reasonably even in situations not explicitly encountered during training.

At the same time, integrating deep learning into reinforcement learning introduces new challenges. Unlike supervised learning, where training examples are independent and identically distributed, reinforcement learning generates highly correlated data influenced by the agent's own behavior. Both the input distribution and the learning targets change as the policy evolves, making training unstable. Neural networks, while expressive, can overfit to recent experiences or amplify estimation errors if not trained carefully. Stability and sample efficiency have therefore become central concerns in the design of modern DRL algorithms.

Broadly speaking, DRL methods can be grouped into three main families. Policy-based algorithms learn a parameterized policy directly, which makes them very general: they can be applied to discrete, continuous, or mixed action spaces, and they optimize the agent's objective function explicitly. By the Policy Gradient Theorem [23], they are guaranteed to converge to a locally optimal policy. Their main drawbacks, however, are the high variance of gradient estimates [24], which stems from the stochasticity of trajectories, and their tendency to be sample-inefficient, since past data cannot easily be reused. Value-based algorithms, in contrast, focus on learning value functions from which a policy is derived. This makes them more sample-efficient, as stored experiences can be leveraged repeatedly. A seminal example is Q-learning [25], later extended to deep Q-networks that demonstrated success in Atari games. However, value-based algorithms learn the policy only indirectly, which may introduce bias or instability, and most practical versions are limited to discrete action spaces. Finally, model-based algorithms incorporate or learn a model of the environment's dynamics to plan ahead. With an accurate model, an agent can reduce costly interactions by simulating trajectories offline, which makes these methods attractive when real-world data collection is expensive. Their effectiveness, however, is often limited by the difficulty of learning reliable models, as prediction errors accumulate when planning over long horizons.

Another important distinction among DRL algorithms is whether they are on-policy or off-policy. On-policy methods learn from data generated by the current policy itself; after each update, new trajectories must be collected. This makes them conceptually simple and often more stable, but also relatively sample-inefficient. Off-policy methods, in contrast, can reuse experiences gathered by past versions of the policy or even by different policies, typically via replay buffers and importance weighting. This dramatically improves sample efficiency, which is particularly valuable in robotics, but also introduces challenges in avoiding bias and ensuring stable updates.

Because these three families exhibit complementary strengths and weaknesses, hybrid approaches have been developed. The most prominent are actor–critic methods, which combine direct policy optimization with value-based estimation. By allowing the policy (the actor) to be guided by value information (the critic), these methods achieve a better balance between efficiency and stability.

1.3.1 Actor–Critic Methods

Actor–critic algorithms combine the strengths of both value-based and policy-based approaches [26, 27]. They maintain two interacting components: an actor, representing the policy, and a critic, estimating value functions. The actor proposes actions given states, while the critic evaluates their long-term utility. In contrast to raw reward signals, which may be sparse or delayed, the critic provides denser and more informative feedback by predicting future returns, thereby guiding the actor more effectively. This interaction results in more stable and sample-efficient learning than pure policy gradient methods.

In practice, actor–critic methods are implemented with deep neural networks. A common design uses a single network with two output heads: one for the policy and one for the value estimate. Sharing the lower layers allows both actor and critic to learn a common representation of the state space, while the separate heads specialize for their respective tasks. This design reduces the number of parameters and speeds up learning, but it can also make optimization less stable because the two components send competing gradient signals through the shared layers. Alternatively, separate networks eliminate this interference at the cost of higher memory and computational requirements.

The dynamics of training also play a role in stability. At the beginning of learning, the critic has little information and may provide poor estimates. Since the actor depends on the critic's feedback, its updates are initially unreliable. As the critic improves, the actor receives more accurate gradients, but the coupling between the two means that instability in one can destabilize the other. Managing this interplay is one of the central challenges of actor–critic methods.

Formally, consider a policy $\pi_\theta(a|s)$ parameterized by θ. The policy gradient theo-

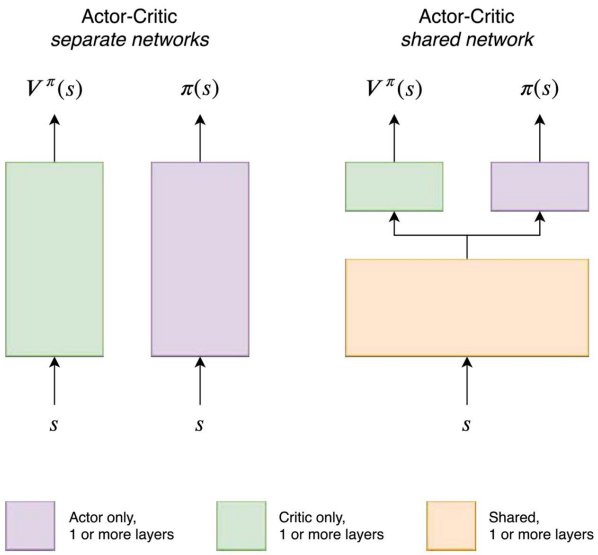


Figure 1.4: Actor–critic architecture. The actor proposes actions based on the current state, while the critic evaluates the expected return of those actions. Both components can be implemented with shared or separate neural networks. [28]

rem [23] shows that the gradient of the expected return can be written as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, Q^\pi(s,a)\right],$$

where d^π is the discounted state distribution under policy π and $Q^\pi(s,a)$ is the action-value function. Since the true Q^π is unknown, it is replaced by an estimate from the critic. A common refinement is to use the advantage function

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

which measures how much better or worse an action is compared to the average performance of the policy in state s. Subtracting this baseline reduces variance in the gradient estimate while preserving its expectation. Beyond variance reduction, the advantage function avoids misleading updates in globally good or bad states: if all actions in a state are equally poor, the advantage is near zero and no action is unfairly penalized; if all are equally good, none is over-rewarded. A useful property is that $\mathbb{E}_{a\sim\pi}[A^\pi(s,a)] = 0$, which ensures that advantage-based updates remain unbiased relative to the state value.

Another refinement is the use of n-step returns to estimate Q^π. Instead of relying only on immediate rewards or on long-term bootstrapping, n-step methods combine a finite sequence of actual rewards with a value estimate of the future state. This balances variance (from raw returns) against bias (from function approximation), with the parameter n controlling the tradeoff. Such bias–variance considerations are central to achieving stability. The critic itself is typically trained via temporal-difference (TD) learning, where predicted values are updated toward bootstrapped targets, and the error is minimized

with a regression loss such as mean squared error.

In continuous action spaces, which are typical in robotics, the actor often parameterizes a probability distribution over actions. For example, the network may output the mean and variance of a Gaussian distribution for each action dimension, from which actual motor commands are sampled. This stochastic formulation naturally supports exploration while allowing gradients to flow through the distribution parameters. As training progresses, the distributions sharpen: the means converge toward actions with higher expected returns, while the variances shrink, reflecting growing confidence in the learned policy and reducing unnecessary exploration.

The actor–critic framework thus provides a general recipe: update the critic to better approximate value functions from experience, and update the actor to select actions that the critic deems advantageous. Many of the most successful deep reinforcement learning algorithms, including Proximal Policy Optimization (PPO) and Twin Delayed Deep Deterministic Policy Gradient (TD3), are particular realizations of this idea, differing mainly in how the actor and critic are parameterized and updated to achieve stability and efficiency.

1.3.2 Proximal Policy Optimization (PPO)

While actor–critic methods provide a flexible framework, early implementations often suffered from instability. A central challenge is that policy gradient methods update parameters in the neural network's parameter space, not directly in policy space. Because the mapping between these spaces is highly nonlinear, even small parameter changes can produce large and unpredictable shifts in the resulting policy. This can trigger performance collapse: once the policy starts performing poorly, it generates bad trajectories, which then reinforce poor updates in subsequent iterations, further degrading performance.

To mitigate this issue, Trust Region Policy Optimization (TRPO) introduced the idea of constraining each policy update to remain within a "trust region" measured by the Kullback–Leibler (KL) divergence between old and new policy distributions [29]. TRPO provided strong theoretical guarantees and improved stability, but it required solving a complex constrained optimization problem, making it computationally demanding and difficult to implement.

Proximal Policy Optimization (PPO) [30] was proposed as a simpler and more efficient alternative. Instead of enforcing a hard constraint, PPO uses a surrogate objective that directly discourages excessively large updates. For a given action a_t in state s_t, the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

measures how much the new policy deviates from the old one. Standard policy gradients maximize $\mathbb{E}[r_t(\theta)A_t]$, where A_t is an estimate of the advantage function. PPO modifies

this by clipping $r_t(\theta)$ to the range $[1-\epsilon, 1+\epsilon]$, with ϵ a small constant (e.g., 0.1–0.2). The clipped objective is

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t) \right].$$

This ensures that updates cannot push the policy too far in a single step, while still allowing improvement when it is consistent with the critic's estimates.

In practice, PPO alternates between collecting trajectories with the current policy and performing several epochs of stochastic gradient ascent on minibatches of this surrogate objective, together with updates to the critic. Advantage estimates A_t are typically computed using generalized advantage estimation (GAE) [31], which strikes a balance between bias and variance in long-term return estimates. These implementation details contribute significantly to PPO's robustness.

Thanks to its combination of simplicity, stability, and performance, PPO has become one of the most widely adopted algorithms in deep reinforcement learning. It has demonstrated strong results in benchmark environments ranging from Atari [32] to MuJoCo continuous control [33], and it is frequently used as a baseline in robotics research, including aerial robotics where stability and sample efficiency are critical.

1.3.3 Twin Delayed Deep Deterministic Policy Gradient (TD3)

While Proximal Policy Optimization has become the standard among on-policy actor–critic algorithms, many applications benefit from off-policy methods, which reuse past experiences stored in a replay buffer. Off-policy algorithms can be far more sample-efficient, since they allow the agent to learn repeatedly from the same trajectories instead of discarding them after each update. This property is especially valuable in robotics, where data collection on physical systems is costly and limited.

The Deep Deterministic Policy Gradient (DDPG) algorithm [34] pioneered the use of deterministic policy gradients with deep neural networks, enabling direct learning in continuous action spaces. However, DDPG was found to be brittle in practice: it often suffered from overestimation bias in the critic, sensitivity to hyperparameters, and unstable training dynamics.

Twin Delayed Deep Deterministic Policy Gradient (TD3) [35] was proposed as an improvement over DDPG, introducing three key modifications that directly address these weaknesses:

- **Twin critics**: Instead of relying on a single estimate of the action-value function, TD3 trains two independent critics and uses the smaller of their predictions to compute targets for the actor update. This introduces a controlled pessimism that reduces overestimation bias, a common failure mode of value-based methods.

- **Target policy smoothing**: When computing target values for critic updates, TD3 adds clipped noise to the target actions. This prevents the critic from exploiting sharp peaks in the value landscape and encourages smoother, more robust policies.
- **Delayed policy updates**: While the critics are updated at every training step, the actor is updated less frequently, typically once every two critic updates. By allowing the critics to stabilize first, this schedule reduces the risk of the actor overfitting to noisy or inaccurate value estimates.

Formally, TD3 follows the standard off-policy actor–critic template. A deterministic actor $\mu_\theta(s)$ outputs a specific action given a state, while two critics $Q_{\phi_1}(s,a)$ and $Q_{\phi_2}(s,a)$ estimate action values. The critics are trained by minimizing the Bellman error [8], which measures the discrepancy between their current predictions and a target value. The target represents a one-step reward plus the discounted value of the next state, and is computed using slowly updated copies of the actor and critics known as target networks. This technique prevents training instabilities that arise when the targets change too quickly.

In TD3, the target value is defined as

$$y = r + \gamma \min_{i=1,2} Q_{\phi_i'}\big(s', \mu_{\theta'}(s') + \epsilon\big),$$

where $Q_{\phi_i'}$ and $\mu_{\theta'}$ are the target critics and target actor, respectively, and ϵ is clipped noise added for policy smoothing. The actor is updated by maximizing the expected critic value

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim\mathcal{D}}\big[\nabla_\theta \mu_\theta(s) \nabla_a Q_{\phi_1}(s,a)|_{a=\mu_\theta(s)}\big],$$

but only on delayed steps. This staggered update schedule improves stability by allowing the critics to provide more reliable gradients before the policy is adjusted.

TD3 has proven to be significantly more stable and reliable than DDPG across a variety of benchmark tasks, including MuJoCo continuous control domains. Its design illustrates a broader principle in deep reinforcement learning: careful modifications to reduce bias, smooth updates, and control the pace of learning can dramatically improve both performance and robustness in continuous control problems.

1.4 Drone Dynamics and Control

The control of multirotor aerial vehicles, and in particular quadrotors, requires a solid understanding of both their dynamics and the different ways control commands can be represented. Although these platforms appear mechanically simple, their nonlinear, underactuated nature makes stabilization and trajectory tracking nontrivial. A variety of control interfaces have been developed, ranging from high-level abstractions such as thrust and body rates to low-level direct motor commands, each with specific advantages and

trade-offs. On top of these representations, different control strategies can be employed. Classical approaches rely on explicit models and hierarchical structures such as cascaded PID loops or model predictive control, whereas learning-based methods attempt to bypass explicit modeling by training policies directly from data. This section reviews the key elements of quadrotor modeling, the commonly used control input representations, and the main control paradigms that have been explored in the literature.

1.4.1 Multirotor Dynamics

Multirotor aerial vehicles, and in particular quadrotors, are among the most studied classes of unmanned aerial vehicles due to their mechanical simplicity, maneuverability, and broad range of applications. Despite this apparent simplicity, they pose considerable control challenges because they are nonlinear, underactuated systems: the four propellers act as independent actuators, but their combined effect can be described in terms of a total thrust force and three body torques, while the vehicle itself has six degrees of freedom, corresponding to translation along three axes and rotation about them [36, 37, 38].

The dynamics of a rigid-body quadrotor can be described in terms of two reference frames: the inertial frame $\mathcal{I}$ and the body-fixed frame $\mathcal{B}$. Let $p = [x, y, z]^T \in \mathbb{R}^3$ denote the position of the center of mass in $\mathcal{I}$, $v = \dot{p}$ the translational velocity, $R \in SO(3)$ the rotation matrix describing orientation from body to inertial frame, and $\omega = [p, q, r]^T$ the angular velocity expressed in $\mathcal{B}$ (with p, q, r denoting roll, pitch, and yaw rates respectively). The translational dynamics follow Newton's second law:

$$m\dot{v} = mge_3 - TRe_3 + F_d,$$

where m is the vehicle mass, g the gravitational acceleration, $e_3 = [0, 0, 1]^T$, T the total thrust generated by the propellers (acting along the negative body z-axis), and F_d represents unmodeled aerodynamic disturbances such as drag and ground effect. The rotational dynamics are governed by Euler's equations:

$$J\dot{\omega} = \tau - \omega \times J\omega,$$

where J is the inertia matrix and $\tau = [\tau_\phi, \tau_\theta, \tau_\psi]^T$ is the torque vector around roll, pitch, and yaw axes.

Each rotor produces two main effects: an upward thrust force that contributes to lifting the vehicle, and a reactive drag torque caused by aerodynamic resistance to rotation. If rotor i spins at angular velocity Ω_i, the thrust force can be approximated as $f_i = k_f\Omega_i^2$ and the drag torque as $q_i = k_m\Omega_i^2$, where k_f and k_m are the thrust and drag coefficients of the propellers, typically obtained from experimental identification or manufacturer

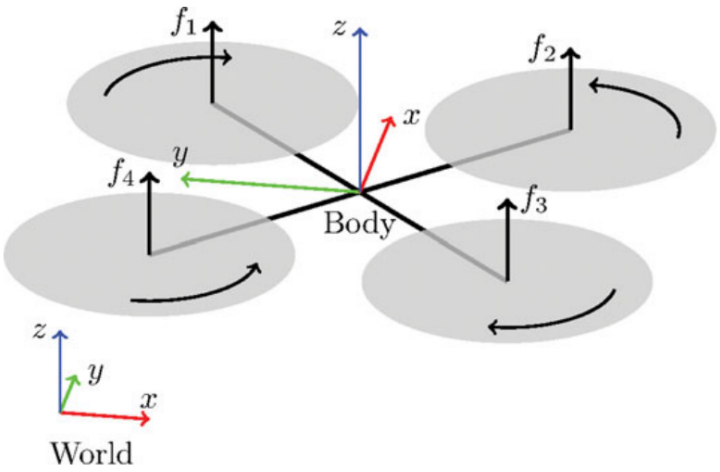


Figure 1.5: Free-body diagram of a quadrotor. The thrusts generated by the rotors combine to produce total thrust and body torques, which interact with gravitational, aerodynamic, and inertial forces. Adapted from [39].

specifications. Assuming a "×" configuration in which rotors 1 and 3 rotate clockwise and rotors 2 and 4 counterclockwise (as illustrated in Fig. 1.5), the mapping from individual rotor speeds to total thrust and body torques is

$$\begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & lk_f & 0 & -lk_f \\ -lk_f & 0 & lk_f & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix},$$

where l is the distance from each rotor to the vehicle's center of mass.

1.4.2 Control Input Representations

While the lowest-level physical inputs to a multirotor are the motor angular velocities, controllers rarely operate directly in terms of individual Ω_i. Instead, several abstractions are used depending on the control architecture. These abstractions can be understood as forming a hierarchy of input representations, from higher-level commands that simplify controller design to lower-level commands that expose more of the raw vehicle dynamics [7].

At the highest level, controllers may command desired positions or velocities in the inertial frame. These inputs are intuitive and task-oriented, allowing the controller to specify where the vehicle should move without dealing directly with its attitude dynamics. Such high-level commands are typically tracked by cascaded lower-level controllers that handle attitude stabilization and thrust allocation internally.

A common intermediate-level interface is the collective thrust and body rates (CTBR)

representation, where the control inputs are (T, p, q, r). Here T denotes the collective thrust, while p, q, r are the commanded angular rates around roll, pitch, and yaw. This representation cleanly separates vertical thrust generation from rotational stabilization and is implemented in many low-level flight controllers, such as PX4[1] or the Crazyflie[2] firmware.

Going down another level, another commonly used interface is Single Rotor Thrusts (SRT), where the control inputs are the individual rotor thrusts (f_1, f_2, f_3, f_4). This representation directly exposes the vehicle geometry and the mixing of the actuators, but avoids the additional non-linear mapping between RPM and thrust. It therefore provides finer control authority than CTBR while remaining more tractable than raw motor commands.

At the lowest level, one may issue direct motor commands, either in the form of motor angular velocities $(\Omega_1, \Omega_2, \Omega_3, \Omega_4)$ or normalized setpoints in $[0, 1]$ that are scaled to RPM ranges by the firmware. This provides maximum control authority and bypasses internal mixing, which is attractive in reinforcement learning research and aggressive control. However, it requires careful handling of motor dynamics, nonlinearities, and saturation, and can be less robust in real-world conditions.

The choice of input level has important implications for both control and learning. Higher-level abstractions reduce complexity and leverage embedded stabilizers, but constrain access to low-level dynamics. Conversely, lower-level interfaces such as motor RPMs increase flexibility but also amplify the system's nonlinearities and enlarge the reality gap between simulation and hardware. In reinforcement learning, this hierarchy affects learning difficulty: high-level actions such as position or velocity commands influence the task outcome more directly, making it easier for the agent to discover useful behaviors, whereas low-level actions such as motor commands affect position only indirectly through multiple layers of integration, which weakens the reward feedback signal and makes exploration noisier and training less sample-efficient [7].

1.4.3 Classical and Learning-Based Control

The control of multirotors has traditionally relied on classical model-based methods. These architectures are typically organized into a perception–planning–control pipeline: the perception block estimates the state of the vehicle from onboard sensors such as IMUs or cameras; the planning block generates feasible trajectories given mission objectives and constraints; and the control block ensures that the vehicle follows the desired references. Within the control block, cascaded proportional–integral–derivative (PID) controllers remain the most widely adopted solution. Inner loops regulate angular rates and attitude,

[1] `https://docs.px4.io/main/`

[2] `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/`

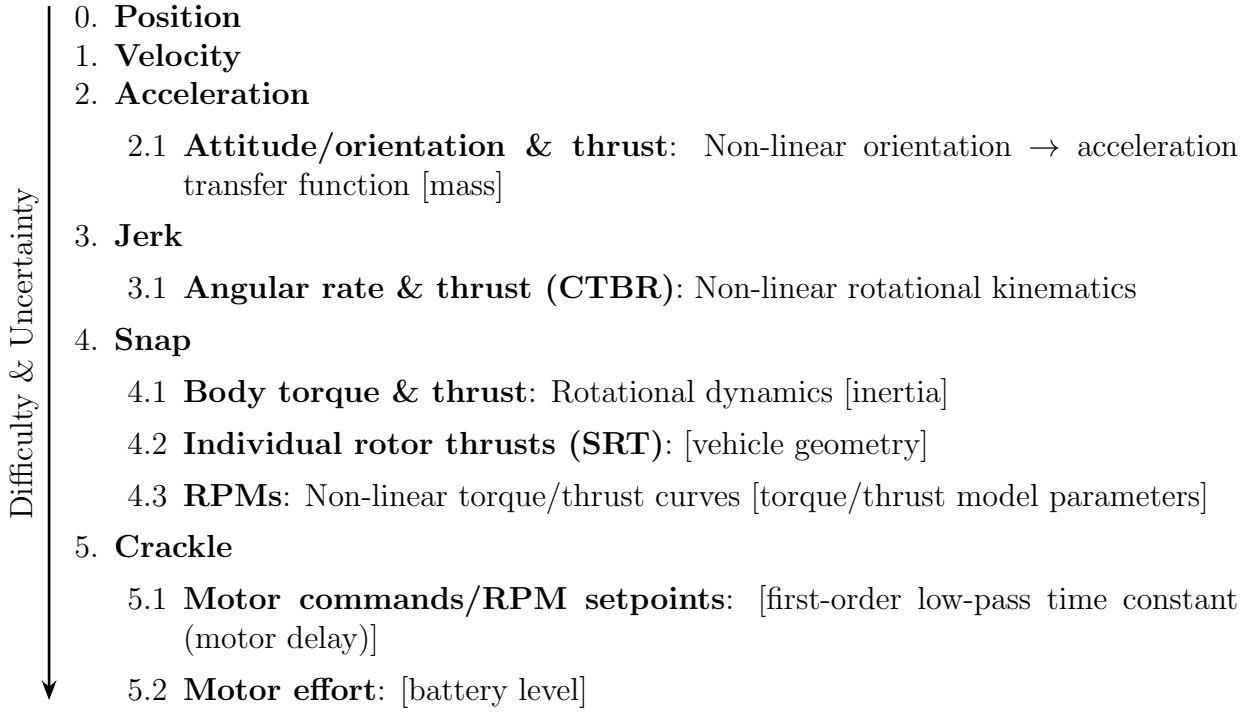


Figure 1.6: Comparison of common control input representations for multirotors. Higher-level abstractions simplify controller design, while lower-level interfaces provide more flexibility at the cost of complexity and increased sim-to-real gap. [7]

providing fast stabilization, while outer loops govern position and velocity. This separation of time scales simplifies tuning and guarantees stability under linearized dynamics. Such cascaded PID designs are standard in both research platforms (e.g., Crazyflie) and commercial autopilots [3]. Other classical approaches include linear–quadratic regulators (LQR), feedback linearization, and, more recently, model predictive control (MPC), which optimizes trajectories over a finite horizon while accounting for actuator limits and predictive dynamics [40]. Despite its advantages, MPC requires accurate models and significant onboard computational resources. Although robust and effective, classical approaches present important limitations. They depend on precise system identification and manual gain tuning, which are time-consuming and sensitive to changes in vehicle configuration or operating conditions. Moreover, their modular separation between perception, planning, and control can lead to inefficiencies in highly dynamic or uncertain environments.

Learning-based control methods take a different perspective. Instead of relying on explicit models, they employ data-driven techniques—often deep neural networks—to approximate control policies. Reinforcement learning enables end-to-end optimization of these policies through interaction with simulated or real environments. These methods can adapt to nonlinearities and unmodeled disturbances, and in principle unify perception, planning, and control. Depending on the design, neural networks may replace individual modules of the classical pipeline (e.g., perception or trajectory generation) or act as

complements to model-based controllers. Hybrid strategies are an especially active research direction. For example, [41] showed how policy search can be used to tune the parameters of an MPC controller for agile flight, while [42] trained a perception-driven policy that outputs dynamically feasible trajectories, which are then tracked by a conventional control stack. These approaches show that learning can be progressively integrated into established control pipelines while retaining the modular structure and well-tested behavior of classical controllers.

End-to-end learning approaches, on the other hand, attempt full mappings from raw sensory data to motor commands. They are attractive because they resemble the way human pilots directly map perception to action. Despite open challenges—such as large data requirements, sensitivity to reward design, limited interpretability, and weaker theoretical guarantees of stability and robustness—they are widely regarded as a promising direction for future autonomous flight, especially in highly dynamic environments and in agile contexts. Recent advances in large-scale simulation, domain randomization, and policy distillation are progressively closing the gap toward practical deployment [43, 44].

A central bottleneck for end-to-end approaches remains transfer to real platforms. While a growing number of studies have demonstrated successful sim-to-real transfer for multirotors, these results typically rely on task-specific randomization strategies, carefully engineered reward functions, and constrained hardware setups. Achieving reliable and scalable transfer across diverse tasks, environments, and platforms is therefore still an open research problem.

1.5 Simulation Environments for RL

Training reinforcement learning policies directly on physical robots is often impractical, particularly for aerial platforms such as nano-drones. Modern algorithms require millions of interaction steps to converge, which would be infeasible on real hardware due to energy constraints, mechanical wear, and the risk of crashes. Since RL relies on trial-and-error, policies are expected to fail many times before improving; in the real world, this would translate into repeated collisions and potential damage to the platform. Real-world experiments are also difficult to parallelize and lack the reproducibility needed for systematic evaluation.

To address these challenges, researchers rely heavily on simulation environments, where virtual agents interact with physics-based models of the robot and its surroundings. Simulators provide several key advantages:

- **Safety**: policies can be tested without risk of damaging hardware.
- **Speed**: simulations can run faster than real time and, crucially, can be parallelized across CPUs and GPUs, enabling millions of steps to be collected in hours rather

than weeks.

- **Flexibility**: initial states, environments, and sensor setups can be reset or randomized arbitrarily, which is impossible or costly in the real world.

- **Control**: ground-truth information (e.g., exact state variables) is directly accessible for analysis, debugging, and reward shaping.

Beyond these advantages, the ideal simulator for robotics has additional desirable properties. It should be physically accurate, reproducing rigid-body dynamics, forces, and actuation with high fidelity; and when vision is involved, it should also provide photo-realistic rendering with realistic lighting, textures, and sensor effects. At the same time, it must remain computationally efficient to support the massive data requirements of RL. These objectives are often in conflict: the more realistic a simulator is, the slower it tends to run. Designing simulation frameworks thus involves balancing realism and speed depending on the application.

The reliance on simulation, however, introduces the well-known sim-to-real gap: discrepancies in dynamics, sensing, and actuation between simulated and real systems often prevent policies from transferring seamlessly. To mitigate this, modern simulators increasingly provide tools for domain randomization and noise injection, exposing policies to diverse dynamics and sensory conditions during training. These techniques improve robustness when policies are deployed on real hardware [45, 46, 47].

Simulation has therefore become a critical component of modern reinforcement learning pipelines for robotics. In the next section, we provide a more detailed overview of simulation platforms commonly used in reinforcement learning for robotics, with particular attention to those applicable to aerial vehicles.

1.5.1 Overview of State-of-the-Art Simulators

A variety of simulators and reinforcement learning frameworks have been developed to support research in aerial robotics, each optimizing for different aspects of the training pipeline. Broadly, they can be grouped into massively parallel platforms designed for scalability, photorealistic environments targeting perception-driven tasks, UAV-specific simulators with lightweight dynamics, and lightweight RL frameworks aimed at portability and embedded deployment. No single tool satisfies all objectives simultaneously, and the choice of simulator often reflects a compromise between speed, realism, and hardware constraints [48].

Isaac Gym / Isaac Lab. These NVIDIA tools provide GPU-accelerated, massively parallel simulation, enabling reinforcement learning with thousands of agents in parallel through tensor-based pipelines [49, 50]. Unlike traditional setups where physics simulation

runs on CPUs and policy training on GPUs, Isaac Gym executes both on the GPU, eliminating communication bottlenecks and dramatically reducing training time. It also exposes raw simulation buffers as tensors in frameworks such as PyTorch, simplifying integration, and allows duplication of environments with variations, a key feature for domain randomization. While extremely efficient for large-scale training, its focus on speed means fewer ready-to-use task-specific modules or sensor models compared to other simulators.

Flightmare. Developed in particular for quadrotors, Flightmare combines a flexible physics engine with a Unity-based rendering pipeline, which can run independently [51]. This design enables both lightweight dynamics simulation and the option for photo-realistic visual input. It also provides a sensor suite (RGB, depth, segmentation, IMU) and an API tailored for RL. Although efficient for UAV research, Flightmare does not provide GPU-based parallelization at the scale of Isaac Gym, which limits its use for very large-scale training.

RLtools. RLtools is a fast, portable, header-only C++ library for deep reinforcement learning in continuous control domains [52]. It emphasizes reproducibility, efficiency, and embedded deployment, enabling training and inference on resource-constrained hardware such as microcontrollers. While it does not provide rich environment rendering or physics simulation by itself, it has been successfully applied in UAV research, powering the training of quadrotor policies in simulation and their real-time deployment on the Crazyflie nano-drone [7].

Chapter 2

Related Work

Classical approaches to quadrotor autonomy have traditionally relied on modular pipelines that separate perception, planning, and control [4]. This design enables independent progress in each component and provides interpretability when combined with analytically proven controllers. It has supported many of the most influential demonstrations in agile flight, including high-speed obstacle avoidance with stereo vision [53], minimum-snap trajectory generation for aggressive maneuvers [54], and real-time optimization for fast autonomous flight [55]. A representative example is [56], where a convolutional neural network maps raw images to waypoints that are then processed by a trajectory planner and a low-level controller. While effective, such modular pipelines discard interactions across modules, depend on simplified assumptions, and introduce latency due to sequential processing [57]. These limitations become critical in dynamic environments, where fast perception–action loops are essential for agility.

The drawbacks of modular design have motivated research into end-to-end controllers, where perception, planning, and control are jointly learned. Reinforcement learning (RL) provides a natural framework for such policies, which directly map sensor observations or state information to control commands. Early works mainly relied on velocity-level interfaces, where the policy outputs a desired translational velocity [58, 59, 60, 61]. This choice simplifies training and leverages the robustness of existing flight stacks, but it hides the underlying platform dynamics and therefore constrains agility. Other studies adopted orientation-level actions, where the policy specifies a target attitude and velocity magnitude [62, 63]. More recently, collective thrust and body rates (CTBR) have emerged as the dominant representation, exposing both translational thrust and angular rates while remaining easier to train than direct motor control [64]. CTBR-based controllers have enabled high-performance demonstrations in drone racing and high-speed flight [42, 5]. Despite these successes, all of these action abstractions still rely on conventional low-level controllers and thus remain one step removed from true end-to-end actuation.

A central challenge across this line of research is bridging the gap between training in simulation and deployment on physical platforms. Policies trained purely in simula-

tion often fail to generalize due to unmodeled dynamics, sensor noise, or environmental variability. Several techniques have been developed to mitigate this gap. Domain randomization exposes the policy to a wide range of randomized physical and sensory parameters during training to improve robustness [45, 65]. Curriculum learning gradually increases task difficulty or penalty weights to stabilize training [66]. Collectively, these strategies enhance the robustness and adaptability of learned controllers, facilitating a more reliable transfer from simulated training environments to real-world aerial platforms.

2.1 Low-Level Sim-to-Real Transfer

While velocity, orientation, and CTBR interfaces have enabled progress in agile flight, they all rely on conventional low-level controllers to translate high-level commands into motor inputs. This reliance introduces a bottleneck: the low-level dynamics of the platform are hidden from the learning agent, constraining agility and leaving robustness dependent on manually tuned control loops. To achieve full end-to-end autonomy, research has begun exploring lower-level actuation spaces, where policies output single-rotor thrusts (SRT) or even direct motor revolutions per minute (RPM).

One of the earliest demonstrations of this approach was presented in [67], where reinforcement learning was used to directly control rotor thrusts on a quadrotor. This showed that RL could, in principle, stabilize and control the platform without relying on cascaded PID loops, marking the first step toward full end-to-end actuation. Subsequent studies expanded on this idea, for example by applying reinforcement learning to low-level autonomous tracking [68] or to drone racing tasks with single-rotor thrust actions [69]. More recent work has addressed sim-to-real robustness and generalization, such as leveraging simulation optimization to improve zero-shot transfer [70] or training a single controller capable of stabilizing multiple quadcopter platforms with different dynamics [71]. Despite these advances, learning directly at the motor level remains substantially more difficult due to sample inefficiency, instability, and sensitivity to unmodeled dynamics such as rotor delays and aerodynamic effects.

A major step forward in overcoming these challenges was the framework proposed in [7]. This approach combined several key design choices: an asymmetric actor–critic architecture to exploit privileged state information during training, explicit rotor-delay modeling and action histories to account for partial observability, and curriculum learning to gradually expose the policy to increasingly difficult conditions. Crucially, these strategies were embedded in an extremely fast custom simulator that enabled training with unprecedented sample efficiency. By leveraging the off-policy TD3 algorithm, the authors showed that policies producing direct RPM commands could be trained in seconds of wall-clock time and transferred successfully to the Crazyflie nano-UAV. This represented one of the first demonstrations of robust low-level reinforcement learning control on such

a resource-constrained platform, with performance competitive with state-of-the-art classical and learned controllers.

Building on this breakthrough, the present thesis investigates the generality of the [7] framework under different choices of tools and algorithms. Specifically, we adopt IsaacLab as the simulation environment, employ the SKRL library for training, and use Proximal Policy Optimization (PPO) instead of TD3. Furthermore, we develop a different deployment pipeline for the Crazyflie nano-UAV based on widely used tools. By evaluating whether the key ideas of [7] can be reproduced in this new setup, this work contributes to validating and extending the reproducibility of low-level sim-to-real transfer for nano-UAVs.

Chapter 3

Methodology

The main objective of this thesis is to achieve the simulation-to-reality (Sim2Real) transfer of a learning-based position controller on the Crazyflie nano-quadrotor. This work builds directly upon the framework proposed by Eschmann et al. [7], which demonstrated that reinforcement learning policies producing direct motor commands can be trained in simulation and transferred successfully to real hardware. In particular, the reward formulation and several Sim2Real strategies introduced in their work form a reliable foundation for low-level control on the Crazyflie.

However, two design choices in their framework limit its generality and accessibility. First, training, evaluation, and deployment were all conducted with RLTools [52], a custom C++ framework. While RLTools achieves remarkable performance and surpasses simulators such as Flightmare [51], its reliance on C++ raises the barrier of entry for much of the machine learning community, which typically favors Python-based frameworks. Moreover, RLTools has limited support for neural network models and algorithms compared to mainstream libraries and tools, such as PyTorch [72] or IsaacGym/IsaacLab [49]. Second, the authors adopted the TD3 algorithm. TD3 was selected in [7] for its high sample efficiency and fast wall-clock training times, properties that are particularly valuable when experimenting with computationally demanding simulators. At the same time, TD3 is less commonly adopted in recent reinforcement learning benchmarks and thus less widely supported in mainstream libraries.

In this thesis, we therefore take a different approach. We leverage IsaacLab as the simulation environment, the SKRL library [73] for training, and Proximal Policy Optimization (PPO) as the learning algorithm. Our working hypothesis is that PPO, despite its lower sample efficiency compared to TD3, can still reproduce the key results of [7] while offering a more accessible and generalizable pipeline. Beyond validating these results, the use of mainstream libraries and tools also facilitates reproducibility and integration with the broader reinforcement learning ecosystem.

The remainder of this chapter is organized as follows. Section 3.1 introduces the problem formulation and task definition. Section 3.2 then describes the simulation en-

vironment, Section 3.3 details the agent design and training procedure, and Section 3.4 outlines the deployment strategy on the real Crazyflie platform.

3.1 Problem Formulation and Task Definition

The task addressed in this thesis is position control, i.e., minimizing the error between the quadrotor's current position and a desired reference in space, including in the presence of disturbances. In both this work and in [7], this is achieved by training a controller that drives the drone back to the origin with zero linear velocity from any initial condition within reasonable bounds. Importantly, the learned policy is not constrained to stabilizing near a fixed hover state. Instead, it is trained to recover from a broad range of initial positions and velocities, rather than operating solely around a single nominal state. This design enables the policy to handle diverse starting conditions, and generalization to arbitrary setpoints follows naturally: by shifting the coordinate system so that the desired reference coincides with the origin, the same policy can be applied to track different positions or even full trajectories, provided that the induced position and velocity errors remain within the range encountered during training.

The state representation is defined as

$$\mathcal{S} = \{\mathbf{p}, \mathbf{v}, \mathbf{q}, \boldsymbol{\omega}, \boldsymbol{\omega}_m\},$$

where $\mathbf{p}$ is the position, $\mathbf{v}$ the linear velocity, $\mathbf{q}$ the orientation (quaternion), $\boldsymbol{\omega}$ the angular velocity, and $\boldsymbol{\omega}_m$ the motor speeds. Motor speeds are important for capturing the effect of actions, but they are not directly observable on many real platforms—including the Crazyflie, where PWM signals are sent open-loop to the motors. Moreover, motor dynamics introduce a noticeable delay: the effect of a new RPM command is not applied instantaneously but becomes visible only after several control cycles. This latency, caused mainly by rotor inertia, creates partial observability in the system and must be explicitly accounted for during training.

To address these challenges, [7] proposed an asymmetric actor–critic scheme. The critic, used only during training, has access to privileged information, including motor speeds and artificially injected disturbances:

$$\mathcal{O}_c = \{\mathbf{p}, R, \mathbf{v}, \boldsymbol{\omega}, \boldsymbol{\omega}_m, \mathbf{f}_r, \boldsymbol{\tau}_r\},$$

where R is the rotation matrix corresponding to $\mathbf{q}$, $\mathbf{f}_r$ is an external disturbance force, and $\boldsymbol{\tau}_r$ a disturbance torque.

The actor, in contrast, only receives proprioceptive information augmented by an

action history:

$$\mathcal{O}_a = \{\mathbf{p}, R, \mathbf{v}, \boldsymbol{\omega}, H\},$$

where H contains the last N_H executed actions. Following [7], we set $N_H = 32$. This design improves robustness in the presence of motor delays while remaining feasible for real-world deployment, since past actions can be stored without additional sensors.

However, since at the time of writing the SKRL library does not support assigning different observation sets to the actor and the critic, in this work we opted not to employ the asymmetric actor–critic scheme. Instead, both networks are provided with the same observation array as the actor, i.e., without access to motor speeds or artificially injected disturbances. In practice, this limitation did not prove detrimental, as the learned policy still exhibited reliable performance during deployment.

The action space is defined as the set of motor speed setpoints, expressed in revolutions per minute (RPM):

$$\mathcal{A} = \{\omega_{sp,1}, \omega_{sp,2}, \omega_{sp,3}, \omega_{sp,4}\},$$

where each $\omega_{sp,i}$ specifies the desired RPM of motor i. In contrast to higher-level abstractions such as thrust or body rates, this low-level formulation requires the policy to directly handle motor dynamics and the nonlinear mapping between RPM and thrust. While this makes the learning problem harder, it eliminates the need for additional low-level controllers and enables a fully end-to-end learned policy.

3.2 Environment Setup

The training environment consists of a single quadrotor tasked with reaching and stabilizing at a fixed reference position. In this work, the reference is chosen at $(0, 0, \alpha)$ with $\alpha > 0$ rather than the origin, in order to avoid explicit ground–contact handling. At the beginning of each episode, the quadrotor is initialized at random positions and orientations within a bounded workspace and is required to converge to the target and hover in its vicinity (see Fig. 3.1).

The simulated platform is based on the Crazyflie 2.1+, a revision of the Crazyflie 2.0 whose use is widespread in research and education due to its low cost and flexibility [74]. The model of the drone is provided directly by NVIDIA's IsaacLab in the form of a USD (Universal Scene Description)[1] file, which specifies geometry, collision shapes, and rigid-body properties. The main physical parameters used in the simulation are summarized in Table 3.1.

Although the IsaacLab USD model is structurally correct, we observed that its default inertia matrix did not match the values reported for the Crazyflie in [7] and was approximately an order of magnitude larger. While policies trained with this default inertia

[1] `https://openusd.org`

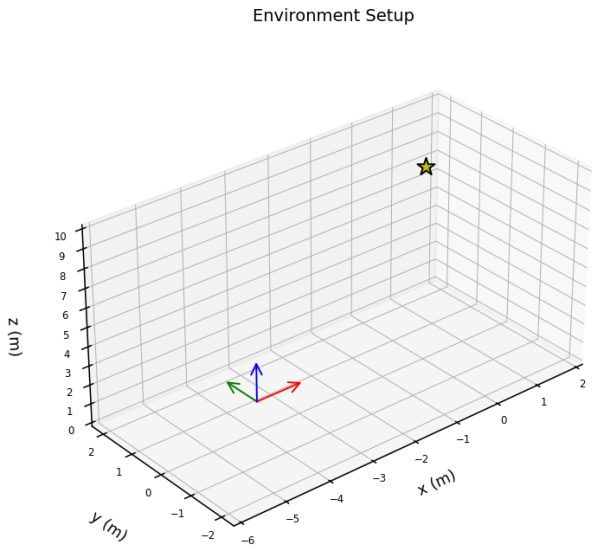


Figure 3.1: Schematic of the simulation environment: the quadrotor is initialized at a random position and must reach and stabilize at the target reference (the star) $(0, 0, \alpha)$ above the ground plane.

Parameter	Value
Mass m	0.027 kg
Arm length l	0.028 m
Inertia J	$\text{diag}(3.85\times10^{-6}, 3.85\times10^{-6}, 5.9675\times10^{-6})$ kg $\cdot$ m^2

Table 3.1: Physical parameters of the Crazyflie model used in simulation.

performed well in simulation, they exhibited strong oscillatory behavior and poor damping when deployed on the real platform. Replacing the inertia with the values from [7] yielded policies that transferred much more stably to the real Crazyflie.

Control inputs are represented as normalized values in $[-1, 1]^4$, mapped to rotor speed setpoints $\omega_{sp,i} \in [0, 21702]$ RPM. The motor dynamics are modeled as a first-order low-pass filter

$$\dot{\omega}_i = \tfrac{1}{\tau}\,(\omega_{sp,i} - \omega_i), \qquad \tau = 0.15 \text{ s},$$

which captures the fact that rotor speeds cannot change instantaneously but instead converge gradually toward their commanded values. As anticipated in Section 3.1, this introduces an actuation delay between the issued action and its effect on the dynamics. On the Crazyflie hardware, the delay typically spans 5–25 control cycles. Including this filter in simulation discourages the policy from exploiting unrealistically instantaneous actuation and improves Sim2Real fidelity.

Rotor thrust is modeled as a quadratic function of the rotor speed in RPM:

$$T_i = c\,\omega_i^2, \qquad c = 3.16 \times 10^{-10},$$

where c was identified by the manufacturer[2] through a fit of thrust–RPM measurements.

The individual thrusts, acting along the body $+z$ axis, are combined into the total body force:

$$\mathbf{F} = \sum_{i=1}^{4} T_i \mathbf{e}_z,$$

where $\mathbf{e}_z$ denotes the body $+z$ axis.

Each rotor also generates a torque contribution from two effects. The first is aerodynamic drag, proportional to thrust:

$$\boldsymbol{\tau}_{\text{drag},i} = k_T \, T_i \, \sigma_i \, \mathbf{e}_z, \qquad k_T = 0.005964552, \;\; \sigma_i \in \{-1, 1\},$$

where σ_i encodes the spin direction of the rotor.

The second is the lever-arm effect, due to the offset of each rotor from the body center:

$$\boldsymbol{\tau}_{\text{arm},i} = \mathbf{r}_i \times (T_i \mathbf{e}_z).$$

Finally, the total body torque is obtained by summing all contributions:

$$\boldsymbol{\tau} = \sum_{i=1}^{4} \big(\boldsymbol{\tau}_{\text{drag},i} + \boldsymbol{\tau}_{\text{arm},i} \big).$$

The environment is implemented as a custom IsaacLab task following the standard reinforcement learning API based on OpenAI Gym / Gymnasium [75]. The physics of the simulation are handled by PhysX[3], a high-performance rigid-body engine optimized for GPU execution, configured to run at 100 Hz. While control loops are often executed at lower rates, the Crazyflie requires updates at 100 Hz (see 3.4), so in our setup the control loop is executed at the same frequency as the physics. Thus, at each simulation step, actions are clipped, mapped to rotor setpoints, integrated through the motor model, and applied as external forces and torques.

To accelerate data collection, this custom environment is instantiated 4096 times in parallel within the simulation. This number was selected after empirical tuning, as it provided the best compromise between GPU utilization and training stability, while further increasing the number of environments offered no additional benefits. All instances run concurrently on a single NVIDIA GeForce RTX 5070 GPU, which allows IsaacLab to fully exploit the GPU pipeline and generate large amounts of experience data in about 3 minutes for 10,000 environment steps.

[2] `https://www.bitcraze.io/2015/02/measuring-propeller-rpm-part-3/`

[3] `https://physics-playground.github.io/PhysX5/physx/5.3.1/`

3.2.1 Episode Termination and Reset Policy

To ensure stable and realistic training, the simulation enforces safety boundaries and resets the environment whenever the quadrotor leaves the desired operating region. This prevents the agent from exploring physically implausible states and keeps the collected data within a useful range for learning.

An episode can end for two reasons: reaching the maximum allowed duration, or violating the position-based safety constraint. The maximum episode length is set to 500 steps, which at the simulation timestep of 0.01 s corresponds to 5 s of simulated time. Additionally, if the position relative to the environment origin exceeds the threshold p_{thr}, the episode is terminated. These conditions prevent the drone from drifting indefinitely far from the target or remaining in unstable configurations where recovery is unlikely and little meaningful reward can be collected, which would otherwise slow down the learning process. The termination parameters are summarized in Table 3.2.

Parameter	Value	Units	Description
$T_{\max}$	500	steps	Max episode length
p_{thr}	0.6	m	Max position error

Table 3.2: Termination parameters used during training.

When an environment ends, it is reset by sampling a new initial state from a predefined randomization range. Randomizing the initial state is essential to prevent overfitting to a narrow set of starting conditions and to improve the robustness of the learned policy. If the agent were always initialized near the target, it could learn behaviors that only work locally and fail to generalize to other parts of the state space.

With probability p_{guide}, the quadrotor is spawned directly at the target position with zero velocities and upright orientation. This occasional "guided" reset, inherited from [7], helps the agent experience near-goal states more frequently and stabilizes learning in the early stages. Otherwise, the initial state is randomized around the target: the position is sampled from a 3D uniform range $\mathbf{p}_0 \sim \mathcal{U}([-p_{\max}, p_{\max}]^3)$ and then shifted to the current environment origin, the orientation is sampled uniformly on SO(3) until the rotation angle from identity does not exceed $\theta_{\max}$, and the linear and angular velocities are sampled as $\mathbf{v}_0 \sim \mathcal{U}([-v_{\max}, v_{\max}]^3)$ and $\boldsymbol{\omega}_0 \sim \mathcal{U}([-\omega_{\max}, \omega_{\max}]^3)$. The sampling ranges used during training are reported in Table 3.3.

To avoid unrealistic transients at the beginning of an episode, the policy's action-history buffer is pre-filled with the normalized command corresponding to the midpoint between the minimum and maximum motor speeds,

$$\boldsymbol{\omega}(0) = \tfrac{1}{2}(\omega_{\min} + \omega_{\max})\mathbf{1}.$$

Parameter	Value	Units	Description
$p_{\max}$	0.2	m	Max initial position per axis
$\theta_{\max}$	1.57	rad	Max initial rotation angle from identity
$v_{\max}$	1.0	m/s	Max initial linear velocity per axis
$\omega_{\max}$	1.0	rad/s	Max initial angular velocity per axis
p_{guide}	0.1	–	Probability of goal reset

Table 3.3: Reset initialization randomization ranges.

3.2.2 Reward Function

The reward function is designed to drive the agent toward stable hovering at the target position while discouraging unnecessary motion and excessive control effort. Because the policy is trained from scratch and starts with random actions, the agent initially cannot stabilize itself. Providing a well-shaped reward is therefore essential to guide learning toward upright and steady flight before optimizing for accuracy.

At each timestep, the reward penalizes deviations from the desired state. The two dominant contributions are the position error and the attitude error. The position term penalizes the squared distance from the target, while the attitude term penalizes tilting of the drone's body away from the world vertical axis. These components carry the largest weights and are the primary drivers of early training progress, encouraging the agent to first learn basic hovering behavior.

Two lighter penalties refine the behavior: a small cost on the body-frame linear velocity discourages rapid and unstable motions, and a small cost on the difference between the current action and a nominal hover command regularizes control outputs. The latter helps reduce command noise once the drone reaches a near-hover state, making the learned policy smoother and more stable.

Formally, the reward at each step is defined like this, in according to [7]:

$$r(\mathbf{s}, \mathbf{a}) = C_{rs} - \Big(C_{rp}\|\mathbf{p}\|_2^2 + C_{rq}(1 - q_w^2) + C_{rv}\|\mathbf{v}_b\|_2^2 + C_{ra}\|\mathbf{a} - C_{rab}\|_2^2 \Big),$$

where $\mathbf{p}$ is the position relative to the target, q_w is the scalar part of the quaternion orientation, $\mathbf{v}_b$ is the body-frame linear velocity, and $\mathbf{a}$ are the normalized actions. The constant C_{rs} serves as a survival bonus: it adds a positive baseline to each timestep's reward, which helps prevent the "learning to terminate" problem [76]—where an agent might learn to deliberately crash or end episodes early to avoid further negative rewards. By ensuring that rewards remain positive when the drone stays near the target, C_{rs} makes prolonged stable flight more rewarding than terminating quickly, encouraging the agent to remain airborne and continue improving its behavior.

Finally, it is worth noting that in [7], the reward weights are modified through a cur-

Constant	Value	Description
C_{rs}	2.0	Survival bonus (base reward offset)
C_{rp}	5.0	Position error weight
C_{rq}	5.0	Orientation (tilt) error weight
C_{rv}	0.01	Linear velocity weight
C_{ra}	0.01	Action deviation weight
C_{rab}	0.334	Nominal hover command

Table 3.4: Constants used in the reward formulation.

riculum in which, every 100,000 steps, the penalties on position error, linear velocity, and action deviation are gradually increased, while the orientation term remains constant. This procedure facilitates the acquisition of basic stabilization skills under loose constraints before refining the policy toward accurate hovering and efficient control. In the present work, the curriculum is omitted, as the resulting policies exhibited satisfactory performance without it.

3.3 Training Setup

The quadrotor is controlled by a deep neural network trained through reinforcement learning. The network is instantiated and trained using the SKRL library, which interfaces with the IsaacLab simulator to collect experience from multiple parallel environments. SKRL was chosen over other reinforcement learning libraries for its readability, simplicity, and flexibility, as well as for its transparent implementations of many popular state-of-the-art reinforcement learning algorithms [73]. In addition, its highly modular design makes it straightforward to swap out individual components—such as trying different algorithms—without major changes to the overall training setup.

3.3.1 Neural Network Architecture

The agent is implemented using two neural networks, representing the policy (actor) and the value function (critic). Both networks share the same architecture: fully connected multilayer perceptrons with two hidden layers of 64 units each and hyperbolic tangent activations.

As discussed in Section 3.1, in contrast with [7] we did not employ an asymmetric actor–critic scheme, so both networks are provided with the same observation array as the actor. The shared input consists of the quadrotor position, orientation (represented as a rotation matrix), body-frame linear and angular velocities, and a 32-step action history buffer, resulting in an input dimension of 146. While [7] identified the asymmetric actor–critic scheme as a key element for achieving successful sim-to-real transfer, our

experiments show that reliable flight was still attainable without it, as further discussed in Chapter 4.

Regarding the outputs, the actor network produces a 4-dimensional action vector in $[-1, 1]$, which is linearly mapped to rotor speed setpoints (RPM) and passed through the motor dynamics model before being applied as forces and torques in the simulator. The critic network outputs a single scalar value estimating the current state.

3.3.2 Training Loop

The training procedure is implemented using the Proximal Policy Optimization (PPO) algorithm through the SKRL library, which manages the collection of experience from the parallel simulation environments running in IsaacLab. The main hyperparameters used for training are reported in Table 3.5.

Hyperparameter	Value
Rollout length	96 steps
Learning rate	5×10^{-4}
Discount factor γ	0.99
GAE parameter λ	0.95
Entropy regularization	0.0
Importance ration clipping ϵ	0.2

Table 3.5: Main PPO training hyperparameters.

Training is continued for a total of 10 000 timesteps per environment since training beyond this point yielded no further improvements. With 4096 parallel environments, this corresponds to approximately 41 million environment–agent interactions, which was found sufficient for the policy to converge to stable hovering behavior. Moreover, since the simulation runs at 100 Hz and each simulation step corresponds to one reinforcement learning timestep, this amounts to about 114 hours of equivalent real-world flight time.

3.4 Deployment

The trained policy network was deployed on the Crazyflie 2.1+ platform by integrating it into the onboard firmware as a custom controller. The network was first converted into embedded-compatible C code using STEdgeAI-Core[4], a tool provided by STMicroelectronics for optimizing and compiling machine learning models for various microcontrollers, including the STM32F405 used on the Crazyflie. STEdgeAI-Core supports several popular machine learning frameworks: TensorFlow and Keras models are supported natively,

[4] `https://www.st.com/en/development-tools/stedgeai-core.html`

while PyTorch models can be used via the ONNX[5] format, an open standard for representing machine learning models. Accordingly, the PyTorch checkpoint of the best policy was exported to ONNX and then imported into STEdgeAI-Core through its command-line interface, which automatically generated optimized C code suitable for the target hardware. Among the available optimization modes and optional quantization settings, the balanced optimization mode was selected without quantization, prioritizing numerical fidelity over memory footprint to preserve control accuracy.

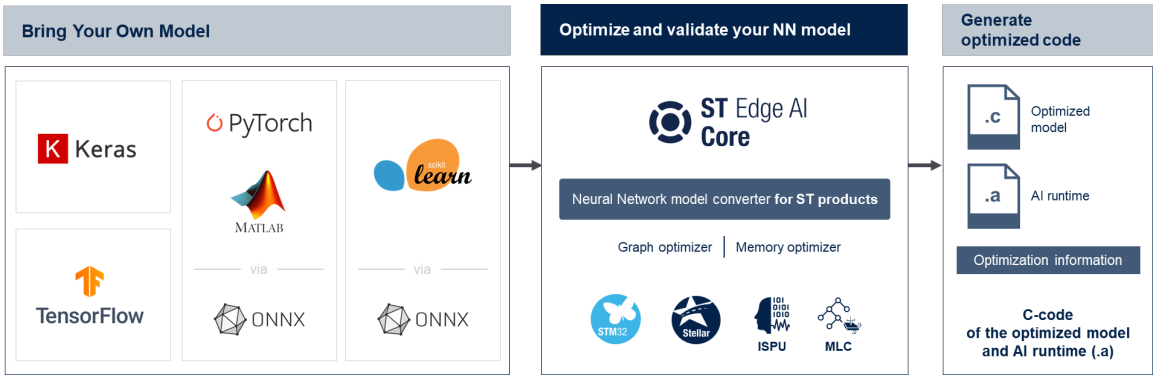


Figure 3.2: Deployment pipeline using STEdgeAI-Core

The generated static library and header files were integrated into the Crazyflie firmware as an out-of-tree (OOT) controller module. OOT modules are self-contained folders with their own build scripts that plug into the firmware build system without modifying its core. This mechanism allows replacing the built-in controller with a custom implementation, provided that it exposes the standardized controller interface (`init`, `test`, `update`). This modular approach simplifies maintenance and preserves compatibility with future upstream firmware updates.

At runtime, the custom controller preserves the same input–output interface used during simulation (see Fig. 3.3). At each control cycle, the onboard state estimator provides the drone state, which is converted into the observation vector defined during training. This vector is composed of the drone's position, orientation (expressed as a rotation matrix), as well as its linear and angular velocities. The vector is then passed to the embedded neural network for inference, producing normalized actions. These actions are mapped to motor commands using the same thrust and torque parameters employed in simulation. By default, the standard Crazyflie controller outputs higher-level commands such as collective thrust and body rates (CTBR). To enable direct motor-level control, the default motor mixing stage was bypassed, allowing the controller to convert RPM setpoints into PWM signals, which are then fed directly to the motors.

The execution of the control stack follows the real-time scheduling constraints of the Crazyflie firmware. The main control loop, which handles sensor acquisition and motor

[5] `https://onnx.ai/`

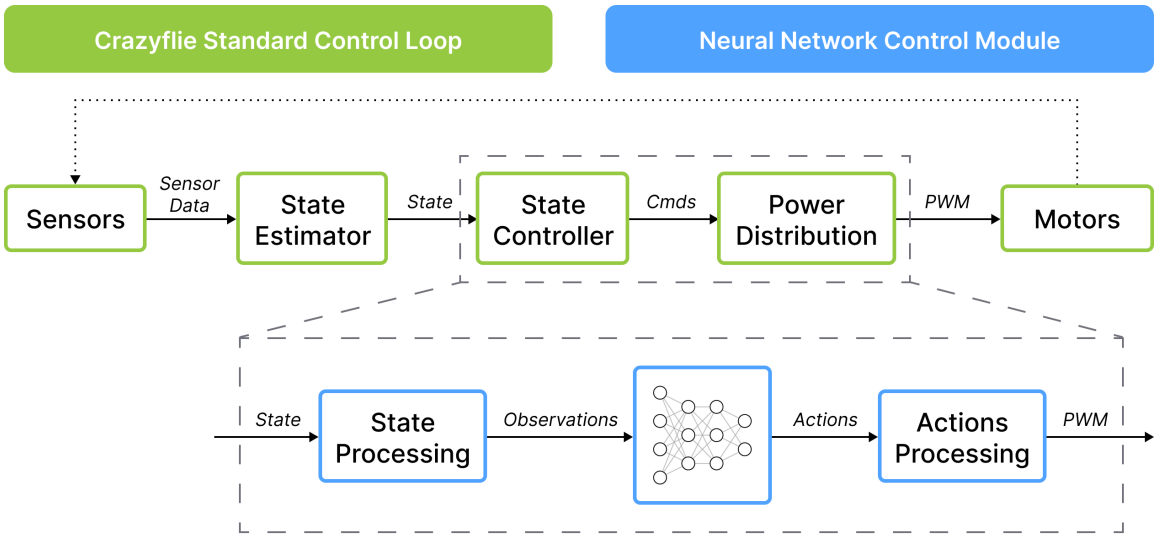


Figure 3.3: Standard controller stack of the Crazyflie firmware. The neural network module replaces the default controller and the power distribution stage and directly outputs motor commands.

actuation, runs at a fixed frequency of 1 kHz to minimize latency and maintain flight stability. Computationally heavier components, such as the state controller or the neural network inference, operate at a lower rate of approximately 100 Hz. To prevent overruns, the firmware permits these modules to skip updates when necessary, ensuring that the 1 kHz loop remains uninterrupted. This design balances responsiveness with computational feasibility, allowing the neural policy to be executed reliably within the resource-constrained onboard environment.

Preserving strict equivalence between simulation and firmware ensures that the policy encounters identical inputs and produces outputs in the same format across domains, enabling a seamless transfer from training in simulation to execution on the real platform.

Chapter 4

Results

This chapter presents the experimental results obtained in both simulation and real-world deployment. We first analyze training and evaluation curves to assess convergence of the proposed policies. We then report real-world experiments on the Crazyflie platform, covering hover stability, position control, and trajectory tracking, and compare our approach with the state-of-the-art *Learning to Fly in Seconds* baseline [7]. An ablation study follows, highlighting the importance of rotor delay modeling and action history for successful sim-to-real transfer. Finally, we discuss additional considerations emerging from our experiments, including the influence of rollout length during training and the impact of battery voltage on flight performance.

4.1 Training Results

Policies were trained with ten different random seeds on the position control task described in Chapter 3. Figure 4.1 reports the learning progress in terms of episode return and episode length. During the initial training phase, both the mean return and the variance across seeds increased, after which they reached a stable plateau. This behavior is a clear indication of convergence. The episode length stabilized slightly below the 500-step environment horizon (around 420–440 steps), while the returns converged near 700. Since each step can yield at most a reward of 2, the theoretical maximum return over a full 500-step episode is 1000. The fact that the learned policies approach values close to this bound indicates that failures were effectively eliminated after convergence.

Although the training curves already suggest consistent progress, Figure 4.2 shows the corresponding evaluation metrics. Unlike the training statistics, which include exploration noise and early terminations, evaluation runs execute the current policy deterministically from randomized initial states. As a result, the evaluation episode length consistently reached the full 500-step horizon, and the returns stabilized around 780 with limited variance across seeds. This confirms that the learned policies reliably maintained position until timeout.

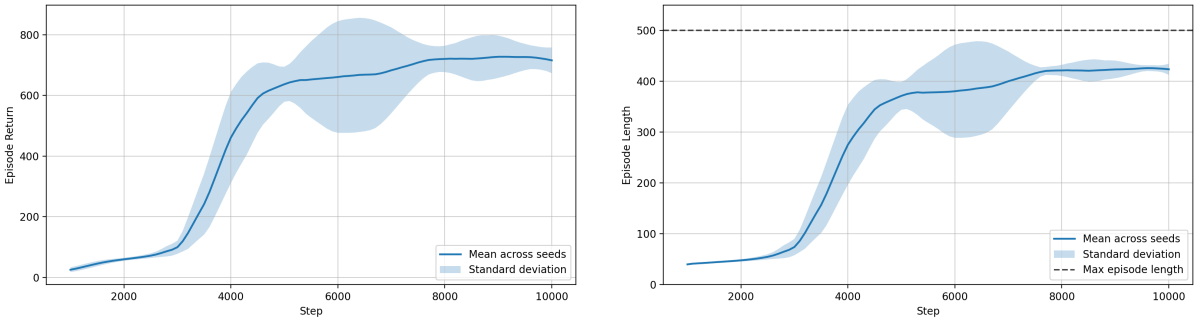


Figure 4.1: Training curves across ten independent seeds, showing mean and standard deviation. Smoothing applied with a moving average of window size 10.

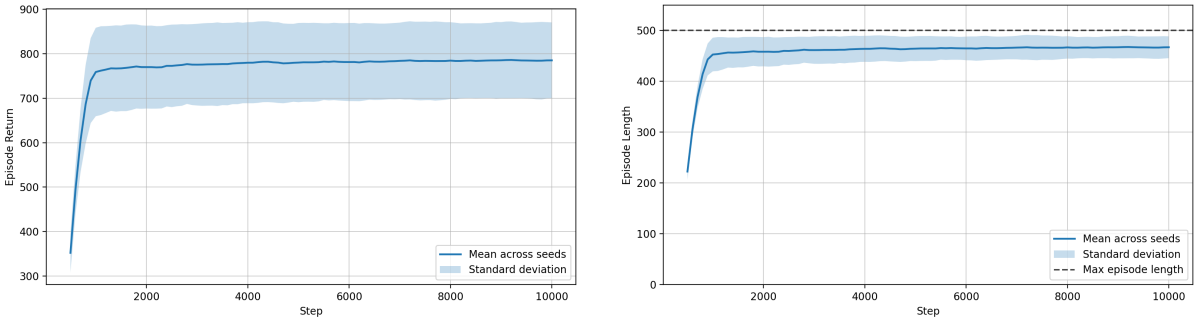


Figure 4.2: Evaluation curves across ten seeds, computed without exploration noise. The episode length consistently reaches the 500-step horizon. Smoothing applied with a moving average of window size 5.

4.2 Real-world experiments

We deployed the policies trained in simulation on the Crazyflie 2.1+ nano drone, equipped with the Flow Deck v2[1], to assess their performance in real-world conditions. The Flow Deck v2 is an expansion board that integrates an optical flow sensor with a downward-facing time-of-flight (ToF) module, providing position and velocity estimates. Its use was mandatory, since all experiments were conducted without a motion-capture system and therefore relied exclusively on the onboard state estimator for feedback and logging. Compared to motion capture, this sensing setup is inherently noisier, particularly in the position estimates dominated by the optical flow sensor.

To reduce this gap, we retrained their policy according to the specifications reported in their work and tested it under the same onboard-sensing conditions adopted in this thesis, using the same sim-to-real mitigations (action history and rotor delay modeling). Since our experiments were conducted without a motion-capture system, the results are not directly comparable to those reported in [7]. To enable a fair comparison, we reproduced their policy within our testing setup and configuration using the same sim-to-real mitigations (action history and rotor delay modeling). Two variants of the [7] policy

[1] `https://www.bitcraze.io/products/flow-deck-v2/`

were considered, hereafter named L2F (300k) and L2F (3M), corresponding to checkpoints trained for 300,000 and 3,000,000 steps respectively. For each policy, only one seed (seed 0) was evaluated, since the original paper reports consistent performance across seeds and explicitly avoids cherry-picking.

Two sets of experiments were carried out. The first focused on hover stability and position control, testing whether policies could take off and remain near a fixed target position for an extended duration. The second addressed trajectory tracking, evaluating the ability of the policies to follow a predefined Lissajous curve at different speeds.

4.2.1 Hover stability and position control

We evaluated hover stability and position control by testing all deployed policies (our 10 seeds plus the L2F (300k) and L2F (3M) baselines) in real flight. For each policy, we performed three runs of at least 20 seconds, during which we manually commanded the drone to take off and hover at 0.5 m above the ground. We chose this duration because, based on our experience and prior work, performance after about 20 s is representative of the controller's overall behavior.

Due to telemetry bandwidth limitations, we logged different signals in different runs: some recorded position and velocity, others position and attitude, and others position together with motor commands. This ensured complementary data while staying within the constraints of the radio link.

All policies were able to take off and sustain stable flight in all runs, even without explicitly modeling well-known phenomena such as ground effect and near-ground aerodynamics, which typically complicate the initial ascent of micro aerial vehicles.

We summarize the aggregated position error metrics in Table 4.1. Position error is computed in the horizontal plane (XY only). For our policies, we first computed metrics per seed as the mean across its three test runs, and then summarized them across seeds (mean, median, and minimum). For the "best seed" and for the L2F baselines, we computed the values directly from their three test runs.

Policy	e_{mean} [m]	e_{med} [m]	e_{min} [m]
Ours (10 seeds)	0.27	0.23	0.11
Ours (best seed = 7)	0.11	0.10	0.09
L2F baseline (300k)	0.36	0.34	0.33
L2F baseline (3M)	0.13	0.12	0.10

Table 4.1: Aggregated position error metrics for hover stability. We compute the mean (e_{mean}), median (e_{med}), and minimum (e_{min}) of the position error in the horizontal plane (XY only). For "Ours (10 seeds)", we first compute metrics per seed across its three test runs, then aggregate them across seeds. For "best seed" and the L2F baselines, we compute metrics directly from their three runs.

As shown in Table 4.1, our policies achieve reliable position control across all seeds. The average horizontal error of 0.27 m demonstrates consistent hover capability, while the minimum error of 0.11 m indicates that some seeds attained very precise control. On average, our policies obtain an error that lies between the two L2F baselines, which indicates comparable performance with the state-of-the-art. Furthermore, our best seed performs on par with the L2F (3M) baseline despite the difference in training setup.

It is also important to consider the training budgets. Our PPO policies used about 40 million environment steps, whereas the L2F baselines used 300,000 and 3,000,000 steps respectively. This higher value is expected for PPO, given that it is an on-policy algorithm and therefore less sample-efficient than TD3, which underlies the L2F approach.

Beyond aggregated error values, it is also useful to examine hover stability over time. We present mid-flight values, since they provide a more representative picture of controller behavior than full-trajectory statistics, which are dominated by the noisy take-off phase. Figure 4.3 reports the altitude of a representative seed of our policy, compared with the L2F baselines. We selected this seed because its performance is aligned with the average metrics reported in Table 4.1. Each curve shows the mean altitude across three runs of the same policy, with shaded areas denoting the standard deviation. All controllers maintained flight close to the 0.5 m target with comparable variability over time, in line with the position-error metrics.

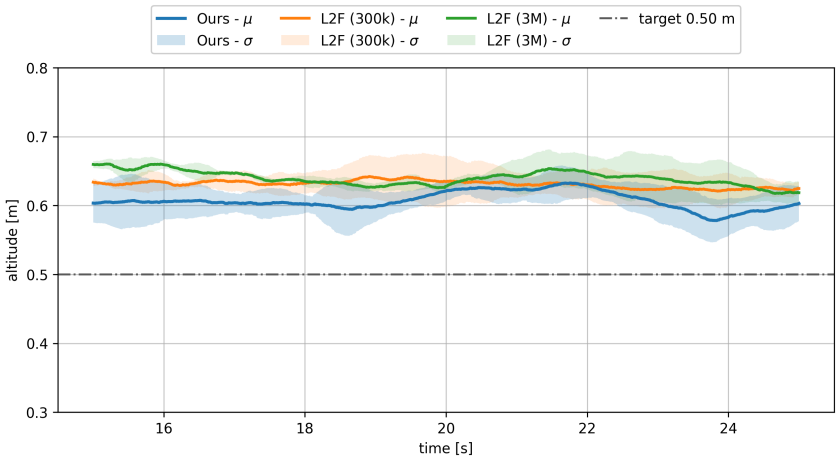


Figure 4.3: Mid-flight altitude during hover stability experiments for our policy and the L2F baselines. The dashed line marks the 0.5 m target. μ and σ denote the mean and standard deviation across three runs of the same policy.

To further probe stability, Figure 4.4 shows the corresponding attitude behavior for the same policies. Roll and pitch remain close to zero across all controllers, indicating stable hover. Yaw is included for completeness: no yaw reference was provided during training, so policies were free to adopt different orientations; nevertheless, yaw stayed bounded without signs of uncontrolled drift or rotation.

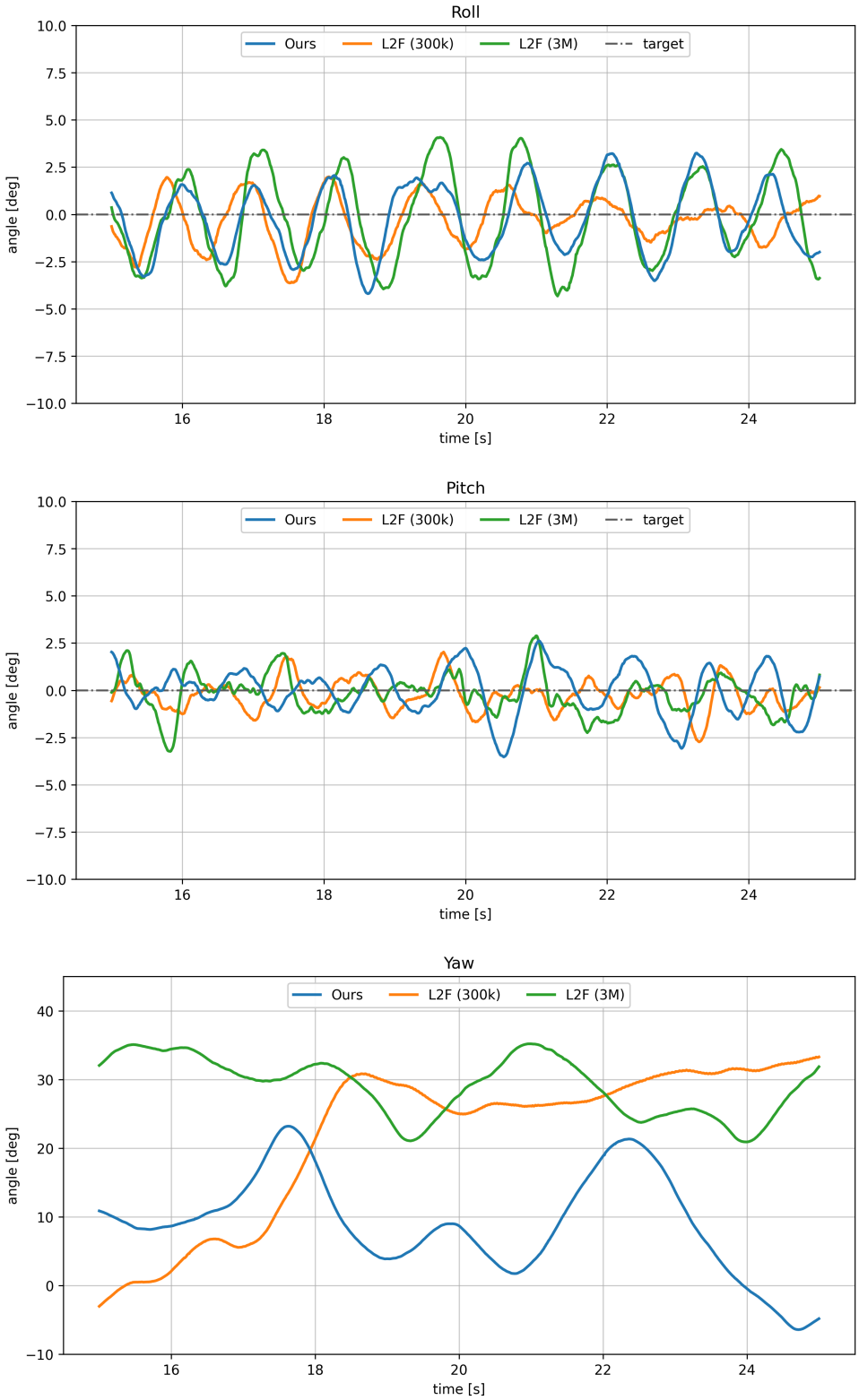


Figure 4.4: Mid-flight attitude (roll, pitch, yaw) during hover stability experiments for our policy and the L2F baselines.

4.2.2 Trajectory tracking

The second real-world experiment evaluated the ability of the trained policies to follow a predefined trajectory. This is feasible because the controller minimizes position error with respect to a reference; by varying the reference over time, the drone is induced to trace a path. We adopted the figure-eight Lissajous curve from [7],

$$p(t) = \begin{bmatrix} \cos(2\pi t/T) \\ \frac{1}{2}\sin(4\pi t/T) \\ \alpha \end{bmatrix},$$

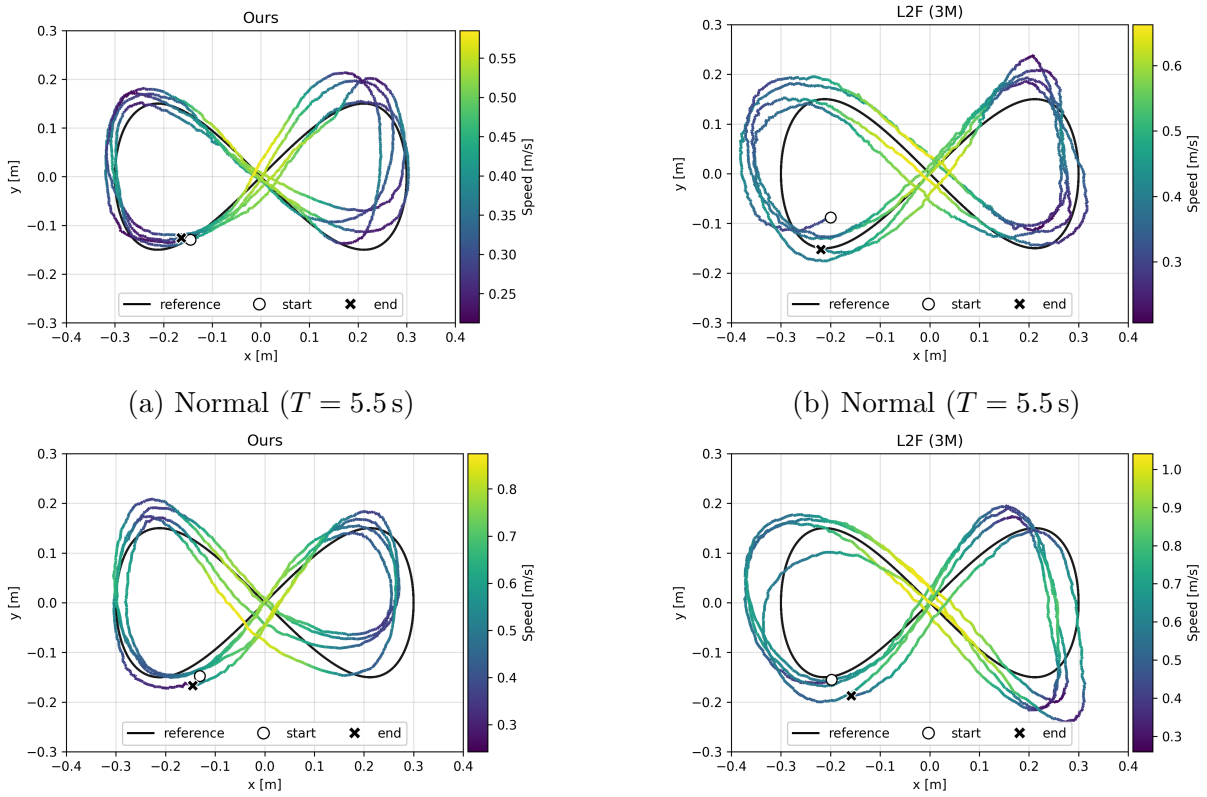
with constant altitude $\alpha = 0.5\,\text{m}$ and two cycle times: $T = 5.5\,\text{s}$ (normal) and $T = 3.5\,\text{s}$ (fast). A run was considered successful if at least four full cycles were completed without a crash. As in the hover experiment, ten seeds were tested for our policies. For comparison we used only the retrained L2F (3M) baseline, since L2F (300k) performed very poorly on this task.

To fit the available flight space, the Lissajous trajectory was scaled by about 30% relative to the original in [7]. The quantitative results are reported in Table 4.2. For our policies, the metrics correspond to the best-performing seed for this task. Overall, our policies achieve tracking errors broadly comparable to the L2F (3M) baseline in both timing regimes. Compared to the hover task, the tracking errors remain within the same range. Regarding success rate, 9 out of 10 seeds completed at least four cycles in the normal case, while 7 out of 10 succeeded in the faster cycle.

Interval	Normal ($T = 5.5\,\text{s}$)		Fast ($T = 3.5\,\text{s}$)	
Policy	$\bar{e}_{xy}$ [m]	$\bar{e}$ [m]	$\bar{e}_{xy}$ [m]	$\bar{e}$ [m]
Ours (best seed = 4)	0.27	0.22	0.27	0.23
L2F (3M)	0.24	0.21	0.26	0.23

Table 4.2: Trajectory tracking performance on the normal and fast Lissajous trajectories. $\bar{e}$ denotes the root mean square error (RMSE) including the vertical dimension z, while $\bar{e}_{xy}$ denotes the RMSE restricted to the horizontal plane. Metrics for our policy correspond to the best-performing seed for this task.

Figure 4.5 compares the tracked trajectories to the reference, using the best-performing seed of our policy (the same reported in Table 4.2) and the L2F (3M) baseline for both cycle times. The color encodes instantaneous speed. Differences in speed distribution across the path reflect the varying control effort required by the timing: segments that induce higher speeds demand faster corrections in orientation and thrust to keep the drone aligned with the reference. Despite this, both controllers reproduce the figure-eight with good fidelity, indicating robust tracking under time-varying references.



(a) Normal ($T = 5.5\,\mathrm{s}$)

(b) Normal ($T = 5.5\,\mathrm{s}$)

(c) Fast ($T = 3.5\,\mathrm{s}$)

(d) Fast ($T = 3.5\,\mathrm{s}$)

Figure 4.5: Reference (black) and tracked Lissajous trajectories, colored by instantaneous speed. Panels show our policy and the L2F (3M) baseline under the normal ($T = 5.5\,\mathrm{s}$) and fast ($T = 3.5\,\mathrm{s}$) timings.

4.3 Ablation Study

We carried out an ablation study in order to assess the contribution of specific design choices in the simulation environment. In particular, we investigated the effect of modeling the rotor delay and of including the action history in the policy input. Both elements were hypothesized to play a central role in bridging the simulation–to–reality gap by making the simulated dynamics closer to the real platform and by providing the policy with sufficient information to cope with the discrepancies between the approximate simulation model and the real-world platform.

Table 4.3 summarizes the outcome across ten random seeds for each configuration. For each policy, the number of successful flights (defined as stable hovering for at least 20 s, as explained in Section 4.2.1) is reported, together with the mean, median, and minimum position error of the successful seeds. Results are compared against the baseline configuration including both rotor delay and action history.

When rotor delay was not included, none of the ten trained policies were able to achieve flight. This highlights the importance of modeling actuator dynamics: in simulation, the policy learns under the assumption that motor commands are applied instantaneously,

Configuration	Successful seeds	e_{mean} [m]	e_{med} [m]	e_{min} [m]
Baseline (delay + history)	10/10	0.42	0.37	0.28
No rotor delay	0/10	∞	∞	∞
No action history (10k)	0/10	∞	∞	∞
No action history (30k)	9/10	0.76	0.74	0.65

Table 4.3: Ablation study across ten seeds. A successful flight is defined as stable hovering for at least 25 s around the target point. Reported errors are RMSE in XYZ. The baseline includes both rotor delay and action history. In the other rows, the description indicates the component that was removed, while the other one was kept.

while in reality thrust generation is subject to lag due to motor and propeller inertia. As a result, the controller overfits to an idealized actuation model that does not transfer to reality, leading to immediate flight failure. Importantly, the training curves (Figure 4.6) confirm that the policies appeared to converge in simulation, indicating that this failure is not due to lack of training but to the actuation mismatch.

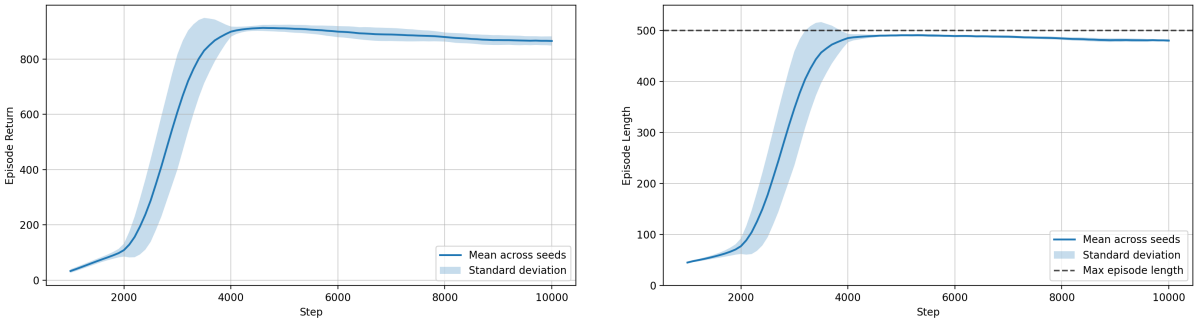


Figure 4.6: Training curves (mean return and mean episode length across seeds) for policies trained without rotor delay. Despite apparent convergence in simulation, none of the policies were able to fly in reality.

When action history was removed, policies trained for 10,000 steps exhibited unstable behavior, characterized by a continuous increase in altitude without stabilization at the target reference. The controllers were unable to regulate thrust around the hover point, which ultimately led to a crash. This persistent climbing behavior is inherent to the partial observability of the system, and is particularly critical during the take-off phase when thrust regulation is most demanding. Extending training to 30000 steps allowed several seeds to achieve flight, but the underlying limitations remained. These policies were able to hover, though with significantly reduced accuracy and stability compared to the baseline that included both rotor delay and action history, especially regarding the reference altitude. For this reason, in this ablation study position errors were computed as RMSE in XYZ rather than restricted to the horizontal plane: excluding the vertical component would have masked the instability. To conclude, Figure 4.7 illustrates the larger roll and pitch oscillations, while Figure 4.8 compares the motor profiles, highlighting

the strong tendency to saturate the motors as a direct consequence of aggressive thrust corrections in the absence of action history.

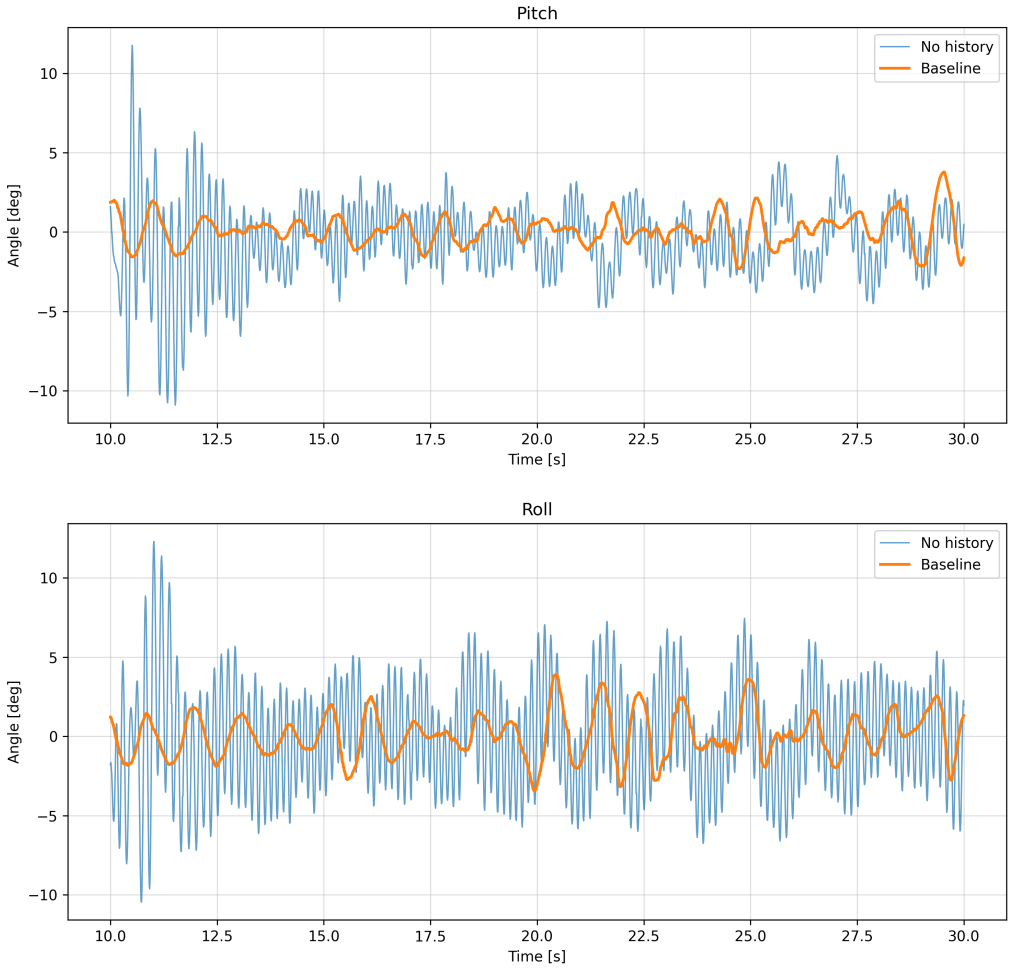
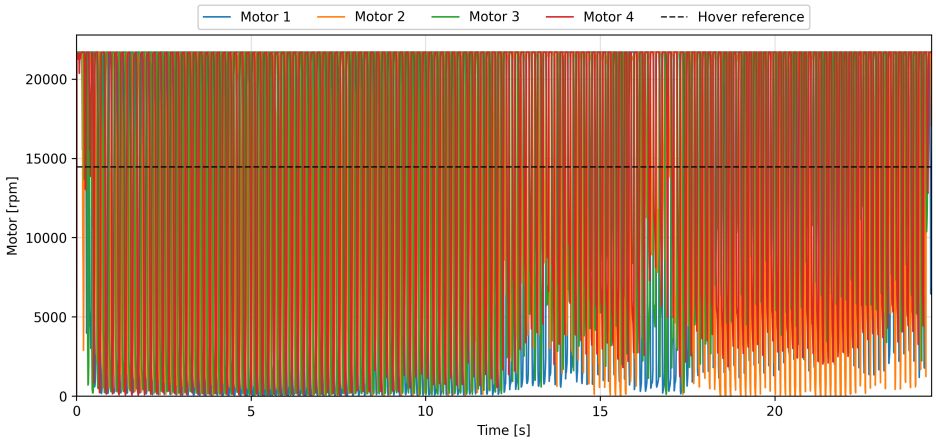
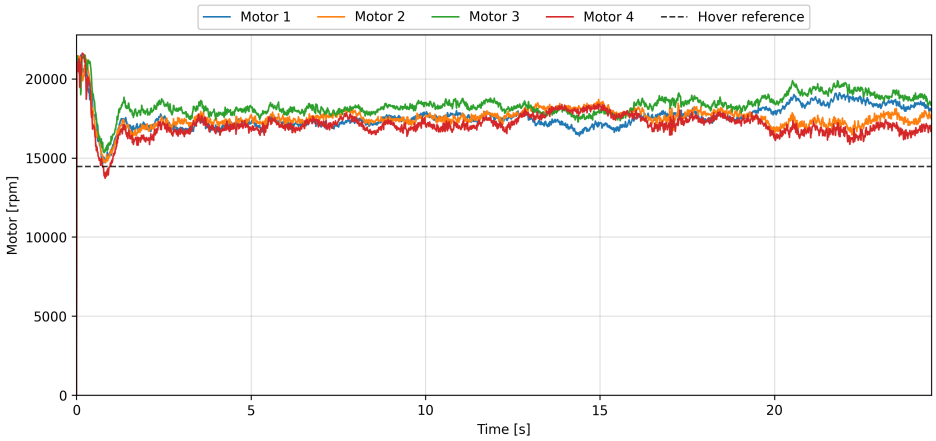


Figure 4.7: Comparison of roll and pitch angles during hover for the baseline policy and for a policy trained without action history. The latter exhibits stronger oscillations, reflecting reduced stability.

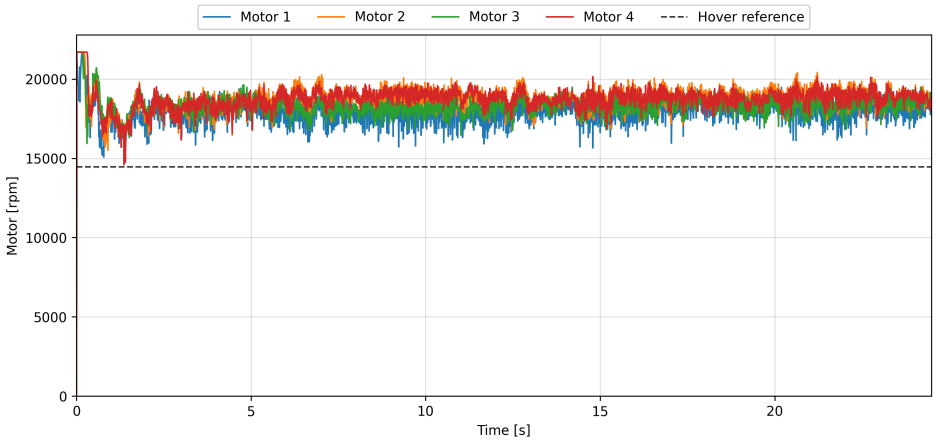
Overall, this ablation study highlights the central role of actuator modeling and action history in achieving reliable sim-to-real transfer on the Crazyflie. Rotor delay modeling was critical. Action history, while not strictly necessary for flight, markedly improved stability and prevented motor saturation.



(a) No action history (30000 steps)



(b) Our baseline (action history and rotor delay)



(c) L2F (3M) policy

Figure 4.8: Motor profiles for a policy trained without action history, our baseline (with rotor delay and action history) and the L2F (3M) policy. The hover reference indicates the theoretical motor command that would yield perfect hovering under ideal conditions; it is shown for clarity, but it does not correspond to a trajectory generated by the Crazyflie firmware. The no-history case shows a tendency to saturate the motors, consistent with aggressive corrections.

4.4 Discussion

Overall, the results demonstrate that our policies can achieve performance comparable to the state-of-the-art TD3 baseline under the same onboard-sensing conditions. Both in hover stability and in trajectory tracking, the proposed controllers reached accuracy and reliability levels that are competitive with the retrained L2F baseline. This confirms that PPO, despite being an on-policy method, can serve as a viable alternative to TD3 for sim-to-real transfer on the Crazyflie.

While Eschmann et al. [7] also proposed additional sim-to-real mitigations such as curriculum learning and an asymmetric actor–critic scheme, these were not employed in our training setup. Exploring their integration with PPO represents a promising direction for future work, with the potential to further improve stability and robustness beyond the results reported here.

A second aspect worth discussing is the effect of rollout length. In PPO, rollouts define the horizon over which trajectories are collected before each policy update. Longer rollouts provide updates based on more samples, which helps the actor compensate for inaccuracies in the critic's value predictions by grounding its decisions in richer trajectory information. In our experiments, rollout length played a crucial role. Reducing it from 96 to 48 or 64 led to poor convergence, with both reward and episode length saturating at low values even after extended training. This is consistent with the expectation that shorter rollouts weaken the quality of the policy update, making training less stable. Conversely, increasing the rollout length to 128 initially appeared promising, with higher convergence values in both reward and episode length. However, the corresponding policies failed in deployment: instead of reaching the target altitude, they hovered close to the ground, where ground effect destabilized flight and led to crashes. Our interpretation is that these policies overfitted to the action baseline defined in the reward function (see Section 3.2.2), i.e., the reference motor command introduced to penalize excessive control effort. As a result, the controllers were biased toward this thrust level instead of adapting to the actual altitude requirement during deployment. Increasing the baseline shifted this behavior upward, but the underlying issue persisted. These observations suggest that rollout length critically shapes PPO training dynamics, but also highlight the risk of overfitting to the reward shaping, specifically the action baseline term, which can bias the controller toward maintaining a fixed thrust level rather than regulating altitude appropriately.

Finally, real-world experiments also highlighted the importance of battery effects. We observed that our policy struggled to maintain altitude once the battery voltage dropped below roughly 3.8 V, whereas the L2F baseline did not show the same sensitivity. This behavior is consistent with the physics of PWM-driven motors: as the battery voltage decreases, the same motor command produces less thrust, so higher commands are required to maintain flight. The Crazyflie firmware normally includes a battery-voltage compen-

sation mechanism that scales motor commands with the measured voltage to keep thrust approximately constant. Since we bypassed this mechanism and the policy was not trained with voltage variation, the loss of altitude is an expected outcome. For the L2F baseline, one possible explanation is that firmware-level compensation was active during deployment. Another hypothesis is that their policy reacted more strongly to altitude error and implicitly increased motor commands when the battery voltage dropped. Investigating this difference and implementing mitigation strategies is left for future work. Some possible directions include adding battery voltage to the observation space, randomizing thrust scaling in simulation to train policies that are robust to changes in motor efficiency, or re-enabling firmware compensation so that thrust remains consistent across battery levels.

Conclusions and Future Work

This thesis investigated the problem of learning-based low-level control for nano-drones, focusing on the Crazyflie 2.1 quadrotor. The main goal was to reproduce and extend the results of *Learning to Fly in Seconds* by Eschmann et al. [7], while adopting different tools and algorithms—IsaacLab as the simulation environment, SKRL as the training library, and Proximal Policy Optimization (PPO) in place of TD3. A complete experimental pipeline was developed, covering training in massively parallel simulation, evaluation across multiple seeds, and deployment on the Crazyflie platform.

The results demonstrate that policies trained under this framework can successfully transfer from simulation to reality, achieving stable hovering and reliable trajectory tracking. Importantly, the performance is comparable to the original baseline despite the use of a different algorithm and toolchain. This confirms that the core ideas of rotor-delay modeling, action history, and domain randomization are not tied to a specific simulator or algorithm, but can generalize across different setups. The work therefore contributes to validating and extending the reproducibility of sim-to-real learning for nano-drones.

At the same time, the experiments revealed that further progress is possible by refining the training process. In particular, the choice of rollout length strongly influenced convergence and deployment performance, highlighting the need for strategies that improve sample efficiency without overfitting to action baselines. Future work could explore the integration of curriculum learning or asymmetric actor–critic structures, which have been shown to stabilize training in related contexts, as well as techniques to account for actuator nonlinearities and battery effects.

Looking ahead, the proposed framework can be extended beyond the Crazyflie. Adapting the training pipeline to different types of drones, such as larger quadrotors, hexarotors, or octarotors, would help assess the scalability of the approach. Moreover, evaluating controllers in more complex scenarios, including obstacle avoidance or drone racing tasks that require passing through static or moving gates, would represent a step toward real-world applications. Such extensions may require adapting the reward function or neural network architecture, but they would significantly broaden the scope and impact of learning-based control on aerial robotics.

Bibliography

[1] J. Verbeke and J. D. Schutter. "Experimental maneuverability and agility quantification for rotary unmanned aerial vehicle". In: in: *International Journal of Micro Air Vehicles* 10 (1) (2018), pp. 3–11.

[2] H. Shakhatreh et al. "Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges". In: in: *IEEE Access* 7 (2019), pp. 48572–48634.

[3] G. Hoffmann et al. "Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment". In: *AIAA Guidance, Navigation and Control Conference and Exhibit*.

[4] D. Hanover et al. "Autonomous drone racing: A survey". In: in: *IEEE Transactions on Robotics* 40 (2024), pp. 3044–3067.

[5] E. Kaufmann et al. "Champion-level drone racing using deep reinforcement learning". In: in: *Nature* 620 (7976) (2023), pp. 982–987.

[6] W. Zhao, J. P. Queralta, and T. Westerlund. "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2020, pp. 737–744.

[7] J. Eschmann, D. Albani, and G. Loianno. "Learning to fly in seconds". In: in: *IEEE Robotics and Automation Letters* 9 (7) (2024), pp. 6336–6343.

[8] R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[9] J. Kober, J. A. Bagnell, and J. Peters. "Reinforcement learning in robotics: A survey". In: in: *The International Journal of Robotics Research* 32 (11) (2013), pp. 1238–1274.

[10] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[11] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement learning: A survey". In: in: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[12] I. Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[13] F. Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: in: *Psychological review* 65 (6) (1958), p. 386.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors”. In: in: *nature* 323 (6088) (1986), pp. 533–536.

[15] K. Hornik, M. Stinchcombe, and H. White. “Multilayer feedforward networks are universal approximators”. In: in: *Neural networks* 2 (5) (1989), pp. 359–366.

[16] A. Vaswani et al. “Attention is all you need”. In: in: *Advances in neural information processing systems* 30 (2017).

[17] H. Lee et al. “Unsupervised learning of hierarchical representations with convolutional deep belief networks”. In: in: *Communications of the ACM* 54 (10) (2011), pp. 95–103.

[18] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: in: *arXiv preprint arXiv:1412.6980* (2014).

[19] T. Tieleman and G. Hinton. “Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning”. In: in: *COURSERA Neural Networks Mach. Learn* 17 (2012), p. 6.

[20] J. Duchi, E. Hazan, and Y. Singer. “Adaptive subgradient methods for online learning and stochastic optimization.” In: in: *Journal of machine learning research* 12 (7) (2011).

[21] V. Mnih et al. “Human-level control through deep reinforcement learning”. In: in: *nature* 518 (7540) (2015), pp. 529–533.

[22] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: in: *nature* 529 (7587) (2016), pp. 484–489.

[23] R. S. Sutton et al. “Policy gradient methods for reinforcement learning with function approximation”. In: in: *Advances in neural information processing systems* 12 (1999).

[24] M. Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing Ltd, 2020.

[25] C. J. Watkins and P. Dayan. “Q-learning”. In: in: *Machine learning* 8 (3) (1992), pp. 279–292.

[26] V. Konda and J. Tsitsiklis. “Actor-critic algorithms”. In: in: *Advances in neural information processing systems* 12 (1999).

[27] A. G. Barto, R. S. Sutton, and C. W. Anderson. “Neuronlike adaptive elements that can solve difficult learning control problems”. In: in: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13 (5) (1983), pp. 834–846.

[28] L. Graesser and W. L. Keng. *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional, 2019.

[29] J. Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[30] J. Schulman et al. "Proximal policy optimization algorithms". In: in: *arXiv preprint arXiv:1707.06347* (2017).

[31] J. Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: in: *arXiv preprint arXiv:1506.02438* (2015).

[32] M. G. Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". In: in: *Journal of artificial intelligence research* 47 (2013), pp. 253–279.

[33] E. Todorov, T. Erez, and Y. Tassa. "Mujoco: A physics engine for model-based control". In: *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2012, pp. 5026–5033.

[34] T. P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: in: *arXiv preprint arXiv:1509.02971* (2015).

[35] S. Fujimoto, H. Hoof, and D. Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[36] R. Mahony, V. Kumar, and P. Corke. "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor". In: in: *IEEE robotics & automation magazine* 19 (3) (2012), pp. 20–32.

[37] S. Bouabdallah. "Design and control of quadrotors with application to autonomous flying". In.

[38] R. W. Beard. "Quadrotor dynamics and control". In: in: *Brigham Young University* 19 (3) (2008), pp. 46–56.

[39] M. Faessler, D. Falanga, and D. Scaramuzza. "Thrust mixing, saturation, and body-rate control for accurate aggressive quadrotor flight". In: in: *IEEE Robotics and Automation Letters* 2 (2) (2016), pp. 476–482.

[40] M. Faessler et al. "Autonomous, Vision-based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle". In: in: *Journal of Field Robotics* 33 (4) (2016), pp. 431–450.

[41] Y. Song and D. Scaramuzza. "Policy Search for Model Predictive Control With Application to Agile Drone Flight". In: in: *IEEE Transactions on Robotics* 38 (4) (2022), pp. 2114–2130.

[42] A. Loquercio et al. “Learning high-speed flight in the wild”. In: in: *Science Robotics* 6 (59) (2021), eabg5810.

[43] J. Hwangbo et al. “Learning agile and dynamic motor skills for legged robots”. In: in: *Science Robotics* 4 (26) (2019), eaau5872.

[44] E. Kaufmann et al. *Deep Drone Acrobatics*. 2020. URL: `https://arxiv.org/abs/2006.05768`.

[45] F. Sadeghi and S. Levine. “Cad2rl: Real single-image flight without a single real image”. In: in: *arXiv preprint arXiv:1611.04201* (2016).

[46] X. B. Peng et al. “Sim-to-real transfer of robotic control with dynamics randomization”. In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2018, pp. 3803–3810.

[47] J. Tobin et al. “Domain randomization for transferring deep neural networks from simulation to the real world”. In: *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2017, pp. 23–30.

[48] T. Kim, M. Jang, and J. Kim. “A Survey on Simulation Environments for Reinforcement Learning”. In: *2021 18th International Conference on Ubiquitous Robots (UR)*. IEEE, 2021, pp. 63–67.

[49] V. Makoviychuk et al. “Isaac gym: High performance gpu-based physics simulation for robot learning”. In: in: *arXiv preprint arXiv:2108.10470* (2021).

[50] M. Mittal et al. “Orbit: A unified simulation framework for interactive robot learning environments”. In: in: *IEEE Robotics and Automation Letters* 8 (6) (2023), pp. 3740–3747.

[51] Y. Song et al. “Flightmare: A flexible quadrotor simulator”. In: *Conference on Robot Learning*. PMLR. 2021, pp. 1147–1157.

[52] J. Eschmann, D. Albani, and G. Loianno. “RLtools: A fast, portable deep reinforcement learning library for continuous control”. In: in: *Journal of Machine Learning Research* 25 (301) (2024), pp. 1–19.

[53] A. J. Barry, P. R. Florence, and R. Tedrake. “High-speed autonomous obstacle avoidance with pushbroom stereo”. In: in: *Journal of Field Robotics* 35 (1) (2018), pp. 52–68.

[54] D. Mellinger and V. Kumar. “Minimum snap trajectory generation and control for quadrotors”. In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 2520–2525.

[55] B. Zhou et al. “Robust and efficient quadrotor trajectory generation for fast autonomous flight”. In: in: *IEEE Robotics and Automation Letters* 4 (4) (2019), pp. 3529–3536.

[56] E. Kaufmann et al. “Deep Drone Racing: Learning Agile Flight in Dynamic Environments”. In: *Proceedings of The 2nd Conference on Robot Learning*. Ed. by A. Billard et al. Vol. 87. Proceedings of Machine Learning Research. PMLR, 2018, pp. 133–145.

[57] D. Falanga, S. Kim, and D. Scaramuzza. “How fast is too fast? the role of perception latency in high-speed sense and avoid”. In: in: *IEEE Robotics and Automation Letters* 4 (2) (2019), pp. 1884–1891.

[58] R. Polvara et al. “Toward end-to-end control for UAV autonomous landing via deep reinforcement learning”. In: *2018 International conference on unmanned aircraft systems (ICUAS)*. IEEE. 2018, pp. 115–123.

[59] C. Sampedro et al. “Image-based visual servoing controller for multirotor aerial robots using deep reinforcement learning”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 979–986.

[60] S. Belkhale et al. “Model-based meta-reinforcement learning for flight with suspended payloads”. In: in: *IEEE Robotics and Automation Letters* 6 (2) (2021), pp. 1471–1478.

[61] B. Rubí, B. Morcego, and R. Pérez. “Deep reinforcement learning for quadrotor path following with adaptive velocity”. In: in: *Autonomous Robots* 45 (1) (2021), pp. 119–134.

[62] J. Lin et al. “Flying through a narrow gap using neural network: an end-to-end planning and control approach”. In: *2019 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2019, pp. 3526–3533.

[63] P. Becker-Ehmck et al. “Learning to fly via deep model-based reinforcement learning”. In: in: *arXiv preprint arXiv:2003.08876* (2020).

[64] E. Kaufmann, L. Bauersfeld, and D. Scaramuzza. “A benchmark comparison of learned control policies for agile quadrotor flight”. In: *2022 International Conference on Robotics and Automation (ICRA)*. IEEE. 2022, pp. 10504–10510.

[65] A. Molchanov et al. “Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 59–66.

[66] M. Wang et al. “Agile Flights Through a Moving Narrow Gap for Quadrotors Using Adaptive Curriculum Learning”. In: in: *IEEE Transactions on Intelligent Vehicles* 9 (11) (2024), pp. 6936–6949.

[67] J. Hwangbo et al. “Control of a quadrotor with reinforcement learning”. In: in: *IEEE Robotics and Automation Letters* 2 (4) (2017), pp. 2096–2103.

[68] C.-H. Pi et al. “Low-level autonomous control and tracking of quadrotor using reinforcement learning”. In: in: *Control Engineering Practice* 95 (2020), p. 104222.

[69] Y. Song et al. “Autonomous drone racing with deep reinforcement learning”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2021, pp. 1205–1212.

[70] S. Gronauer et al. “Using simulation optimization to improve zero-shot policy transfer of quadrotors”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 10170–10176.

[71] D. Zhang et al. “Learning a single near-hover position controller for vastly different quadcopters”. In: in: *arXiv preprint arXiv:2209.09232* (2022).

[72] A. Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: in: *Advances in neural information processing systems* 32 (2019).

[73] A. Serrano-Munoz et al. “skrl: Modular and flexible library for reinforcement learning”. In: in: *Journal of Machine Learning Research* 24 (254) (2023), pp. 1–9.

[74] W. Giernacki et al. “Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering”. In: *2017 22nd international conference on methods and models in automation and robotics (MMAR)*. IEEE. 2017, pp. 37–42.

[75] M. Towers et al. “Gymnasium: A Standard Interface for Reinforcement Learning Environments”. In: in: *arXiv preprint arXiv:2407.17032* (2024).

[76] J. Eschmann. “Reward function design in reinforcement learning”. In: in: *Reinforcement learning algorithms: Analysis and Applications* (2021), pp. 25–33.