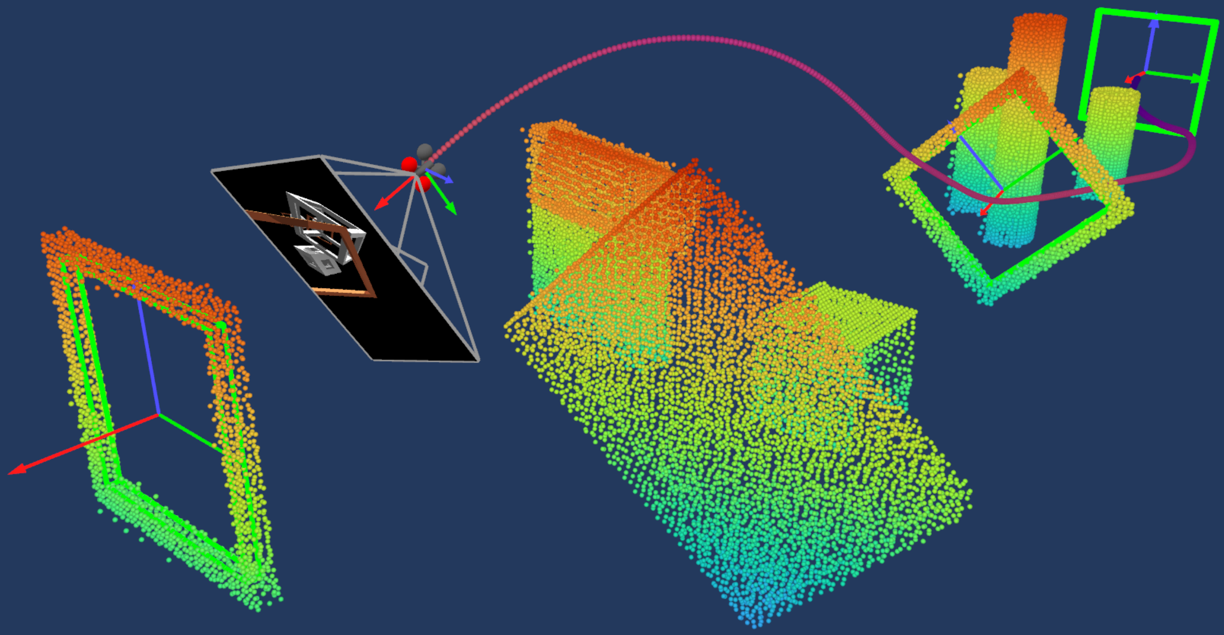
Learning Generalizable Policy for Obstacle-Aware Autonomous Drone Racing

Yueqian Liu



TU Delft

Learning Generalizable Policy for Obstacle-Aware Autonomous Drone Racing

Thesis report

by

Yueqian Liu

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on November 25, 2024 at 13:00.

Thesis committee:

Chair:	Dr. Ir. E. Smeur
Supervisors:	Dr. Ir. C. De Wagter
	Ir. H. Yu
External examiner:	Dr. S. Hulshoff
Place:	Faculty of Aerospace Engineering, Delft
Project Duration:	February 2024 - October 2024
Student number:	5758386

Faculty of Aerospace Engineering · Delft University of Technology

Delft
University of
Technology

Preface

Welcome, and thank you for taking the time to read my thesis. I wish I could provide a grander opening, but as I'm writing this after finishing everything else, I'd like to save a moment for other things. To save you a bit of time, feel free to skip this preface for now and perhaps return to it later—if you're still curious to read a few irrelevant words from someone who, at 24, has yet to achieve anything particularly noteworthy.

For those who chose to keep reading, here's a quick overview of this report to help set your expectations—maybe even lower them—so that the main content might surprise you in a positive way. This thesis focuses on my attempt to train a policy for drone racing that can also avoid obstacles in unseen environments, using reinforcement learning and domain randomization. You will read sentences written by a non-native English speaker, which are likely full of imperfections, mixed with some that GPT helped generate. You'll also find clumsy code implementations, naive methodology designs, sub-optimal performance metrics, and plenty of drone crashes. To put it bluntly, this might all seem like trash. But wait—calling it ”trash” might be a bit harsh, after all, there's value in sharing what didn't work well. If you're working on something similar, I hope my findings save you a few hours of trial and error.

For those returning after reading the entire thesis—welcome back! And for those who stuck with the preface, thank you as well. Reflecting on this project, I know that many aspects could have been done better. However, as a non-EU student paying €2,400 a month in tuition just to remain registered, extending this project indefinitely was not an option. The second half of the project was particularly stressful, especially when waiting for training to be completed without any certainty of success. And now, finally, it's done. I know, this thesis is far from perfect; it's flawed, buggy, and full of holes. But at least, the research objective has been achieved. One important lesson I've learned is that while it's good to think things through, getting the hands dirty and starting to try at the early stages is critical. There's no such thing as perfect preparation, so a rough early start is often good enough, and we can always improve everything on top of it. This way, the whole journey might be more enjoyable.

Lastly, I want to sincerely thank everyone who has supported me throughout this journey. I could attempt to list all your names, but there's always the risk of accidentally leaving someone out, which has happened to me before in my bachelor's thesis preface. So this time, I want to thank everyone who: (a) is reading this thesis, (b) has heard me talk about it, (c) helped me in any way—whether with code, papers, or life in general—and (d) feels connected to this project and believes they deserve credit. This way, I hope no one feels forgotten. Cheers!

Yueqian Liu
October 16, 2024

Contents

1 Introduction

1.1. Research Background

Drones have found diverse applications in various sectors, such as surveying, mapping, package delivery, and search-and-rescue operations. They are also utilized for tasks such as infrastructure inspection, environmental monitoring, agricultural management, and entertainment purposes like aerial photography. Many of these tasks require the drone to navigate through cluttered environments, avoid obstacles, and reach the goals in minimum time.

In our world, changes occur continuously, presenting challenges for autonomous navigation systems. Plans based on prior knowledge of the environment can become unsafe or outdated over time. To address this challenge, autonomous systems must have the intelligence to adapt to these changes effectively. This thesis explores possible solutions to make autonomous systems “smart” enough to adapt to environmental changes in the task of obstacle-aware autonomous drone racing.

Autonomous drone racing has emerged as a hot topic in research and a perfect testing ground for testing cutting-edge technologies that drive diverse drone applications, particularly those related to minimum-time autonomous navigation. In its basic form, obstacle-free autonomous drone racing involves navigating through gates to be traversed. However, in obstacle-aware drone racing, the navigation system faces the additional challenge of guiding the drone to safely avoid obstacles positioned between these waypoints. This task models various practical applications, such as search-and-rescue in the forest and urban food delivery.

In research, topics of general-purpose drone navigation in clutter and obstacle-free drone racing are relatively well studied, but the combination of the two, i.e. obstacle-aware autonomous drone racing, has received less attention. In this particular setting, there already exist strong optimization-based and learning-based baselines for static and previously known environments. However, for unknown environments, most methods rely on online planning and optimization, and learning-based methods have difficulty handling the uncertainty of environments. So we have decided to look into the problem of learning obstacle-aware drone racing policies that work in unknown environments.

The research background can be summarized in the following points. They collectively motivate the thesis and shape the upcoming research objectives and questions.

- Drones have found various applications, many of which require the drone to be capable of time-optimal navigation, avoid obstacles, and adapt to environmental changes.
- Obstacle-aware drone racing is a good model of various practical applications and is a testing ground for the underlying technologies. So, this task is relevant to the society.
- Despite its relevance, obstacle-aware drone racing remains unsolved with learning-based methods in unknown environments, presenting a research direction worth exploiting.

1.2. Research Objective and Questions

Based on the social background and the research landscape, the research object is formulated below. It also serves to narrow down the scope and to state a clear goal of this thesis project.

Research Objective

The research objective is to design and train a neural policy that navigates a quadcopter through static racing tracks while avoiding static obstacles by generating low-level throttle and body rate commands. The policy will have access to accurate quadcopter states, waypoint poses and sizes, and depth images of a front-facing camera. Any prior knowledge of obstacles will not be accessible, which means that the policy must make decisions on the fly to avoid crashing into obstacles.

Toward the research objective, we formulate three research questions that will be addressed sequentially as the research progresses.

Research Question 1

What are existing methods in the literature that can achieve the objective in whole or partially?

Research Question 2

If there is no existing method that achieves the objective completely, what is our proposed method?

Research Question 3

How does the policy trained using the proposed method perform in terms of navigation success rate, speed, and generalization ability?

To answer question 1, an extensive literature review will be conducted and closely related work will be studied in depth. Hopefully, the answer could give directions to answer question 2. Once question 2 is solved, the proposal will be implemented, then experiments will be designed and conducted to validate the proposed method, answering question 3.

1.3. Report Structure

This report is divided into four main parts, each contributing to the overall understanding of the thesis work and addressing different aspects of the project:

- **Part I: Preliminary Analysis.** This part lays the groundwork for the research conducted in the project. It begins with a literature review in Chapter 2, Chapter 3 then introduces the proposed method, explaining the rationale behind the approach and outlining the preliminary efforts made towards its implementation.
- **Part II: Early Results.** While the scientific article in Part III has the core contributions, this section showcases early findings that enrich the thesis but do not fit well into the article. These results are valuable in illustrating the broader implications of the work and include performance evaluations of the implemented software and experiments related to obstacle-free drone racing. They also serve the purpose of validating the work done in Part I.
- **Part III: Scientific Article.** This part provides a concise summary of the central work and contributions of the entire thesis. It highlights the core research questions, the methodology employed, and the key findings and results. The article serves as the primary representation of the contributions made in the project and is designed to stand alone as a compact synthesis of the thesis.
- **Part IV: Closure.** This concluding chapter summarizes the major findings, revisits the research questions, and assesses the extent to which the research objectives have been achieved. Additionally, it provides recommendations for future work, identifying areas where further research could extend or build upon the contributions made in this project.

Part I

Preliminary Analysis

2 Literature Review

2.1. Autonomous Drone Racing

Drone racing is an exhilarating sport that combines the thrill of high-speed flight with the excitement of competitive racing. In drone racing, pilots use specially designed drones, often highly agile quadcopters, equipped with front-facing cameras to navigate through complex courses filled with twists, turns, and perhaps obstacles. The pilots wear First Person View (FPV) goggles that provide a live video feed from the camera's perspective, allowing them to fly as if they were on the drones.

Drone racing competitions take place in various settings, from indoor arenas to outdoor tracks, each offering unique challenges. Several drone racing competitions have gained fame and recognition within the drone racing community. Organizations like the Drone Racing League (DRL) and MultiGP have hosted many human-piloted local and global competitions in the past few years.

Autonomous drone racing removes the human pilot from the process. In autonomous drone racing, the drones are equipped with artificial intelligence systems that allow them to navigate through a course without human control. Competitions for autonomous drone racing include the 2016-2019 IROS Autonomous Drone Racing (ADR) competitions [3, 4, 5], the 2019 AlphaPilot Challenge [6, 7], the 2019 NeurIPS Game of Drones [8], and the more recent 2022-2023 DJI RMUA UAV Challenges [9, 10]. These competitions have significantly encouraged research and development in the field.

Autonomous drone racing is essentially time-optimal autonomous drone navigation. The software architecture for general navigation usually consists of separate modules for perception, planning, and control. The software stack for racing can also be broken into these modules, with the planning module focused more on finding time-optimal trajectories and the perception and control modules adapted to high-speed agile flight in racing tracks. With advances in neural networks and learning methods, some approaches replace modules with neural networks and combine multiple modules into a single neural network. A survey on autonomous drone racing [11] reviews both learning-based and classical methods in the drone racing navigation software stack.

Since this thesis project concerns obstacle-aware drone racing, the literature review will categorize papers based on the racing environment: whether or not there are additional obstacles between the gates.

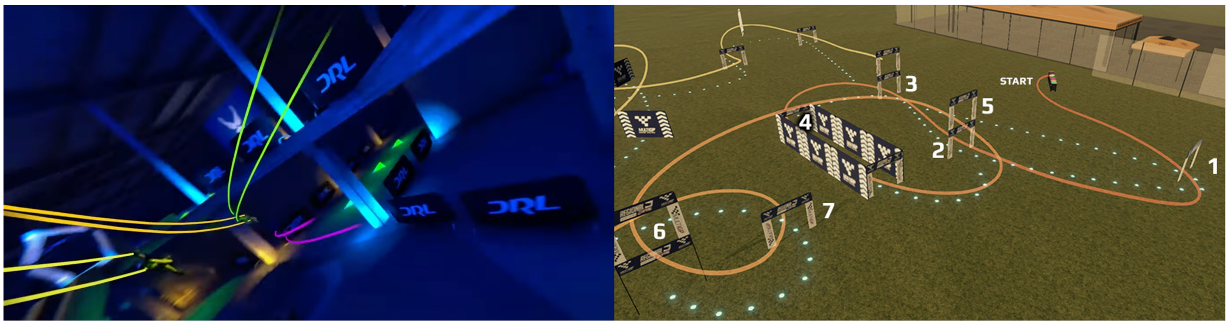


Figure 2.1: FPV of a drone in DRL Simulator [1] and 2023 MultiGP Championship track [2].

The tracks in early competitions, such as the IROS ADR, AlphaPilot, and NeurIPS Game of Drones, are situated in less cluttered space [12, 7, 13], which allow drones to complete the tracks without considering obstacle avoidance, as long as the drones do not deviate too far from the line segments connecting the gates. However, for harder tracks in cluttered environments, such as the tracks of the more recent DJI RMUA Challenges [9, 14], the absence of obstacle awareness could cause safety issues. Additionally, in human-piloted drone racing, DRL competitions for example, and in drone racing video games, there are plenty of tracks that require obstacle-avoidance ability.

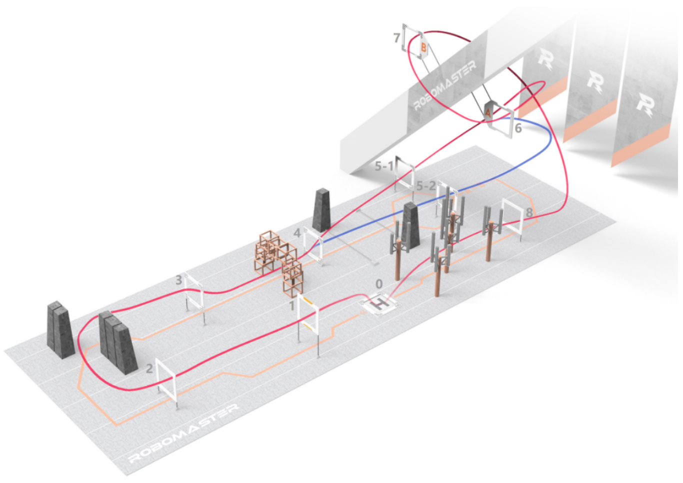


Figure 2.2: Track layout of 2023 DJI RMUA UAV Challenge [14].

2.1.1. Obstacle-Free Autonomous Drone Racing

We will start by looking at solutions for autonomous drone racing competitions. Team KIRD from KAIST won the 2016 IROS ADR competition using a visual servoing method [15]. KIRD used precomputed altitude, heading, and forward velocity commands, which were obtained from the given racing track, to roughly navigate the drone from gate to gate. Then RGBD-based visual servoing was employed to guide the drone through gates. To execute velocity commands for gate-to-gate navigation and visual servoing, optical flow was calculated using a downward-facing camera for velocity feedback.

In the 2017 IROS ADR, team INAOE took the lead [3]. INAOE used monocular ORB-SLAM [17] and height measurement from a barometer and an ultrasound sensor to obtain drone position up to scale. Gates were detected using color filtering and the gate positions were used to compensate for visual SLAM drift. With position feedback, INAOE employed a classical cascaded PID position controller to follow relative waypoints to complete the track.

Team RPG won the 2018 IROS ADR [4]. On the perception front, they used Visual Inertial Odometry (VIO) for state estimation, and a neural network for detecting relative gate poses. With the drone's states estimated by VIO, and relative gate poses, gate poses in the odometry frame were obtained using Extended Kalman Filter (EKF). Regarding planning, waypoints were first generated using gate poses in the odometry frame, then reference trajectories were interpolated based on the waypoints. Finally, Perception-Aware

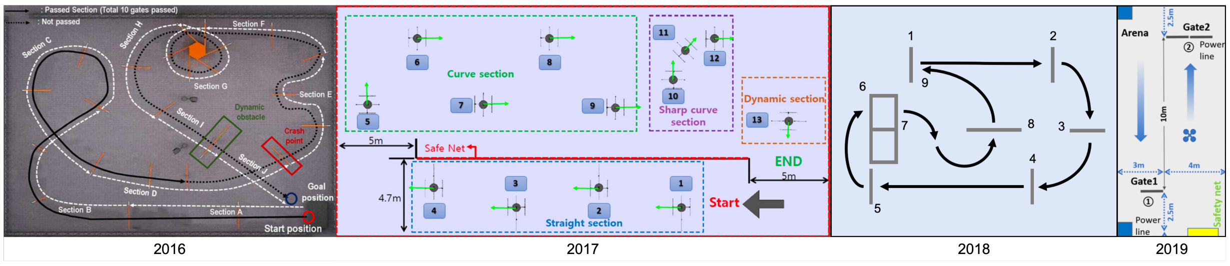


Figure 2.3: Tracks of 2016-2019 IROS ADR [15, 3, 16].

Model Predictive Controller (PAMPC) [18] was employed for trajectory tracking.

The 2019 IROS ADR featured a simpler track but the drone was only allowed to pass through dynamically illuminated gates. Team UMD got the second place [16]. UMD used HSV color space gate detection and a lightweight visual-inertial localization method for perception. Waypoints were then generated with detected gate poses. A nonlinear model predictive controller was used for position control, which generates attitude commands for a PD attitude controller.

The first place of the 2019 AlphaPilot was team MAVLab [7]. Team RPG [6] was placed the second. MAVLab used a semantic segmentation neural network for gate detection. State estimation was built upon the Visual Model-predictive Localization (VML) method [19]. Planning was on the waypoint-heading level and control was done by a classical cascaded PID controller with gate-aware lateral position control. RPG also used a learning-based method for gate detection, but the other modules in the pipeline were different from MAVLab's. For state estimation, they opted for fusing results from an off-the-shelf VIO, ROVIO [20], and relative gate pose estimations, for joint estimation of VIO drift and global gate poses. Paths were planned in a receding horizon with sampling-based methods. The paths were approximated by polynomials before being tracked using a classical PID control scheme. To ensure that the VIO works correctly, the maximum velocity in the planning module was set to a conservative value, which resulted in a longer lap time than team MAVLab.

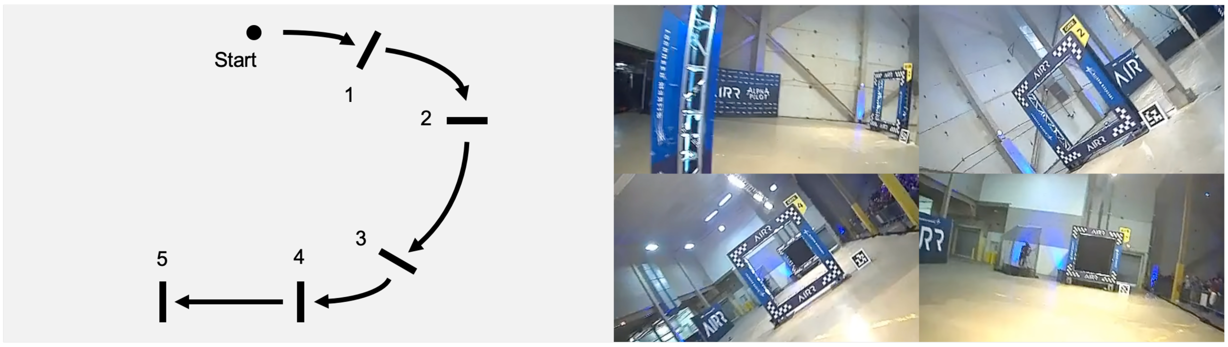


Figure 2.4: Track layout of 2019 AlphaPilot final race and FPV images [7].

The 2019 NeurIPS Game of Drones included three Tiers, for all Tiers the ground truth of drone pose was accessible. In Tier 1, the gate poses were accurately known, while in Tier 2 and 3 only noisy gate poses were given. A baseline opponent competitor was present in Tier 1 and 3 but not in 2. There aren't detailed descriptions about team Dedale's approach [21], which was placed the first for Tier 1. They probably relied on drag-aware and flight-corridor-based joint planning and control via MPC similar to [22]. In Tier 2 and 3, since accurate gate poses were not provided, the winner team [23] employed spline trajectory planning and tracking to navigate the drone to the approximate poses where the gates could be detected visually, then employed an action generation network trained using reinforcement learning [24] to pass through the gates. Parameters, such as maximum velocity and acceleration, for spline trajectory generation and tracking, were optimized using the Genetic Algorithm for the minimum lap time.

Figure 2.5: Racing environments of 2019 NeurIPS Game of Drones [8].

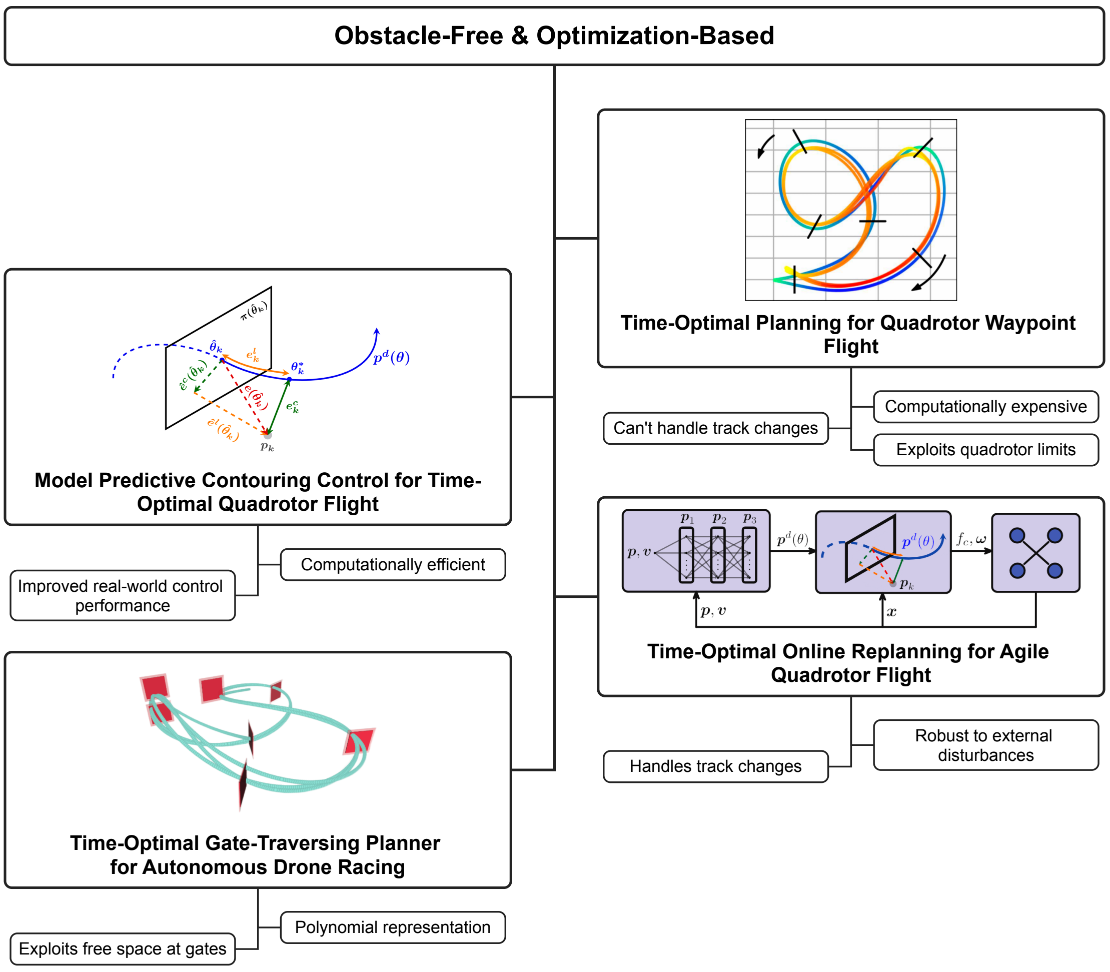

Obstacle-Free & Optimization-Based
Time-Optimal Planning for Quadrotor Waypoint Flight
Can't handle track changes
Computationally expensive
Exploits quadrotor limits
Model Predictive Contouring Control for Time-Optimal Quadrotor Flight
Improved real-world control performance
Computationally efficient
Time-Optimal Online Replanning for Agile Quadrotor Flight
Handles track changes
Robust to external disturbances
Time-Optimal Gate-Traversing Planner for Autonomous Drone Racing
Exploits free space at gates
Polynomial representation

Figure 2.6: An overview of serveral optimization-based methods [25, 26, 27, 28].

Around the time team RPG participated in the 2018 IROS ADR, they published the Deep Drone Racing series [29, 30], presenting methods similar to that made for the competition. The main idea was also using a neural network for gate detection, whose outputs were then fed into a trajectory generator, finally the trajectory was tracked by a controller. However, the output modality of the neural network, training framework, and control method were different. The neural network's outputs were directions in the image frame and normalized desired speed. The training was done using imitation learning, and for control, classical approaches were employed. The second paper in the series augmented network training with domain randomization, which allowed the drone to adapt to environments not seen at training time.

Besides the competitions, the AGILEFLIGHT project [31] has also facilitated research and development in autonomous drone racing. Optimization is a powerful and important tool in this field. For a static and obstacle-free racing track, each gate could be assigned a waypoint at the middle, then the time-optimal trajectory passing the waypoints can be planned using optimization with Complementary Progress Constraint (CPC) [25]. It beats team Dedale's record in the 2019 NeurIPS Game of Drones. However, it is computationally expensive and has difficulty adapting to changing track layouts. To address the first issue, a Model Predictive Contouring Control (MPCC) approach has been proposed [26]. MPCC can produce a near-time-optimal flight trajectory when tested in simulation and outperforms the CPC-MPC approach in the real world. In addition, it requires orders of magnitude less computational time when combined with point-mass-model (PMM) reference path generation. MPCC can also work with an online reference path generation module for fast re-planning, which enables adapting to changing tracks and coping with external disturbances [27]. With re-planning enabled, lap times in real-world testing are further reduced.

Although the CPC method plans time-optimal "waypoint" flights, yet due to the freedom of selecting the crossing point of each gate, CPC's result might not be the time-optimal solution for the given racing track.

A recent work [28] demonstrates that lap times can be further reduced even with polynomial trajectories by leveraging the spatial potential of the gates.

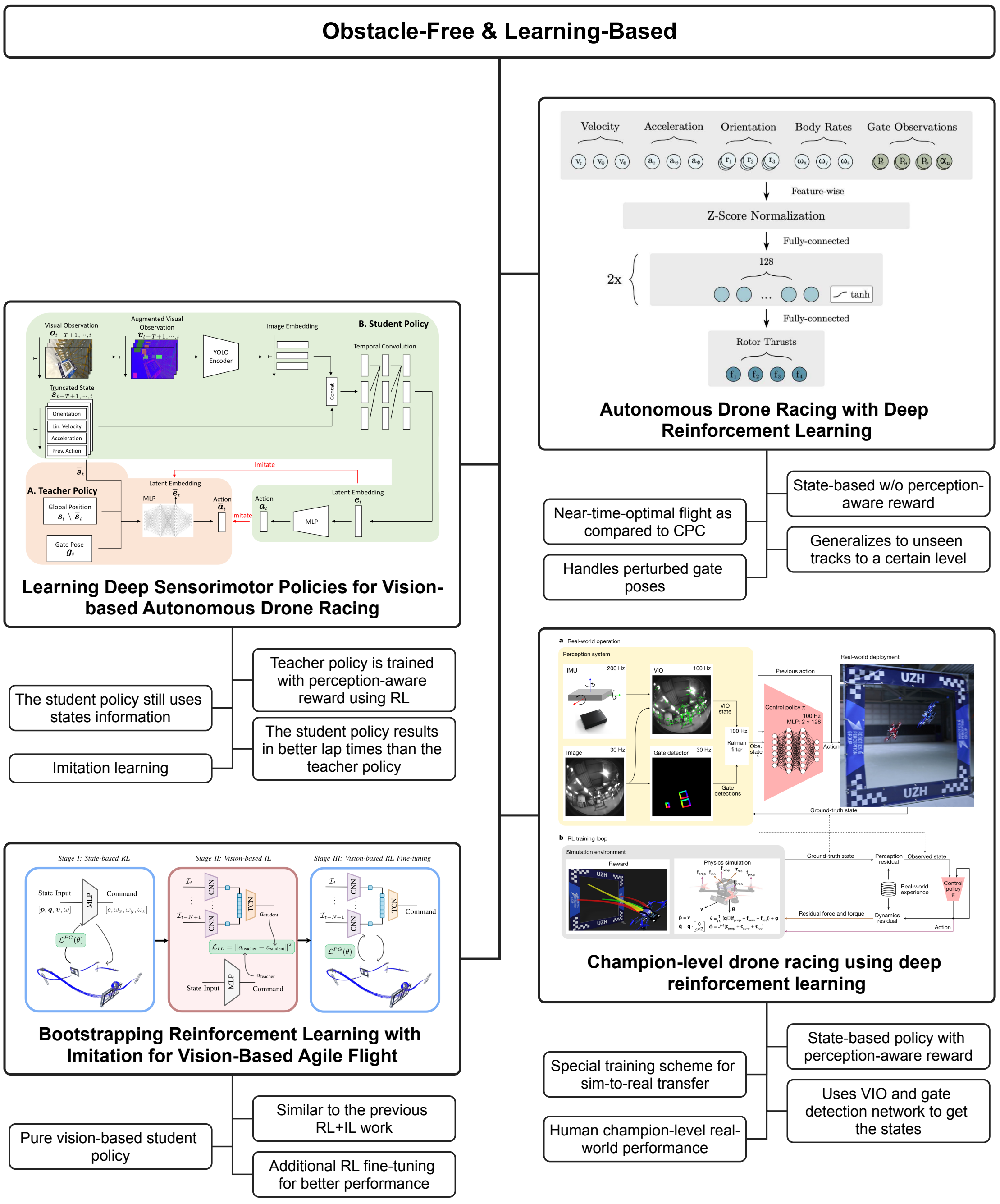

Obstacle-Free & Learning-Based
Autonomous Drone Racing with Deep Reinforcement Learning
Velocity
Acceleration
Orientation
Body Rates
Gate Observations
Feature-wise
Z-Score Normalization
Fully-connected
128
2x
tanh
Rotor Thrusts
Near-time-optimal flight as compared to CPC
Handles perturbed gate poses
State-based w/o perception-aware reward
Generalizes to unseen tracks to a certain level
Learning Deep Sensorimotor Policies for Vision-based Autonomous Drone Racing
B. Student Policy
A. Teacher Policy
YOLO Encoder
Imitate
The student policy still uses states information
Imitation learning
Teacher policy is trained with perception-aware reward using RL
The student policy results in better lap times than the teacher policy
Champion-level drone racing using deep reinforcement learning
Real-world operation
Real-world deployment
RL training loop
Special training scheme for sim-to-real transfer
Human champion-level real-world performance
State-based policy with perception-aware reward
Uses VIO and gate detection network to get the states
Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight
Stage I: State-based RL
Stage II: Vision-based IL
Stage III: Vision-based RL Fine-tuning
Pure vision-based student policy
Similar to the previous RL+IL work
Additional RL fine-tuning for better performance

Figure 2.7: An overview of several learning-based methods [32, 33, 34, 35].

Reinforcement Learning (RL) is another approach to autonomous drone racing. It has been demonstrated that near-time-optimal agile flight can be achieved by leveraging state-based RL. In [32], A deep neural network trained with RL maps the current drone states and relative gate states directly to rotor thrusts, which are applied to the simulated drone model to produce roll-out trajectories. Compared to the CPC method, this RL method under-performs slightly in terms of lap time, but can better handle dynamic gates and generalize to unseen tracks. Moreover, deep policies optimized using RL can be a good teacher policy in the imitation learning framework: Fu et al. [33] trained a deep policy with additional perception reward for keeping the next gate in camera FOV (Field of View) as the teacher policy, then employ imitation learning to train a deep policy that maps both states and FPV images to commands. It achieves better lap times compared to the state-only policy in [32], while preserving robustness to disturbances and noises.

Deploying policies learned via reinforcement learning in the real world is challenging. It requires special designs on sim-to-real transfer and careful system engineering. With these requirements satisfied, learning-based approaches have the potential to reach human-level performance. For autonomous drone racing, this is demonstrated by Kaufmann et al. [34]. Their racing drone, powered by the Swift system, has a state-based neural policy at the core of the control module. The neural policy is initially trained similarly to the teacher policy in [33] with perception reward. To enable sim-to-real transfer, the policy weights are fine-tuned in simulation with residual dynamics and noises identified using real-world data. Since the policy is state-based, the perception module performs gate detection using methods seen in their AlphaPilot solution [6], and estimates drone states fusing both the VIO results, gate detection results, and ground truth gate poses. Throughout multiple races with human world champions, Swift can achieve average shorter lap times and better performance consistency.

Xing et al. [35] revisit the idea in [33] of using imitation learning to train vision-based student policies. Obtaining the student policy from imitation learning is not the end in [35], it is further fine-tuned through RL for better performance. Interestingly, the monocular vision-based policy does not have access to any of the quadcopter states, suggesting that high-speed agile flight is possible with identical input-output modalities as human pilots.

The learning-based methods using RL have several advantages over optimization-based methods in autonomous drone racing, thanks to RL's task-level objective optimization, expressiveness of deep neural networks, and domain randomization techniques [36]. The advantages include not only better lap times but also higher success rates in real-world flights, where unmodeled effects and disturbances are non-negligible.

2.1.2. Obstacle-Aware Autonomous Drone Racing

Obstacle avoidance in general-purpose drone navigation, which does not prioritize aggressive flight or the shortest navigation time, is a widely researched topic [37, 38, 39, 40, 41, 42]. While these methods are not optimized for high-speed, minimum-time flight, they can still be used to navigate drones through races. In contrast, obstacle avoidance in autonomous drone racing is a relatively less explored area, gaining attention only in recent years.

The teach-and-repeat navigation framework is widely used in autonomous robot missions, and has been applied to drone racing in [43]. In the teaching phase, a human pilot would navigate the drone in the cluttered environment and pass through the gates, and a flight corridor is generated, capturing the topological structure of the demonstrated path. A global trajectory is then optimized within the flight corridor using the Coordinate Descent Algorithms [44]. The repeat phase involves re-planning, which deforms the global trajectory in a receding horizon, to avoid additional obstacles and to handle VIO drift. The deformed trajectories are then tracked using a geometric controller [45]. This teach-repeat-replan design enables the drone to fly through the track while avoiding previously unseen and dynamic obstacles.

In static and deterministic environments, Fast-Racing provides a polynomial baseline for obstacle-aware autonomous drone racing [46]. At its core is a GPU-accelerated global planning algorithm based on the GCOPTER framework [47]. It plans on SE(3) within the flight corridor, so a planned trajectory might pass through narrow gaps instead of spending more time going around the obstacles. This method is relatively efficient: it takes seconds on GPU and less than one minute on CPU to plan a trajectory for a track that has a comparable size to 2019 NeurIPS Game of Drone tracks. The winner solution [9] of the 2022 DJI RMUA UAV Challenge is also polynomial-based. Compared to Fast-Racing, this method has two major upgrades, one is using time-uniform MINCO trajectory representation instead of the vanilla MINCO, and another is adding an online re-planning module for avoiding dynamic obstacles and passing through moving gates.

Obstacle-free time-optimal waypoint flight is tackled in CPC [25], then how to plan a time-optimal flight in environments with obstacles? Penick et al. [48] provide a sampling-based baseline. This sampling-based method is capable of finding high-quality solutions in complex cluttered environments, but doesn't scale very well: as the environment complexity increases, the computation time increases, and solution quality degrades. In complex environments, a follow-up work, a reinforcement-learning-based method [49] outperforms it. Regarding success rate, two real-world deployments are tested: RL-based with

Betaflight [50] rates controller, and sampling-based with MPC: the RL deployment has higher success rates, demonstrating its ability to adapt to unmodeled effects.

In obstacle-free environments, state-based RL-trained policy can be the teacher policy for training a vision-based policy in [33]. Similarly, in the obstacle-aware case, a vision-based student policy [51] is obtained from [49] using almost the same methodology. The teacher policy in [51] is retrained with perception reward, which results in a slight increase in lap times, but the vision-based student policy has the shortest lap times in general, due to the observed cutting-corner behavior. Despite its good performance, the vision-based policy still includes numerical state input for decision-making.

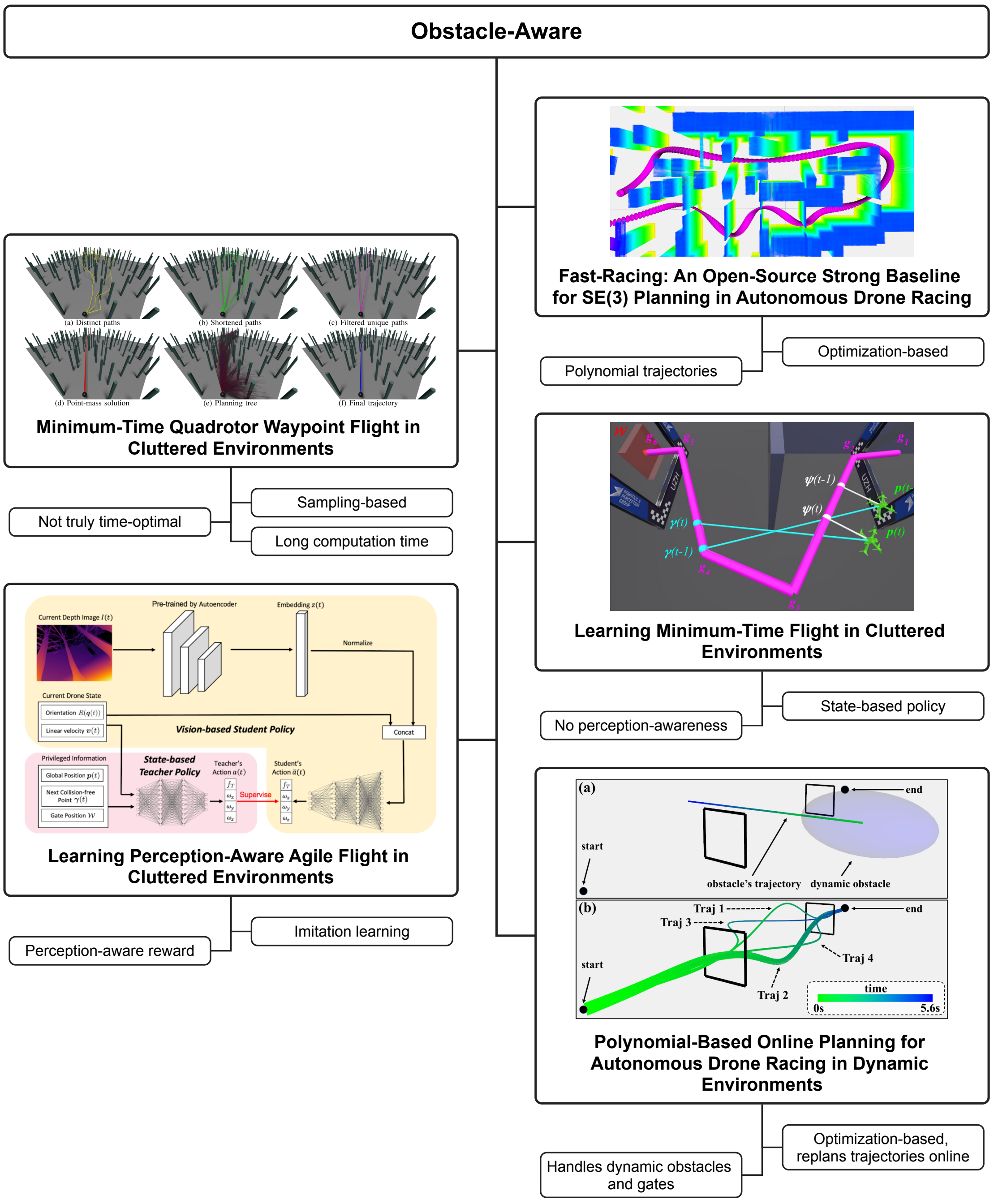


Figure 2.8: An overview of several methods for obstacle-aware drone racing [46, 49, 51, 52].

2.2. Vision-Involved Navigation via Deep RL

Learning-based methods for obstacle-aware drone racing [49, 51] often fail to generalize to environments different from those in which they were originally trained. This limitation suggests that the policy networks tend to “memorize” specific actions in response to sensor data, rather than genuinely learning the principles of obstacle avoidance. To address this issue, research on general-purpose navigation using Deep RL offers valuable insights.

Near-perfect discrete-action indoor navigation for ground robots has been demonstrated with DD-PPO [53]. The agent utilizes a policy network comprising a Convolutional Neural Network (CNN) as the encoder and a Long Short-Term Memory (LSTM) network to integrate information from different timestamps and make navigation decisions. This network maps visual observations and relative goal poses to discrete actions, as illustrated in Figure 2.9. Training occurs across multiple reconstructed indoor scenes to enhance generalization. During the RL training process, the weights of both the CNN and LSTM are jointly optimized, enabling the CNN to extract features particularly useful for navigation, outperforming a CNN pre-trained on ImageNet [54].

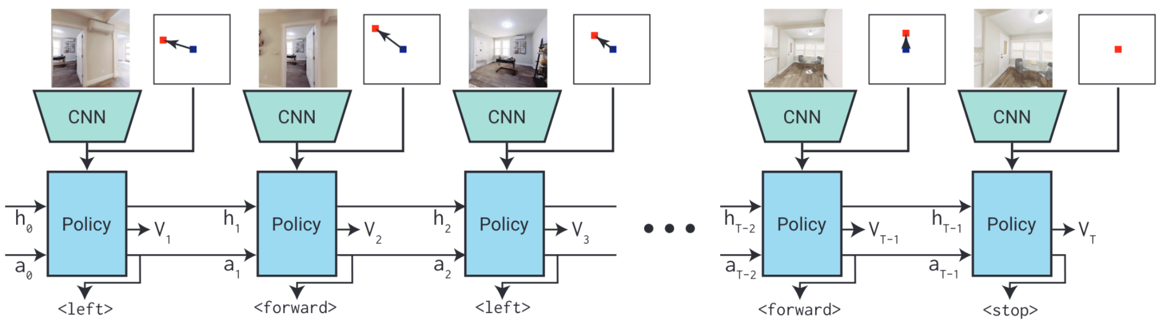


Figure 2.9: Network architecture of the point-goal navigation agent in [53].

However, jointly optimizing all modules in the policy network using only RL, as described in DD-PPO, is inefficient and costly. Incorporating auxiliary tasks is one method to enhance training efficiency. Desai et al. [55] and Ye et al. [56] demonstrate that using auxiliary tasks in training point-goal navigation agents reduces the number of steps needed to reach peak performance to about one-fifth to one-quarter of the steps required without auxiliary tasks. Additionally, when trained for the same number of steps, including auxiliary tasks improves the final performance.

Another method to enhance training efficiency is modular learning, where network modules are learned separately. Once an upstream module is learned, it is frozen while downstream modules are optimized. Although this approach may seem less “clean” compared to using auxiliary tasks, it offers the potential for easier integration of multi-domain knowledge, such as knowledge from both simulations and the real world.

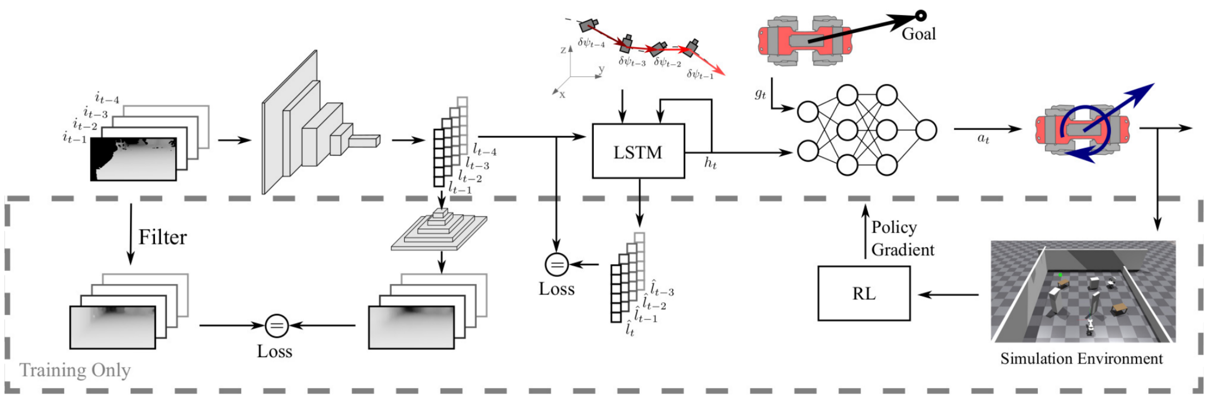


Figure 2.10: Network architecture and training methods of the quadruped agent in [57].

Hoeller et al. [57] propose a modular learning framework for training a quadruped robot to navigate in cluttered dynamic environments. The network consists of a CNN encoder, an LSTM recurrent neural network (RNN), and a multi-layer perceptron (MLP) action network. The training process involves four steps: (1) training a variational autoencoder (VAE) to obtain the encoder weights, which are then frozen, (2) training a baseline policy (action network) to map encoded images to actions, (3) collecting data, including depth images and camera poses, with the baseline policy and using the data to train the LSTM to predict future latent vectors, and (4) training the MLP with the encoder and LSTM weights fixed. An overview of this framework is provided in Figure 2.10.

Inspired by [57], MAVRL [41] adopts a similar framework for drone navigation but introduces additional supervision during the LSTM network training phase. As depicted in Figure 2.11, the LSTM is explicitly trained to memorize or predict past, current, and future latent spaces without the aid of vehicle state information. MAVRL's experimental results suggest that retaining memory of the current and past latent spaces is more beneficial for efficient training than predicting the future latent space. Kulkarni et al. [42] propose a similar method, using a procedural and modular training framework where the encoder is first trained using Deep Collision Encoding (DCE) [58], followed by training the policy network (comprising the MLP and RNN) with the DCE weights fixed.

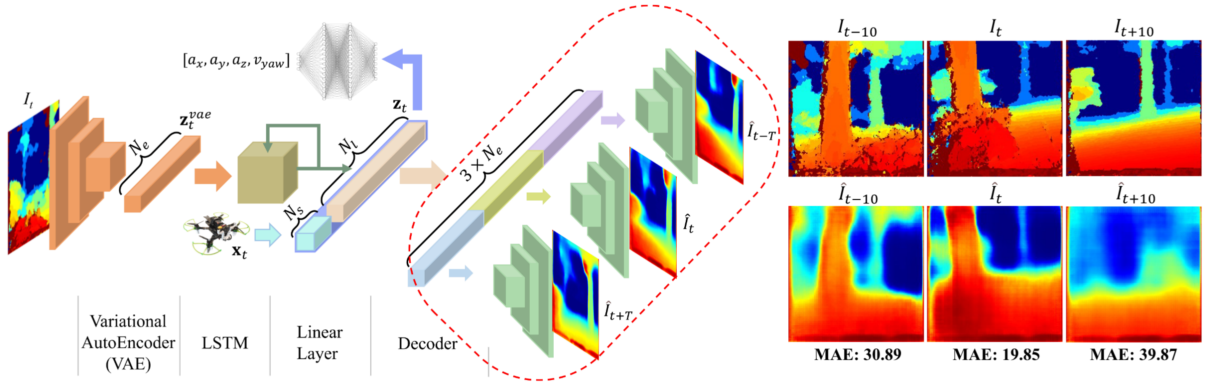


Figure 2.11: Network architecture and LSTM training (red dotted line) of the drone agent in [41].

Zhao et al. [59] propose incorporating a learned agility policy into a traditional navigation pipeline to create a hybrid system, as shown in Figure 2.12. Unlike MAVRL and Kulkarni's approaches, the action in this hybrid system is a parameter that indicates the level of agility for an optimization-based planner. More importantly, instead of encoding images, they encode the local occupancy map, which represents geometric data from multiple fused depth images. The voxel encoder is trained from scratch in a two-stage setup: the first stage focuses on learning a good representation of the voxel map, and the second stage fixes the encoder weights while training the MLP to maximize other reward terms. This approach provides an alternative method for extracting useful information from depth images without relying on a 2D encoder and an RNN.

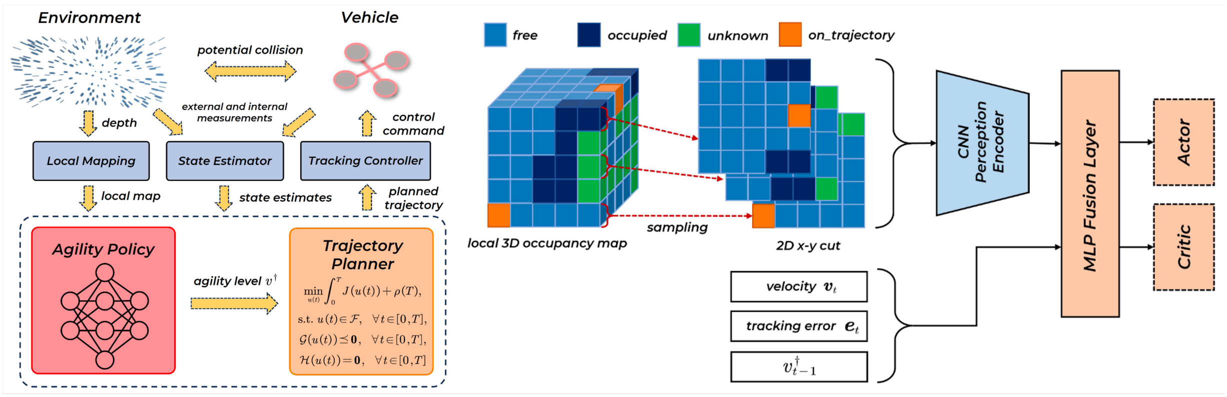


Figure 2.12: Overview of the hybrid system and agility policy network architecture in [59].

To this point, we have reviewed several works on enabling general-purpose vision-based navigation through Deep RL. We've seen multiple strategies for designing policy networks, training agents, and enhancing training efficiency. Usually, a policy network would include CNNs for extracting visual features, RNNs for fusing multi-step information, and MLPs for bridging network modules and deciding on actions. To improve training efficiency, employing auxiliary tasks and modular learning have proven to be effective approaches. For generalization ability, these works all involve training in multiple different environments to avoid "environmental overfitting". These studies serve as important references and inspirations for developing agents capable of doing flexible and "real" obstacle avoidance in drone racing.

2.3. Modeling and Simulation

Accurately modeling and simulating the drone and the environment is crucial in the research and development of autonomous drone racing. Good models are important to both optimization-based and RL-based methods. For optimization-based methods, which mostly rely on a reference model, the fidelity of the reference model greatly influences real-world performance. In the RL setting, the simulated environment with which the agent interacts is a model of the real world. How well the model abstracts the real world, and to which parts domain randomization can be applied both affect the overall performance and generalization ability of the learned policy. We will first provide an overview of related simulators and then elaborate on drone physics modeling.

2.3.1. Simulators

Gazebo [60] is one of the most popular simulators for robotics. It can simulate multi-rigid-body dynamics and accepts plugins for customized simulation. RotorS [61] builds upon Gazebo for multirotor-oriented simulation by providing a set of sensor and actuator plugins. A similar set of plugins is also integrated into PX4 [62] for SITL (Software In The Loop) flight with Gazebo. Although widely used, Gazebo offers limited capabilities in photo-realistic rendering and efficient parallel simulation, making it a less favorable choice for developing vision-based or learning-based algorithms.

AirSim [63] is a simulator built on top of Unreal Engine 4. Thanks to Unreal Engine, AirSim can do photo-realistic rendering and can take advantage of the Unreal Marketplace which provides a rich set of assets and scenes. Besides visual rendering, it simulates additionally aerodynamic forces and torques, which are not seen in RotorS. AirSim is widely used for the validation of vision-based navigation and training vision-based networks. Despite its rendering strength, there are limitations. Firstly, it is inconvenient to customize scene and track layouts, as the editor is tightly coupled with Epic Game Launcher and is only available on the Windows operating system. Secondly, the physics simulation speed is limited to about 1000 steps per second, which is quite slow compared to newer simulators.

FlightGoggles [64] is another simulator capable of photo-realistic rendering. It is based on Unity, which has better cross-platform support than Unreal. Compared to AirSim, FlightGoggles uses simpler aerodynamic effects but adds motor dynamics and the torque induced by motor acceleration. The physics updating frequency is on par with AirSim, achieving about 960 steps per second. Using decoupled rendering and physics engines represents a more flexible architecture.

Flightmare [65] is a Unity-based simulator for quadrotors. Like FlightGoggles, it adopts the architecture that decouples rendering and physics simulation. Flightmare offers three implementations for physics simulation: RotorS, real-world dynamics, and parallelized classical dynamics. Using the third option and running on an i7-8850H CPU, physics can be simulated at 25,000 steps per second for one quadcopter, and 175,000 steps per second for 50 to 150 quadcopters. However, the RGB image rendering speed is limited to about 30 frames per second (FPS) for 512$\times$512 RGB images. The rendering speed may be boosted on a GPU, but it goes up to about only 200 FPS (640$\times$480 RGB) on an RTX 4090. These characteristics suggest that Flightmare is better suited for state-based RL, rather than vision-based RL.

Isaac Sim [66] is a general robotics simulator built on top of NVIDIA's Omniverse platform, whose underlying renderer is Omniverse RTX and the physics engine is PhysX. Isaac sim is shipped with simulated cameras and Lidars, articulated robotic arms and ground vehicles, plus communication APIs, but without direct support for drone simulation. Fortunately, Isaac Sim allows adding features through extensions. Pegasus Simulator [67] comes as an extension that implements missing sensors and actuators for drones. Pegasus to Isaac Sim is like RotorS to Gazebo.

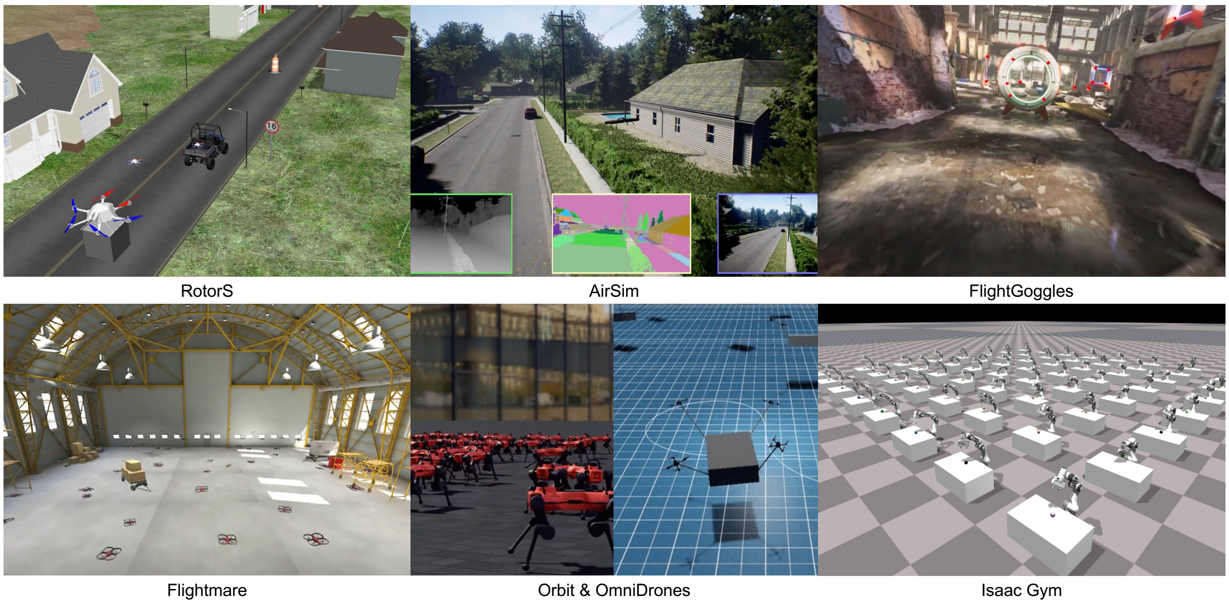


Figure 2.13: Visuals of the simulator and frameworks.

Orbit (Isaac Lab) [68], an open-source Isaac-Sim-powered robot learning framework hosted by the official Omniverse developer team, is designed to “be the environment zoo for Isaac Sim with contributions from the community as well as internal development” [69]. Orbit includes models for quadrupeds, robotic arms, grippers, hands, and mobile manipulators, also wrappers to learning libraries, and benchmark tasks. Simulation performance on an RTX 3090 GPU has also been reported in [68], physics simulation runs at up to in total of 125,000 steps per second for multiple robots, and rendering runs at 270 FPS for 640$\times$480 RGB images. The biggest advantage of using Isaac-Sim is that the physics simulation can be implemented as tensor operations, which scales very well for multiple instances in parallel.

OmniDrones [70], inspired by Orbit and being aware of the lack of drone support in Orbit, is a recently built drone learning framework parallel to Orbit. The authors acknowledge that the “abstractions and implementation of OmniDrones were inspired by Isaac Orbit. Some of the drone models (assets) and controllers are adopted from or heavily based on the RotorS simulator”. So OmniDrones can be regarded as “Orbit for drones” and its authors have planned to merge it with Orbit, but this has not yet been finished as of the time of writing [71]. Notably, OmniDrones implements the down-wash effect for multi-drone coordination tasks. Although vision-based tasks are not included, OmniDrones is still a good source of reference for drone-related learning tasks.

Isaac Sim and its associated frameworks, despite providing scalable physics simulations and photo-realistic rendering, have slow rendering speeds. what’s worse, rendering camera sensors would consume a large amount of video RAM (VRAM). These limitations make Isaac Sim and related products less suitable for vision-based reinforcement learning (RL). A viable alternative is Habitat Sim [72], which can achieve thousands of FPS for RGB-D rendering. However, Habitat Sim lacks direct support for drones and there is little information regarding customizing the environments. Another option is Isaac Gym [73], a simulator designed for robot learning that supports parallel physics simulations. Isaac Gym is the proof-of-concept product for robot learning technologies to be integrated into Isaac Sim. Although Isaac Gym is no longer actively developed, it achieves around two thousand FPS for image rendering and is VRAM efficient. Therefore, Isaac Gym is a better option than Isaac Sim for vision-based RL, especially for tasks that only require depth images.

Figure 2.13 illustrates the visuals produced by the mentioned simulators. The degree of photo-realism depends on both the quality of the assets used for rendering and the ability to simulate lighting effects within the scene. With high-quality assets, all simulators and frameworks can achieve a high level of photo-realism, except for RotorS and Isaac Gym, which have limited capabilities in simulating lighting effects.

Table 2.1: Drone modeling in previously reviewed literature.

Literature	Major Actuation	Rotor Drag	Dynamic Lift	Airflow Interaction	Body Drag	Battery Model	Rotor Dynamics	Rotor Acceleration Torque
RotorS [61]	Quadratic	Linear	-	-	-	-	1st-Order Lag	-
AirSim [63, 46, 9]	Quadratic	-	-	-	Quadratic	-	-	-
FlightGoggles [64]	Quadratic	-	-	-	Quadratic	-	1st-Order Lag	Linear
Flightmare [65, 32]	Direct	-	-	-	Linear	-	1st-Order Lag	-
Pegasus [67]	Quadratic	-	-	-	Linear	-	-	-
Omnidrones [70]	Direct	-	-	Downwash	Random	-	-	-
NeuroBEM [74]	BEM	BEM	BEM	NN	NN	-	1st-Order Lag	-
Optimization [25, 26, 27]	Direct	-	-	-	Linear	-	-	-
Swift [34]	Quadratic	Fitting	Fitting	Fitting	Fitting	Gray-Box	1st-Order Lag	Linear
Penicka et al. [48]	Direct	-	-	-	-	-	-	-
Penicka et al. [49]	Quadratic	-	-	-	Linear	-	1st-Order Lag	-
MAVRL [41] (Evaluation)	Quadratic	Linear	-	-	-	-	1st-Order Lag	-
Kulkarni et al. [42]	Direct	-	-	-	-	-	-	-

2.3.2. Drone Physics Modeling

The central part of a multirotor is often modeled as a rigid body, whose linear and angular accelerations are induced by forces and torques produced by aerodynamic effects, rotor acceleration, and gravity. Once the forces and torques are known, the rigid body motion can either be simulated by a custom implementation of numerical integration, or by plugging the forces and torques into a physics engine. So essentially, drone physics modeling is about determining forces and torques acting on the central rigid body. A survey on autonomous drone racing [11] provides an overview of methods for modeling drone aerodynamics, batteries, and motors. Here we will follow a similar narration outline, and review methods adopted in the aforementioned simulators and research papers.

Aerodynamic forces and torques are generally caused by: (1) major actuation, (2) rotor drag, (3) dynamic lift, (4) rotor-rotor, rotor-body airflow interaction, and (5) body drag. At low flight speed, it is a popular choice to only consider major actuation and ignore other sources. A simple model is the quadratic model: the force and torque are proportional to the square of the propeller's angular velocity. However for high-speed flights, other sources could produce non-negligible aerodynamic effects, and there are many approaches towards more accurate modeling. For example, (2) and (5) could be modeled individually, often with linear or quadratic assumptions, and flow interactions (4) are ignored. Another choice is to combine first-principle with data-driven models. The combination is flexible and is usually tailored for different applications. In prioritizing model accuracy, NeuroBEM [74] combines a Blade-Element-Momentum (BEM) model capturing effects from (1) to (3), with a data-trained neural network (NN) for effects from (4) to (5). For better computing efficiency, Swift [34] combines a simple quadratic propeller model with a fitted polynomial model that accounts for all other sources.

Both the quadratic major actuation model and the BEM model depend on angular velocity as the input. A simple assumption is that the angular velocity is proportional to the normalized motor command. However most low-cost electronic speed controllers do not provide closed-loop control, and the angular velocity depends on both the motor command and the instantaneous battery voltage. On real hardware, one can measure the instantaneous voltage directly, in simulation, however, a battery model is needed. Bauersfeld et al. [75] propose a gray-box model that remains accurate even in experiments with highly varying power consumption.

The rotor is usually modeled as a first-order lag system with an adjustable time constant [61, 11, 74, 34]. In addition, rotor acceleration also causes torque, which is generally modeled as the product of the rotor moment of inertia and acceleration. This product is referred to as the "linear" rotor acceleration torque in Table 2.1, which summarises drone modeling methods in the previously reviewed literature. The "direct" model for major actuation refers to directly using the commanded rotor forces as the simulated rotor forces, without considering rotor rotation. A dash means "not considered".

2.4. Deep Reinforcement Learning

Deep reinforcement learning (Deep RL) represents a combination of two powerful paradigms in machine learning: reinforcement learning and deep learning. RL addresses the challenge of how agents can learn to make decisions through iterative trial and error. When deep learning is integrated into this framework, as in Deep RL, agents gain the ability to learn from high dimensional data, such as pixels rendered on a screen in a video game, and use neural networks to determine optimal actions to achieve specified objectives, such as maximizing game scores. This capacity has led to a diverse array of applications across multiple fields. The remaining part of the literature review will cover some highlight moments in the history of Deep RL, a mathematical definition of the RL problem, and algorithms for training policies.

2.4.1. Highlight Moments

The history of Deep RL is marked by a series of significant milestones that underscore its rapid evolution and impact across diverse domains. In 1992, the development of TD-Gammon [76] showcased one of the earliest successful applications of reinforcement learning with neural networks, demonstrating the potential of this framework in complex decision-making tasks. In 2003, Deep RL was applied to train a differential wheeled robot to complete the box-pushing task [77].

However, it wasn't until around 2013 when DeepMind's work in applying deep RL to play Atari video games captured widespread attention [78]. In 2015, AlphaGo's [79] historic victory against a human professional Go player marked a pivotal advancement, showcasing the prowess of deep RL in tackling challenges previously thought insurmountable. Subsequent projects, such as AlphaZero [80] in 2017 and MuZero [81] in 2019, further demonstrated Deep RL 's versatility across multiple board games, including chess and Shogi. Beyond simple Atari and board games, Deep RL agents have also reached human or super-human levels in games like Dota 2 [82] and Gran Turismo [83]. In the recent AI boom characterized by generative AI, RL was involved in the creation of the famous ChatGPT [84].

Deep Reinforcement Learning has also found widespread application in various other engineering disciplines, such as robotics and aerospace engineering. Notably, in robotics, a notable achievement is exemplified by OpenAI's Rubik's cube manipulator project [85]. In addition, Deep RL has emerged as a popular framework for tackling challenges in embodied AI [86] and visual navigation [53], offering effective solutions in these domains. Moreover, within aerospace engineering, Deep RL techniques have been successfully employed to control a diverse range of aircraft, spanning from high-altitude balloons [87] and unmanned aerial vehicles [34] to jets [88].

2.4.2. Problem Formulation

Mathematically, the (Deep) RL problem is often defined under the Markov Decision Process (MDP), which is a framework used to model decision-making in scenarios where the results of actions are partly random and partly under the control of a decision-maker, i.e. the agent. An MDP has the following elements:

- S: the set of valid states,
- A: the set of actions,
- R: the reward function $S \times A \times S \to \mathbb{R}$,
- P: the transition probability function $S \times A \to \mathcal{P}(S)$, with $\mathcal{P}$ denoting probability.

The agent takes actions $a \in A$ based on the policy π, which can be deterministic or stochastic. A deterministic policy is usually expressed as $\pi : S \to A$, and a stochastic one is usually $\pi : \mathcal{P}(a|s)$, meaning the probability of taking action a at state s. With a deterministic policy, the action is directly mapped from the state, while with a stochastic policy, the action is sampled from the policy probability distribution. Actions cause transitions of states, as described by the transition probability function S, forming a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$, which is often evaluated using either the infinite-horizon discounted total return $r(\tau)$ defined in equation (2.1) with the discount factor γ, or the finite-horizon total return defined in equation (2.2).

$$r(\tau) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \tag{2.1}$$

$$r(\tau) = \sum_{t=0}^{T} R(s_t, a_t, s_{t+1}) \tag{2.2}$$

The central problem is how to find the policy that maximizes the return expectation of the associated trajectory, which is mathematically described as:

$$\max_{\pi} \ \mathbb{E}_{\tau}[r(\tau)]. \tag{2.3}$$

The (Deep) RL problem is closely related to the optimal control problem, which concerns about finding the optimal control input sequence that drives the system to minimize a cost function. Sutton et al. [89] considered the field of modern RL the product of the joining of three threads: (1) learning by trial and error, (2) optimal control, and (3) temporal difference learning. One approach to tackling the optimal control problem is through dynamic programming [90], which relies on value functions to recursively solve the problem.

Throughout the development of RL algorithms, the concept of value functions, rooted in dynamic programming, holds significant importance. The value represents the expected return, considering the initial condition and the policy dictating subsequent actions and states. Values functions appear almost in every RL algorithm. The value function V of a state s, given the policy π, is defined as:

$$V_{\pi}(s) = \mathbb{E}_{\tau}[r(\tau)|s_0 = s]. \tag{2.4}$$

The action-value function Q of a state s and action a, given the policy π is defined as:

$$Q_{\pi}(s, a) = \mathbb{E}_{\tau}[r(\tau)|s_0 = s, a_0 = a]. \tag{2.5}$$

By the definition of expectation, we have the following relation between the two value functions:

$$V_{\pi}(s) = \mathbb{E}_{a}[Q_{\pi}(s, a)]. \tag{2.6}$$

Moreover, with the infinite-horizon discounted total return, the value functions can be expanded and rewritten as the Bellman equations to work with recursive algorithms. For the value function V, let a be the action of the current step and s' be the state at the next step:

$$V_{\pi}(s) = \mathbb{E}_{a,s'}[R(s, a, s') + \gamma V_{\pi}(s')]. \tag{2.7}$$

Similarly, for the action-value function Q, let a' denote the next action:

$$Q_{\pi}(s, a) = \mathbb{E}_{s'}[R(s, a, s') + \gamma \mathbb{E}_{a'}[Q_{\pi}(s', a')]]. \tag{2.8}$$

Finally, the difference between the action-value function and value function is defined as the advantage function in equation (2.9), which quantifies the superiority of taking a particular action a in state s compared to randomly selecting an action according to policy π and following the policy afterward.

$$A_{\pi}(s, a) = Q_{\pi}(s, a) - V_{\pi}(s) \tag{2.9}$$

This brief description of the problem formulation and important functions paves the way for understanding the reinforcement learning algorithms to be introduced below.

2.4.3. Algorithms

We will review algorithms following OpenAI's taxonomy [91], as illustrated in Figure 2.14. Algorithms are first categorized by whether they are model-free or model-based. A model-free algorithm doesn't learn or access the environment model, i.e. the state transition function and the reward function, while a model-based algorithm makes use of such information to make decisions. Then model-free ones are branched based on what they learn, and model-based algorithms are branched based on whether the model is given or not.

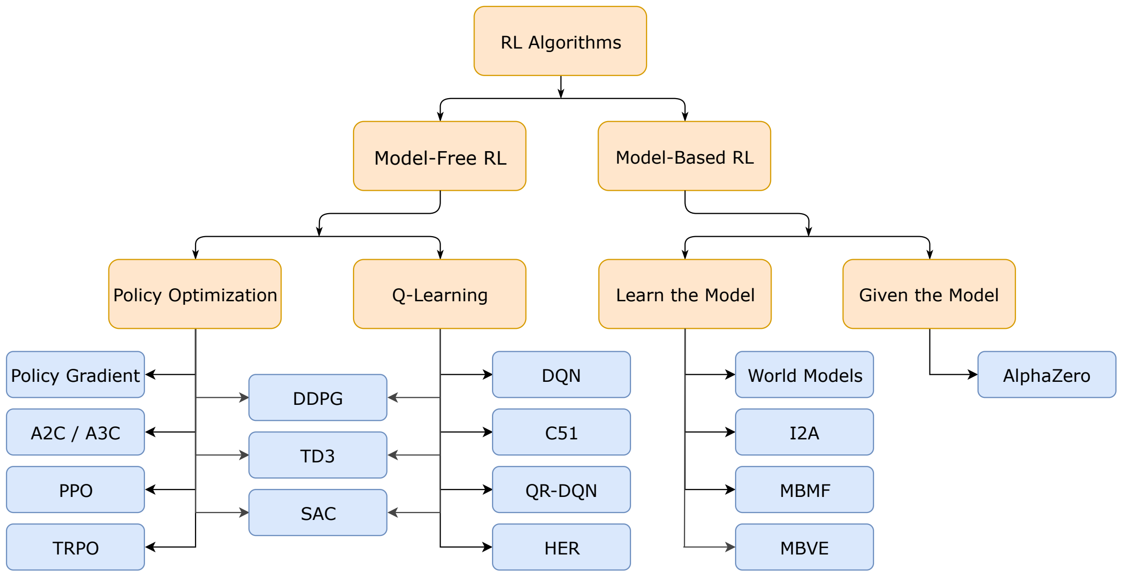


Figure 2.14: A non-exhaustive taxonomy of algorithms from OpenAI [91].

Model-based algorithms include World Models [92], Imagination-Augmented Agents (I2A) [93], Model-Based RL with Model-Free Fine-Tuning (MBMF) [94], Model-Based Value Expansion (MBVE) [95], and AlphaZero [80]. They work well and achieve high sample efficiency if the ground-truth model is known or the sim-to-real gap is not a concern. However, the accurate ground-truth model is hard to obtain and many real-world applications necessitate sim-to-real transfer, limiting the application of model-based algorithms.

Model-free algorithms have lower sample efficiency, compared to model-based ones, but are generally easier to implement, and have seen wider application [91]. Algorithms that fall into the policy optimization side learn the approximated value function V_π and use it to optimize policy parameters for the maximum return $r(\tau)$. They are also known as "policy-based" methods. On the Q-learning side, algorithms try to learn the parameters θ for approximating the optimal action-value function Q^*_π, making $Q_\theta \to Q^*_\pi$. So are also referred to as "value-based" algorithms. In this way, the policy is to take actions that maximize the action-value function:

$$a(s) = \arg\max_a Q_\theta(s, a). \tag{2.10}$$

Policy optimization is usually performed on-policy, where each iteration utilizes only the trajectory generated by the most recent policy and excludes old data. Notable policy-based algorithms include Policy Gradient (PG) [96], Trust Region Policy Optimization (TRPO) [97], Proximal Policy Optimization (PPO) [98], as well as Advantage Actor-Critic (A2C/A3C) [99]. This on-policy approach often results in lower sample efficiency compared to the off-policy approach, which leverages past data and is often seen within the value-based algorithms. However, since policy optimization directly optimizes for the maximum reward, policy-based algorithms have better stability and reliability. On the other hand, value-based algorithms like Deep Q-Network (DQN) [78], Quantile Regression DQN (QR-DQN) [100], Categorical 51-Atom DQN (C51) [101], and Hindsight Experience Replay (HER) [102] are known for their sample efficiency, but they do not guarantee achieving the maximum return [103].

Besides algorithms that can be clearly classified as policy-based or value-based, there are plenty of algorithms that are blends of both, carefully trading off the advantages and disadvantages of each side. Typical examples are Deep Deterministic Policy Gradient (DDPG) [104], Twin Delayed DDPG (TD3) [105], and Soft Actor-Critic (SAC) [106].

We will focus on introducing PPO because it is the most adopted algorithm in previously reviewed work on learning-based autonomous drone racing, moreover, it is close to state of the art on stability and sample efficiency among policy-learning algorithms.

The line of research that has evolved into PPO consists primarily of three algorithms, ordered chronologically which are PG, TRPO, and PPO. PG establishes the framework involving (1) policy gradient calculation, (2) policy parameter update, and (3) value function parameter update. TRPO improves PG by introducing a more complex policy parameter update step that takes the largest step size within the Kullback–Leibler divergence (KLD) constraint. The good side of calculating such a "trust region" is that it effectively avoids bad steps that can collapse the policy performance. But the downside is that TRPO has increased computational complexity. PPO offers a simpler, more efficient, yet effective alternative to TRPO.

In PG and TRPO, denoting the objective $\mathbb{E}_\tau[r(\tau)]$ to maximize as $J(\pi_\theta)$, where θ is the parameters of the policy network, the policy gradient has the general form:

$$\nabla_\theta[J(\pi_\theta)] = \mathbb{E}_\tau\left[\sum_{t=0}^{T} \nabla_\theta[\log \pi_\theta(a_t|s_t)] \cdot \Phi_t\right]. \tag{2.11}$$

There are many valid options for Φ_t, with the advantage function $\Phi_t = A_{\pi_\theta}(s_t, a_t)$ being a widely adopted choice. Estimating the advantage function A requires estimating the value function, which is often represented by a network parameterized by ϕ, written as V_ϕ. Practically, the expectation in equation (2.11) is approximated using values of a finite set of trajectories. With the approximated policy gradient, the algorithms proceed to the policy parameter update step. In the simplest form, this step is:

$$\theta' = \theta + \alpha \nabla_\theta[J(\pi_\theta)]. \tag{2.12}$$

PG does not set constraints over the learning rate α, but TRPO carefully selects the learning rate to take the largest yet trusted step. Finally, by the definition of the value function, the parameters of the value function ϕ network are updated by making $V_\phi(s_t)$ close to $\hat{r}(t) \coloneqq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})$:

$$\min_\phi \; \mathbb{E}_\tau\left[\sum_{t=0}^{T} (V_\phi(s_t) - \hat{r}(t))^2\right]. \tag{2.13}$$

The updated ϕ will be used in the next iteration to estimate the policy gradient.

Although TRPO has the same general form as PG, equations for TRPO are derived from the surrogate advantage $\mathcal{L}(\theta, \theta')$ that quantifies how much better the new policy $\pi_{\theta'}$ performs compared to the current policy π_θ. This being said, the goal is to find the new parameters θ' that maximize the surrogate advantage while conforming to the KLD constraint:

$$\begin{aligned} \max_{\theta'} \; & \mathcal{L}(\theta, \theta') = \mathbb{E}_{s,a}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A_{\pi_\theta}(s, a)\right] \\ & \text{s.t. } \overline{D}_{\text{KL}}(\theta' \,||\, \theta) \le \delta, \end{aligned} \tag{2.14}$$

where $\overline{D}_{\text{KL}}$ is the average KLD, and δ is the hyper-parameter limiting the average KLD. The idea behind the surrogate advantage is that if the advantage of a state-action pair is positive, then we want to increase the probability of generating such a pair with the new policy as much as possible, i.e. $\pi_{\theta'}(a|s) > \pi_\theta(a|s)$, which is equivalent to maximizing the surrogate advantage. Similarly, if the advantage is negative, then we want to decrease the probability. But at the same time, we don't want to step too far from the old policy to avoid "collapsing" performance. So we have the KLD of two sets of parameters constrained by δ.

In practice, to simplify computation, these equations are approximated with their Taylor expansions:

$$\begin{aligned} \mathcal{L}(\theta, \theta') &\approx \nabla_\theta [J(\pi_\theta)]^\intercal \cdot (\theta' - \theta) \\ \overline{D}_{\text{KL}}(\theta' \,||\, \theta) &\approx \frac{1}{2}(\theta' - \theta)^\intercal \cdot \nabla_\theta^2 [\overline{D}_{\text{KL}}(\theta' \,||\, \theta)] \cdot (\theta' - \theta). \end{aligned} \tag{2.15}$$

To this end, the approximated solution of equation (2.14) can be analytically derived using Lagrangian duality. Together with backtracking line search, the policy parameter update rule can be expressed in the form of equation (2.12). Finding the analytical solution involves calculating the inverse (often via conjugate gradient) of the KLD's Hessian matrix, which is quite expensive. This is the major problem that motivated the development of PPO.

There are two variants of PPO, namely PPO with penalty and PPO with clipping. PPO with penalty turns the hard KLD constraint into a soft penalty term that discourages large KLD between the new and old parameters. For this variant, equation (2.14) is turned into:

$$\max_{\theta'} \; \mathcal{L}(\theta, \theta') = \mathbb{E}_{s,a} \left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A_{\pi_\theta}(s,a) - \beta \overline{D}_{\text{KL}}(\theta' \,||\, \theta) \right]. \tag{2.16}$$

The coefficient β is adjusted by the algorithm throughout training. PPO with clipping, as the name suggests, clips the surrogate advantage so that the optimizer will not push excessively hard to maximize the advantage, and thus the new policy will not deviate too much from the old. The surrogate advantage is then:

$$\max_{\theta'} \; \mathcal{L}(\theta, \theta') = \mathbb{E}_{s,a} \left[\min \left(\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A_{\pi_\theta}(s,a), \text{clip} \left(\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}, 1-\epsilon, 1+\epsilon \right) A_{\pi_\theta}(s,a) \right) \right], \tag{2.17}$$

where ϵ denotes the hyper-parameter to tune. By removing the hard KLD constraint, both variants avoid expensive calculations and thus lower the computational overhead. Nevertheless, PPO with clipping is generally more favored for its simplicity, ease of implementation, and broadly demonstrated effectiveness.

2.5. Conclusion

In conclusion, this chapter has provided an overview of key aspects within the domain of autonomous drone racing and its associated fields. We began by elucidating the landscape of autonomous drone racing competitions and the solutions proposed therein, spanning both obstacle-free and obstacle-aware scenarios. This exploration offers a comprehensive understanding of the advancements made in this field, serving as a foundation for this thesis project.

Recognizing the limited generalization ability of learning-based methods for obstacle-aware autonomous drone racing, we examined research on general-purpose vision-based navigation via Deep RL to gain insights for enhancing behavioral obstacle avoidance in intelligent agents. The network architectures, training frameworks, and domain randomization schemes discussed in the reviewed papers provide valuable inspiration for this purpose.

Moving forward, we delved into the topic of modeling and simulation, highlighting the diverse array of simulators available for aerial and general robotics. By examining existing quadcopter physics modeling within these simulators and previously reviewed work, we have a clear vision for building a simulator tailored to specific task requirements.

Furthermore, our exploration extended into the domain of deep reinforcement learning, motivated by its significance and applicability within the context of autonomous drone racing. We have focused on the line of research, from PG and TRPO to PPO, due to PPO's significance, popularity, and suitability for RL-based autonomous drone racing.

To end this chapter, we provide the answer to Research Question 1 based on the reviewed literature. For readers' convenience, the question is restated below.

Research Question 1

What are existing methods in the literature that can achieve the objective in whole or partially?

To the best of the author's knowledge, no existing method in the literature fully achieves the desired objective. However, several learning-based approaches partially address it. For instance, the state-based drone racing policy by Song et al. [32] demonstrates generalization to unseen racing tracks but does not account for obstacle avoidance. On the other hand, the vision-based policy [51] for drone racing in cluttered environments strikes a balance between high-speed racing and obstacle avoidance, but it compromises on generalization ability. Methods like MAVRL [41] and the navigation approach using DCE [42] enable drones to navigate unseen cluttered environments but are not specifically optimized for racing and rely on high-level controllers, offloading the complex control tasks from the policy itself. This gap in the literature opens the door for new methodological proposals that could fully achieve the research objective, addressing all the requirements.

3 Preliminary Work

3.1. Overview of Proposed Methodology

The proposed methodology is presented first as an overview to provide essential guidance for the practical implementations introduced in the subsequent sections. While this section outlines the high-level approach, more detailed explanations of the methodology can be found in the scientific article in Part III. As concluded in the previous chapter, no existing method fully satisfies the research objective, highlighting the need for novel methodological proposals and opening opportunities for innovation.

The core of our methodology is built upon three primary components. First and foremost, we propose incorporating domain randomization of racing tracks and environments to promote the learning of generalizable policies. This approach is motivated by the success of similar techniques in the works of Song et al. [32], MAVRL [41], and Kulkarni et al. [42]. Domain randomization helps the reinforcement learning agent to adapt to varied conditions.

Secondly, given that both vehicle state information and waypoint data are accessible, we propose training a hybrid-input policy solely through reinforcement learning, in contrast to approaches that combine state-based reinforcement learning with imitation learning, as demonstrated in [51]. The policy network we employ will accept three types of input concatenated as a vector: (1) an encoded depth image, (2) the vehicle's state vector, and (3) waypoint information for two future waypoints. With these inputs, the policy can hopefully learn to navigate complex environments effectively.

Lastly, to maintain the agility of the drone, we propose relying only on a low-level angular velocity controller, similar to the approach used in Swift [34]. We also plan to design a reward function inspired by those used in the drone racing literature [32, 34, 51]. This approach allows for high levels of maneuverability, avoiding the more restrictive controllers used in MAVRL and Kulkarni et al., while still providing an effective framework for training agile flight behaviors.

Implementing this proposed methodology involves non-trivial engineering challenges. Two key tasks are: (1) creating the environment for reinforcement learning and (2) integrating the necessary components to form a complete training loop. This requires either implementing or effectively utilizing existing libraries to handle the various aspects of reinforcement learning, including environment management, neural network construction, gradient calculation, and model parameter optimization. These efforts constitute the primary focus of the remainder of this chapter.

3.2. Environment for Reinforcement Learning

We propose using depth images for obstacle avoidance, prioritizing high rendering speed over visual fidelity. Given this, Isaac Gym [73] and Habitat Sim [72] are both strong candidates. Habitat Sim claims to reach higher rendering speeds, but it is primarily designed for ground robots navigating in mesh datasets like Gibson [107] and Matterport3D [108], making it less adaptable for drone racing tasks. In contrast, Isaac Gym offers similarly high rendering speeds, extensive documentation for object spawning and pose customization, and, most importantly, examples for developing custom environments, including the Aerial Gym [109] implementation. As a result, Isaac Gym was chosen as the base simulator for this thesis project.

At the start of this thesis project, no existing drone racing environments were available in Isaac Gym, and Aerial Gym was still in its early development stages. Therefore, we opted to develop our custom environments using the bare-bones Isaac Gym simulator. Our environment is composed of multiple modules optimized with vectorized operations, and peripheral utilities extending Isaac Gym's scene simulation ability. They will be introduced in the following subsections.

3.2.1. Angular Velocity Controller

The angular velocity controller employed is a PID controller modeled after the Betaflight [50] controller, which is widely used in FPV drone systems. We examined Betaflight's original C code and replicated its core functionalities in Python using PyTorch, enabling the simultaneous control of multiple drones.

This angular velocity controller processes normalized operator commands, either from the neural policy or a human pilot, and outputs the desired angular velocity in the drone's body frame as well as normalized motor commands. The process involves three main steps: (1) mapping operator commands to desired angular velocity, (2) computing the PID control sum, and (3) performing control allocation or mixing.

In the first step, operator commands are mapped to the desired angular velocity, a process referred to as "rates mapping" in Betaflight. We implemented the default mapping known as Actual Rates [110] from Betaflight. Given $\boldsymbol{a} := [\boldsymbol{a}_{\text{rate}}^\intercal \, a_{\text{throttle}}]^\intercal \in [-1, 1]^4$ as the operator commands and $\boldsymbol{K}_{\text{d}}$, $\boldsymbol{K}_{\text{f}}$, $\boldsymbol{K}_{\text{g}}$ as adjustable coefficients, the mapping from $\boldsymbol{a}_{\text{rate}}$ to the desired angular velocity in the body frame $\boldsymbol{\omega}_{\text{des}}$ is given by:

$$\boldsymbol{\omega}_{\text{des}} = \text{sgn}(\boldsymbol{a}_{\text{rate}}) \left(\boldsymbol{K}_{\text{d}} \, |\boldsymbol{a}_{\text{rate}}| + (\boldsymbol{K}_{\text{f}} - \boldsymbol{K}_{\text{d}}) \left((1 - \boldsymbol{K}_{\text{g}}) \boldsymbol{a}_{\text{rate}}^2 + \boldsymbol{K}_{\text{g}} \boldsymbol{a}_{\text{rate}}^6 \right) \right) . \tag{3.1}$$

In the second step, the PID control sum $\boldsymbol{u}_{\text{PID}}$ at time t is calculated using:

$$\boldsymbol{u}_{\text{PID}}(t) = \boldsymbol{K}_{\text{P}} \boldsymbol{e}_{\boldsymbol{\omega}}(t) + \boldsymbol{K}_{\text{I}} \int_0^t \boldsymbol{e}_{\boldsymbol{\omega}}(\tau) \, d\tau - \boldsymbol{K}_{\text{D}} \mathcal{L} \left(\frac{d\boldsymbol{\omega}(t)}{dt} \right) + \boldsymbol{K}_{\text{FF}} \boldsymbol{\omega}_{\text{des}}(t), \tag{3.2}$$

where $\boldsymbol{K}_{\text{P}}$, $\boldsymbol{K}_{\text{I}}$, $\boldsymbol{K}_{\text{D}}$, and $\boldsymbol{K}_{\text{FF}}$ are diagonal coefficient matrices for the proportional, integral, derivative, and feed-forward terms, respectively. Here, $\boldsymbol{e}_{\boldsymbol{\omega}}$ represents the angular velocity error, and $\mathcal{L}$ denotes a simple first-order low-pass filter in the time domain. This step is simplified compared to Betaflight's implementation, which includes more complex feed-forward terms and dynamic coefficient updating algorithms, such as anti-gravity compensation, throttle PID attenuation, I-term relaxation, and dynamic damping.

In the final step, the PID control sum is converted into normalized motor commands through a mixing function that supports various Betaflight features, including airframe customization, air-mode, throttle boost, and thrust linearization. Airframe customization allows for the mixing table to adapt to different motor layouts, air-mode maintains attitude control at low throttle values, throttle boost enhances response to high-frequency throttle commands, and thrust linearization ensures consistent control at extreme throttle levels. The mixing function $\mathcal{M}$ calculates the motor commands $\boldsymbol{u}_{\text{motor}} \in [0, 1]^4$ as follows:

$$\boldsymbol{u}_{\text{motor}} = \mathcal{M}(\boldsymbol{u}_{\text{PID}}, a_{\text{throttle}}). \tag{3.3}$$

The mixing function involves several clamping operations and min-max normalizations, which are detailed in Listing A.1.

3.2.2. Drone Model

The drone is modeled as a rigid body with a rectangular collision box in PhysX. To simulate the dynamics of the drone, there are two common approaches: (1) teleporting the body to the desired poses while using a standalone numerical integrator for rigid body dynamics, or (2) applying wrenches (forces and torques) directly to the body, allowing the PhysX solver to handle the forward dynamics. For simplicity, we opted for the latter approach. We developed modules to convert motor commands into total actuator wrenches and implemented a basic drag model, which adds additional forces and torques based on the linear and angular velocities in the body frame.

First, the motor commands are translated into rotor (motor and propeller) angular velocities, $\boldsymbol{\Omega}$, using a first-order lag model:

$$\dot{\boldsymbol{\Omega}} = \boldsymbol{K}_{\text{rotor}} \cdot (\boldsymbol{\Omega}_{\text{ss}}(\boldsymbol{u}_{\text{motor}}) - \boldsymbol{\Omega}), \tag{3.4}$$

where $\boldsymbol{K}_{\text{rotor}}$ represents the spin-up or slow-down constant, which depends on the difference between the steady-state angular velocity, $\boldsymbol{\Omega}_{\text{ss}}$, and the current angular velocity, $\boldsymbol{\Omega}$. The steady-state angular velocity is determined by the motor commands $\boldsymbol{u}_{\text{motor}}$ from the angular velocity controller, using a polynomial model fitted to the motor manufacturer's data.

Next, based on the current rotor angular velocities $\boldsymbol{\Omega}$ and their time derivatives $\dot{\boldsymbol{\Omega}}$, the total actuator wrenches generated by the rotors are computed as follows:

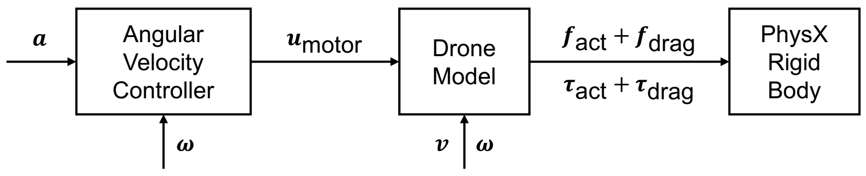
$$\begin{bmatrix} \boldsymbol{f}_{\text{act}} \\ \boldsymbol{\tau}_{\text{act}} \end{bmatrix} = \begin{bmatrix} \sum_i \boldsymbol{f}_{\text{prop}}(\boldsymbol{\Omega}_i) \\ \sum_i \left[\boldsymbol{\tau}_{\text{prop}}(\boldsymbol{\Omega}_i) + \boldsymbol{r}_i \times \boldsymbol{f}_{\text{prop}}(\boldsymbol{\Omega}_i) + \zeta_i \boldsymbol{J}_{\text{rotor}} \dot{\boldsymbol{\Omega}}_i \right] \end{bmatrix}, \tag{3.5}$$

where $\boldsymbol{f}_{\text{prop}}(\boldsymbol{\Omega}_i)$ and $\boldsymbol{\tau}_{\text{prop}}(\boldsymbol{\Omega}_i)$ represent the thrust force and torque generated by the i-th spinning propeller, respectively. The mappings from $\boldsymbol{\Omega}$ to $\boldsymbol{f}_{\text{prop}}$ and $\boldsymbol{\tau}_{\text{prop}}$ are based on polynomial models derived from the motor manufacturer's data. Here, $\boldsymbol{r}_i$ is the displacement of the i-th rotor relative to the drone's body frame, $\boldsymbol{J}_{\text{rotor}}$ denotes the moment of inertia of the rotor (which includes the propeller and the spinning parts of the motor), and ζ_i indicates the rotational direction of the i-th rotor.

Finally, we compute the aerodynamic drag acting on the drone's body, which depends on the linear velocity, $\boldsymbol{v}$, and the angular velocity, $\boldsymbol{\omega}$, in the body frame:

$$\begin{bmatrix} \boldsymbol{f}_{\text{drag}} \\ \boldsymbol{\tau}_{\text{drag}} \end{bmatrix} = -\frac{1}{2}\rho \begin{bmatrix} A_{\text{tran}} \left(\boldsymbol{C}_{\text{a}} \boldsymbol{v}^2 + \boldsymbol{C}_{\text{b}} \boldsymbol{v} \right) \\ A_{\text{rot}} \left(\boldsymbol{C}_{\text{c}} \boldsymbol{\omega}^2 + \boldsymbol{C}_{\text{d}} \boldsymbol{\omega} \right) \end{bmatrix}, \tag{3.6}$$

where ρ is the air density, A_{tran} and A_{rot} are the effective areas responsible for generating translational and rotational drag, respectively, and $\boldsymbol{C}_{\text{a}}$, $\boldsymbol{C}_{\text{b}}$, $\boldsymbol{C}_{\text{c}}$, and $\boldsymbol{C}_{\text{d}}$ are adjustable coefficients in the polynomial drag model. The aerodynamic drag wrench is then summed with the total actuator wrench and applied to the gravity-enabled PhysX rigid body. Figure 3.1 illustrates the data flow in the drone simulation pipeline.



$\boldsymbol{a}$
Angular Velocity Controller
$\boldsymbol{u}_{\text{motor}}$
Drone Model
$\boldsymbol{f}_{\text{act}} + \boldsymbol{f}_{\text{drag}}$
$\boldsymbol{\tau}_{\text{act}} + \boldsymbol{\tau}_{\text{drag}}$
PhysX Rigid Body
$\boldsymbol{\omega}$
$\boldsymbol{v}$
$\boldsymbol{\omega}$

Figure 3.1: Data flow in the drone simulation pipeline.

All of these models have been implemented using PyTorch and optimized for GPU-based parallel physics simulation. Moreover, the model's coefficients and parameters provide a wide scope for domain randomization in reinforcement learning, potentially enabling the training of robust policies that can generalize well.

3.2.3. Waypoints

The primary task in drone racing is to navigate through all gates on a track, passing from the correct side of each gate to the other. Each gate is modeled as a "waypoint" and is parameterized by its center position, $\boldsymbol{p}_{\text{wp}}$, orientation quaternion, $\boldsymbol{q}_{\text{wp}}$, passing region width, w_{wp}, passing region height, h_{wp}, and a binary parameter, χ_{wp}, which indicates the presence of physical bars around the waypoint. This flexible parameterization allows for the definition of various types of waypoints:

- Points: $w_{\text{wp}} = h_{\text{wp}} = \chi_{\text{wp}} = 0$;
- Bounded planar regions: $w_{\text{wp}} \neq 0$, $h_{\text{wp}} \neq 0$, $\chi_{\text{wp}} = 0$;
- Collidable gates: $w_{\text{wp}} \neq 0$, $h_{\text{wp}} \neq 0$, $\chi_{\text{wp}} \neq 0$.

With this waypoint definition, a racing track is constructed as an ordered sequence of waypoints. To make track definition more intuitive, our implementation supports the use of Euler angles as an alternative to quaternions for defining waypoint orientations. An example of defining the Split-S track [34] is provided in Listing A.2.

Inspired by the work of Song et al. [32], we implemented a random track generator that can create varied or uniform racing tracks across multiple parallel environments. Our implementation parameterizes the transformation between consecutive waypoints using five variables, allowing for intuitive adjustment of the track's difficulty and enough room for randomization. The parameters include four angles, $(\psi, \theta, \alpha, \gamma)$, and a distance, r. The pose of the next waypoint is determined by starting with the current waypoint's pose and applying the following transformations:

1. Rotate about the body z-axis by angle ψ;
2. Rotate about the updated body y-axis by angle θ;
3. Translate in the direction of the updated body x-axis by distance r;
4. Rotate about the updated body x-axis by angle α;
5. Rotate about the updated body y-axis by angle γ.

These transformations, excluding α, and their corresponding effects are illustrated in Figure 3.2. Generally, for a fixed distance r, the track becomes more challenging as the angles increase. Similarly, if the angles are held constant, the track's difficulty increases when r exceeds an upper threshold or falls below a lower threshold. This implementation is also useful for curriculum learning, where task difficulty needs to be progressively adjusted.

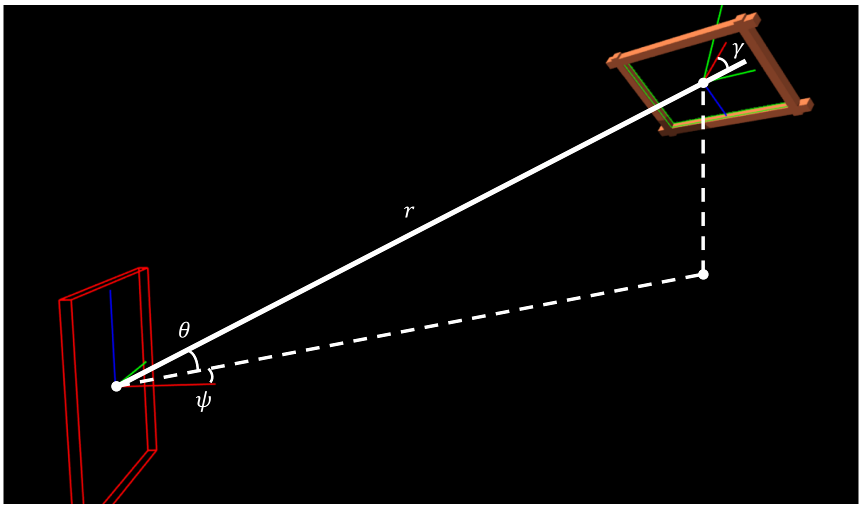


Figure 3.2: Partial parameters of the transform between two consecutive waypoints.

3.2.4. Procedural Assets

Isaac Gym provides limited functionality for spawning procedural assets. To overcome this limitation, we extended it by implementing additional interfaces that allow for the procedural generation of a wider variety of geometries, assemblies, and racing tracks. This enhancement significantly increases the achievable randomness of the environments and makes it easy to design racing tracks. Leveraging Isaac Gym's ability to load assets from URDF files, we use the Urdfpy package to procedurally create URDF files, which are then loaded into Isaac Gym.

Isaac Gym's built-in procedural assets include cuboids, capsules, and spheres, as shown in Figure 3.3(a). In addition to these, we implemented functions to spawn more complex shapes, such as cylinders, hollow cuboids, cuboid wireframes, abstracted trees from Aerial Gym [109], and racer quadcopter models, as illustrated in Figure 3.3(b)-(c).

Beyond basic geometries and abstracted assemblies like trees and racing drones, we also provide several racing track assets, as shown in Figure 3.4. New tracks can be easily created by defining a list

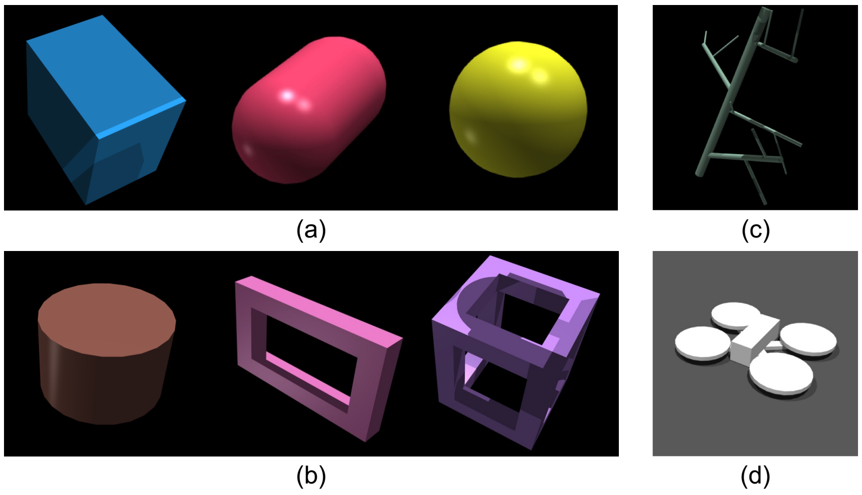


Figure 3.3: Extended set of assets for our drone racing environment.

of waypoints and, optionally, adding obstacles as URDF links. An example of the code for defining such obstacles is provided in Listing A.3.

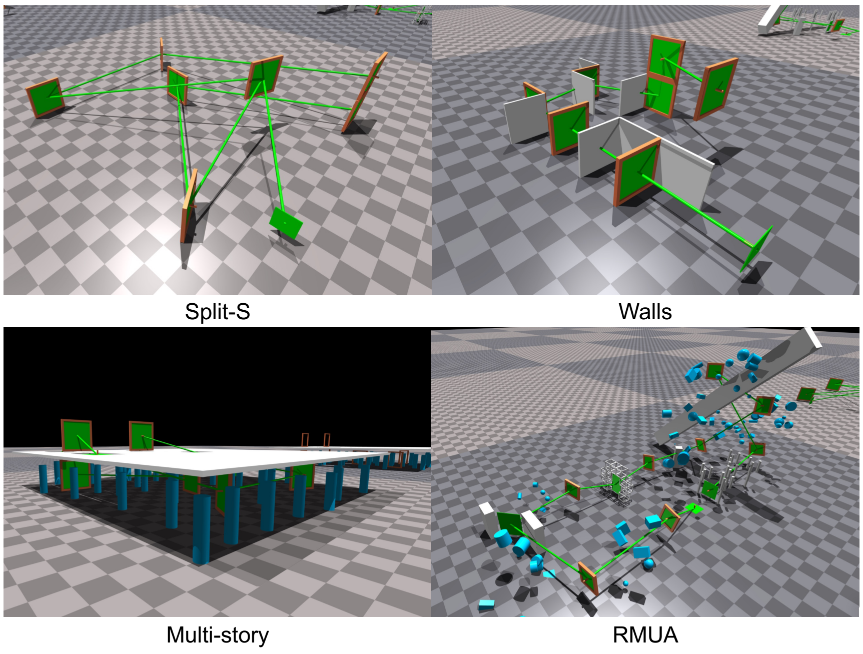


Figure 3.4: Racing track assets.

3.2.5. Obstacle Manager

Song et al. [32] suggest that training agents on multiple randomly generated tracks fosters the development of generalizable skills for obstacle-free drone racing. We propose extending this idea to obstacle-aware drone racing, which requires not only a random track generator but also a module to place random obstacles along the track. This task is handled by the obstacle manager.

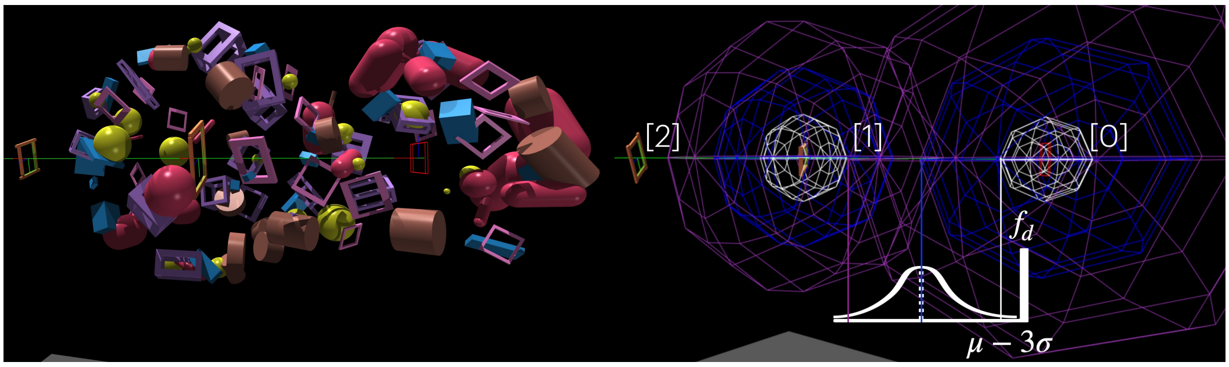
The obstacle manager places obstacles in such a way that the flight paths between waypoints are “efficiently” influenced while allowing control over the level of difficulty. Here, efficiency refers to the ratio of obstacles that potentially obstruct flight paths to the total number of obstacles. A higher efficiency translates to faster simulation speeds for the same level of difficulty, as fewer objects are in the environment.

A naive approach would involve distributing obstacles uniformly in space with random poses. However, this method is highly inefficient: obstacles positioned far from the waypoints or the segments connecting them are less relevant to the problem. A simple improvement to efficiency is to confine obstacles to regions where the racing drone is likely to fly. In our implementation, obstacles of various shapes orbit around waypoints, obstructing omnidirectional flight paths, while walls and trees are placed along the segments connecting waypoints.

Random cuboids, spheres, capsules, cylinders, hollow cuboids, and cuboid wireframes are managed to orbit waypoints, as illustrated in Figure 3.5(a). They are distributed uniformly in all directions around the waypoint, with their distances from the central waypoint, denoted as d, following a normal distribution. For obstacles orbiting a waypoint with index i, the normal distribution is determined by the distance between waypoint i and the next waypoint $i+1$, denoted as r_i, along with the radii of the no-obstacle safe zones around both waypoints, labeled s_i and s_{i+1}:

$$\mu = \frac{1}{2}(r_i + s_i - s_{i+1}),\ \sigma = \frac{1}{6}(r_i - s_i - s_{i+1}), \tag{3.7}$$

where μ and σ represent the mean and standard deviation of the normal distribution, respectively. These parameters are visualized in Figure 3.5(b), where the safe zones around each waypoint are outlined in white wireframes. This strategy ensures that obstacles are positioned relative to the track which effectively challenges the drone to learn obstacle avoidance.



(a) (b)

Figure 3.5: Obstacles orbiting waypoints of a straight racing track.

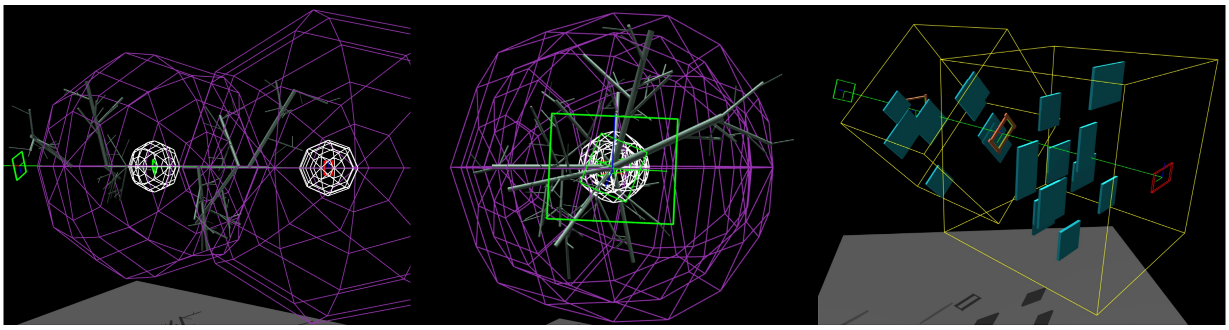

(a) (b) (c)

Figure 3.6: Trees and walls managed by the obstacle manager.

Trees are rooted along the line segments connecting the waypoints. The distances of the tree roots from the starting waypoint, indexed as i, are uniformly sampled from the range between s_i and $r_i - s_{i+1}$, as shown in Figure 3.6(a). Each tree's orientation in space is defined by a random angle, uniformly sampled between 0 and 2π, as depicted in Figure 3.6(b). Walls, represented by thin cuboids, are also distributed along the line segments. Their orientations are aligned with that of the starting waypoint for simplicity. Figure 3.6(c) illustrates how the walls are placed along the track to further challenge the drone's maneuverability.

Figure 3.5 and 3.6 demonstrate obstacles placed along a straight-line track and visualize different collections of obstacles for clarity. Now we can put everything including the random track generator and all kinds of obstacles together to create the complete scene. Figure 3.7 demonstrates that the current implementation of the track generator and obstacle manager can create very challenging scenes for obstacle-aware drone racing.

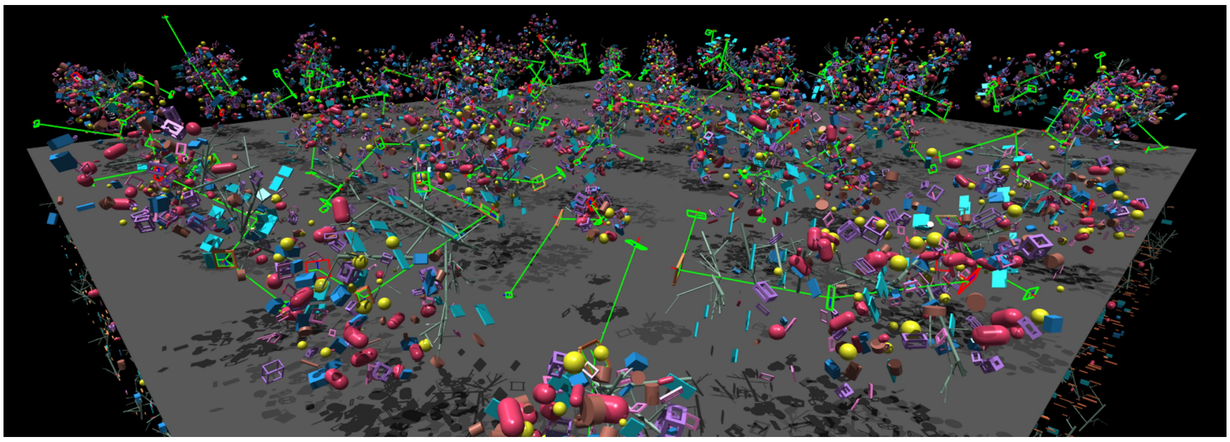


Figure 3.7: Randomly generated racing tracks with all obstacles enabled in parallel environments.

3.2.6. Isaac Gym Environment

Our environment, referred to as a "task" in the `IsaacGymEnvs` framework, inherits from the `VecTask` class to integrate the previously mentioned modules alongside other utility components. However, the base `VecTask` class is not natively designed to handle physics simulation in conjunction with a low-level controller and isn't optimized for learning from vision-involved inputs. To address these limitations, we put in extra engineering effort to re-implement several functions. Additionally, the asynchronous auto-resetting of environments posed further challenges in managing resets efficiently.

The `step` function receives the policy's action, then transitions the environment to a new state, and returns information including the reward, observation, termination flags, and auxiliary data for logging or curriculum learning. Given the auto-resetting nature of the environments, care must be taken to compute observations only after any necessary resets to avoid extra rendering. The `step` function can be summarized as follows:

1. Clamp the input action and pass it to the angular velocity controller as the setpoint.
2. For multiple physics steps: (a) run the angular velocity controller and drone model to compute the forces and torques to be applied to the rigid bodies, (b) call the PhysX simulation API to update the rigid body states, and (c) check for collisions.
3. Check the `done` condition. An environment is considered `done` if any of the following occur: (a) the final waypoint is reached, (b) a timeout occurs where too many steps have been taken without reaching the final waypoint, or (c) the drone collides with an obstacle.
4. Compute the reward components and sum them into a scalar reward.
5. Reset environments that have reached the `done` state. This involves teleporting the drone to its initial state and clearing memory variables in all relevant modules.
6. Render camera sensor data and calculate the observation vector. These steps occur after resetting to comply with the auto-reset requirements.
7. Return the observation, reward, `done` flags, and any additional information.

In addition to resets occurring within the `step` function, environments are also reset at the start of each rollout via the `reset` function. The primary objective of the `reset` function is to initialize the environments for the rollout and provide the initial observation. In our implementation, this also includes randomizing the racing tracks and obstacle positions.

3.3. Reinforcement Learning Library

There are several reinforcement learning libraries available, including RL Games [111], Stable-Baselines3 [112], CleanRL [113], SKRL [114], and TorchRL [115]. RL Games is the default library bundled with `IsaacGymEnvs`, but it lacks comprehensive documentation and tutorials. Stable-Baselines3 is a widely used library; however, it relies on NumPy for its data interface, requiring an additional step to convert PyTorch data to NumPy, which may introduce inefficiencies. Both SKRL and CleanRL offer strong documentation and directly use PyTorch tensors, but neither seamlessly integrates into the repository structure of `IsaacGymEnvs`. TorchRL, despite receiving considerable attention and providing excellent documentation, is still in an early stage of development.

Eventually, we opted for RL Games version `1.6.1` after encountering several issues with TorchRL version `0.5`. Due to the limited documentation of RL Games, we had to rely on directly reading the source code to learn to customize the training loop and define new network architectures. To assist other RL practitioners who may face similar challenges, we hereby document our insights and practical tips for working with RL Games in `IsaacGymEnvs`, which, in the author's view, contribute valuable additions to this thesis project.

3.3.1. Program Entry Point

The primary entry point for both training and testing within `IsaacGymEnvs` is the `train.py` script. This file initializes an instance of the `rl_games.torch_runner.Runner` class, and depending on the mode selected, either the `run_train` or `run_play` function is executed. Additionally, `train.py` allows for custom implementations of training and testing loops, as well as the integration of custom neural networks and models into the library through the `build_runner` function, a process referred to as "registering". By registering custom code, the library can be configured to execute the user-defined code by specifying the appropriate names within the configuration file. Environments also need to be registered in `train.py` so that they can be used in the training or testing loop.

In RL Games, the training algorithms are referred to as "agents", while their counterparts for testing are known as "players". In the `run_train` function, an agent is instantiated, and training is initiated through the `agent.train` call. Similarly, in the `run_play` function, a player is created, and testing begins by invoking `player.run`. Thus, the core entry points for training and testing in RL Games are the `train` function for agents and the `run` function for players.

```python
def run_train(self, args):
    """Run the training procedure from the algorithm passed in."""

    print('Started to train')
    agent = self.algo_factory.create(self.algo_name, base_name='run', params=self.params)
    _restore(agent, args)
    _override_sigma(agent, args)
    agent.train()

def run_play(self, args):
    """Run the inference procedure from the algorithm passed in."""

    print('Started to play')
    player = self.create_player()
    _restore(player, args)
    _override_sigma(player, args)
    player.run()
```

Listing 3.1: Implementation of `run_train` and `run_play`.

3.3.2. Training Algorithms

The creation of an agent is handled by the `algo_factory`, as demonstrated in Listing 3.1. By default, the `algo_factory` is registered with continuous-action A2C, discrete-action A2C, and SAC. This default registration is found within the constructor of the `Runner` class, and its implementation is shown in Listing 3.2. From this code, it's easy to trace the actual algorithm implementations. Our primary focus will be on understanding `A2CAgent`, as it is the primary algorithm used for most continuous-control tasks in `IsaacGymEnvs`.

```python
self.algo_factory.register_builder('a2c_continuous',
                                   lambda **kwargs: a2c_continuous.A2CAgent(**kwargs))
self.algo_factory.register_builder('a2c_discrete',
                                   lambda **kwargs: a2c_discrete.DiscreteA2CAgent(**kwargs))
self.algo_factory.register_builder('sac',
                                   lambda **kwargs: sac_agent.SACAgent(**kwargs))
```

Listing 3.2: Default algorithms in RL Games.

At the base of all RL Games algorithms is the `BaseAlgorithm` class, an abstract class that defines several essential methods, including `train` and `train_epoch`, which are critical for training. The `A2CBase` class inherits from `BaseAlgorithm` and provides many shared functionalities for both continuous and discrete A2C agents. These include methods such as `play_steps` and `play_steps_rnn` for gathering rollout data, and `env_step` and `env_reset` for interacting with the environment. However, functions directly related to training—like `train`, `train_epoch`, `update_epoch`, `prepare_dataset`, `train_actor_critic`, and `calc_gradients`—are left unimplemented at this level. These functions are implemented in `ContinuousA2CBase`, a subclass of `A2CBase`, and further in `A2CAgent`, a subclass of `ContinuousA2CBase`.

The `ContinuousA2CBase` class is responsible for the core logic of agent training, specifically in the methods `train`, `train_epoch`, and `prepare_dataset`. In the `train` function, the environment is reset once before entering the main training loop. This loop consists of three primary stages: (1) calling `update_epoch`, (2) running `train_epoch`, and (3) logging key information, such as episode length, rewards, and losses. The `update_epoch` function, which increments the epoch count, is implemented in `A2CAgent`. The heart of the training process is the `train_epoch` function, which operates as follows:

1. `play_steps` or `play_steps_rnn` is called to generate rollout data in the form of a dictionary of tensors, `batch_dict`. The number of environment steps collected equals the configured `horizon_length`.
2. `prepare_dataset` modifies the tensors in `batch_dict`, which may include normalizing values and advantages, depending on the configuration.
3. Multiple mini-epochs are executed. In each mini-epoch, the dataset is divided into mini-batches, which are sequentially fed into `train_actor_critic`. Function `train_actor_critic`, implemented in `A2CAgent`, internally calls `calc_grad`, also found in `A2CAgent`.

The `A2CAgent` class, which inherits from `ContinuousA2CBase`, handles the crucial task of gradient calculation and model parameter optimization in its `calc_grad` function. Specifically, `calc_grad` first performs a forward pass of the policy model with PyTorch's gradients and computational graph enabled. It then calculates the individual loss terms as well as the total scalar loss, runs the backward pass to compute gradients, truncates gradients if necessary, updates model parameters via the optimizer, and finally logs the relevant training metrics such as loss terms and learning rates.

With an understanding of the default functions, it becomes straightforward to customize agents by inheriting from `A2CAgent` and overriding specific methods to suit particular needs. A good example of this is the implementation of the AMP algorithm [116] in `IsaacGymEnvs`, where the `AMPAgent` class is created and registered in `train.py`, as shown in Listing 3.3.

```python
_runner.algo_factory.register_builder(
    'amp_continuous',
    lambda **kwargs: amp_continuous.AMPAgent(**kwargs)
)
```

Listing 3.3: Registration of `AMPAgent`.

3.3.3. Players

Similar to training algorithms, default players are registered with `player_factory` in the `Runner` class. These include `PpoPlayerContinuous`, `PpoPlayerDiscrete`, and `SACPlayer`. Each of these player classes inherits from the `BasePlayer` class, which provides a common `run` function. The derived player classes implement specific methods for restoring from model checkpoints (`restore`), initializing the RNN (`reset`), and generating actions based on observations through `get_action` and `get_masked_action`.

The testing loop is simpler compared to the training loop. It starts by resetting the environment to obtain the initial observation. Then, for `max_steps` iterations, the loop feeds the observation into the model to generate an action, which is applied to the environment to retrieve the next observation, reward, and other necessary data. This process is repeated for `n_games` episodes, after which the average reward and episode lengths are calculated and displayed.

Customizing the testing loop is as straightforward as customizing the training loop. By inheriting from a default player class, one can override specific functions as needed. As with custom training algorithms, customized players must also be registered with `player_factory` in `train.py`, as demonstrated in Listing 3.4.

```
self.player_factory.register_builder('a2c_continuous',
                                     lambda **kwargs: players.PpoPlayerContinuous(**kwargs))
self.player_factory.register_builder('a2c_discrete',
                                     lambda **kwargs: players.PpoPlayerDiscrete(**kwargs))
self.player_factory.register_builder('sac',
                                     lambda **kwargs: players.SACPlayer(**kwargs))

_runner.player_factory.register_builder(
    'amp_continuous',
    lambda **kwargs: amp_players.AMPPlayerContinuous(**kwargs)
)
```

Listing 3.4: Default players and registration of custom `AMPPlayerContinuous`.

3.3.4. Models And Networks

The terminology and implementation of models and networks in RL Games version `1.6.1` can be confusing for new users. We aim to clarify these concepts and provide a high-level overview of their functionality and relationships.

- **Network Builder.** Network builder classes, such as `A2CBuilder` and `SACBuilder`, are subclasses of `NetworkBuilder` and can be found in `algos_torch.network_builder`. The core component of a network builder is the nested `Network` class (we name it the "inner network" class), which is typically derived from `torch.nn.Module`. This class receives a dictionary of tensors, such as observations and other necessary inputs, and outputs a tuple of tensors from which actions can be generated. The `forward` function of the `Network` class handles this transformation. Additionally, a network builder includes a `load` function to load parameters that define the network architecture and a `build` function to instantiate the `Network` class.
- **Model.** Model classes, like `ModelA2C` and `ModelSACContinuous`, inherit from `BaseModel` in `algos_torch.models`. They are similar to network builders, as each contains a nested `Network` class (referred to as the "model network" class) and a `build` function to construct an instance of this network. To instantiate a model network class, it requires an inner network class, commonly named `a2c_network` or `sac_network`, depending on the algorithm to be used. The model network class, also derived from `torch.nn.Module`, incorporates both the inner network and normalization modules as submodules. Its `forward` function supports different modes for training and playing and uses tensor dictionaries for both the input and output.
- **Model & Network in Algorithm.** In a default agent or player algorithm, `self.model` refers to an instance of the model network class, while `self.network` refers to an instance of the model class.
- **Model Builder.** The `ModelBuilder` class, located in `algos_torch.model_builder`, is responsible for loading and managing models. It is usually instantiated in the `load_networks` function of an agent or player class. Within its constructor, `ModelBuilder` registers the default network builders and models. It also provides a `load` function, which creates a model instance based on the specified name. Note that there is also a `NetworkBuilder` class in the same file, but it is used internally by `ModelBuilder`.

Customizing models requires implementing a custom network builder and a model class. These custom classes should be registered in the `Runner` class within `train.py`. A good reference example is again the AMP implementation. Listing A.4 demonstrates the default registration of models and network builders, along with the AMP example for registering customized components.

3.4. Conclusion

In this chapter, we laid the groundwork for the development of our obstacle-aware drone racing framework. We began by outlining the core methodology, which is using domain randomization to enhance the generalizability of the learned policies. This overview served as a guide for making informed design choices and implementing practical solutions.

Following this, we focused on creating environments suited for reinforcement learning. This section highlighted the necessary components for simulating complex racing tracks and obstacles while ensuring scalability and efficiency. The environment design plays a pivotal role in achieving our objective of training policies that can navigate a variety of tracks with obstacles.

Additionally, we explored the reinforcement learning library, RL Games, in-depth, dissecting key aspects such as algorithm registration, training, and testing loops, as well as models and networks. Understanding the code of the library is essential for efficient customization and implementation of our approach. We also highlighted areas where specific adaptations and custom implementations are required to tailor the defaults to our problem. This section of the report has been merged into the main branch of the RL Games repository as part of its documentation.

In conclusion, this chapter provides a comprehensive introduction to the preliminary steps required to train a generalizable policy for obstacle-aware drone racing. This preliminary work is crucial for anything that builds upon it, as presented in the standalone article.

Part II

Early Results

4
Simulation Performance

4.1. Controller Response

The angular velocity controller was designed with adjustable parameters to allow for domain randomization, which could improve sim-to-real transfer by varying controller performance. However, for this project, we employed a fixed set of hand-tuned parameters throughout training and experimentation to maintain a clear focus on training policies that generalize to unseen tracks and obstacles. This set of parameters was tuned for a controller running at 250 Hz. The resulting angular velocity response is demonstrated below in Figure 4.1 by plotting both the desired and measured angular velocities over simulation time as the drone completes a lap on the Split-S track [34].

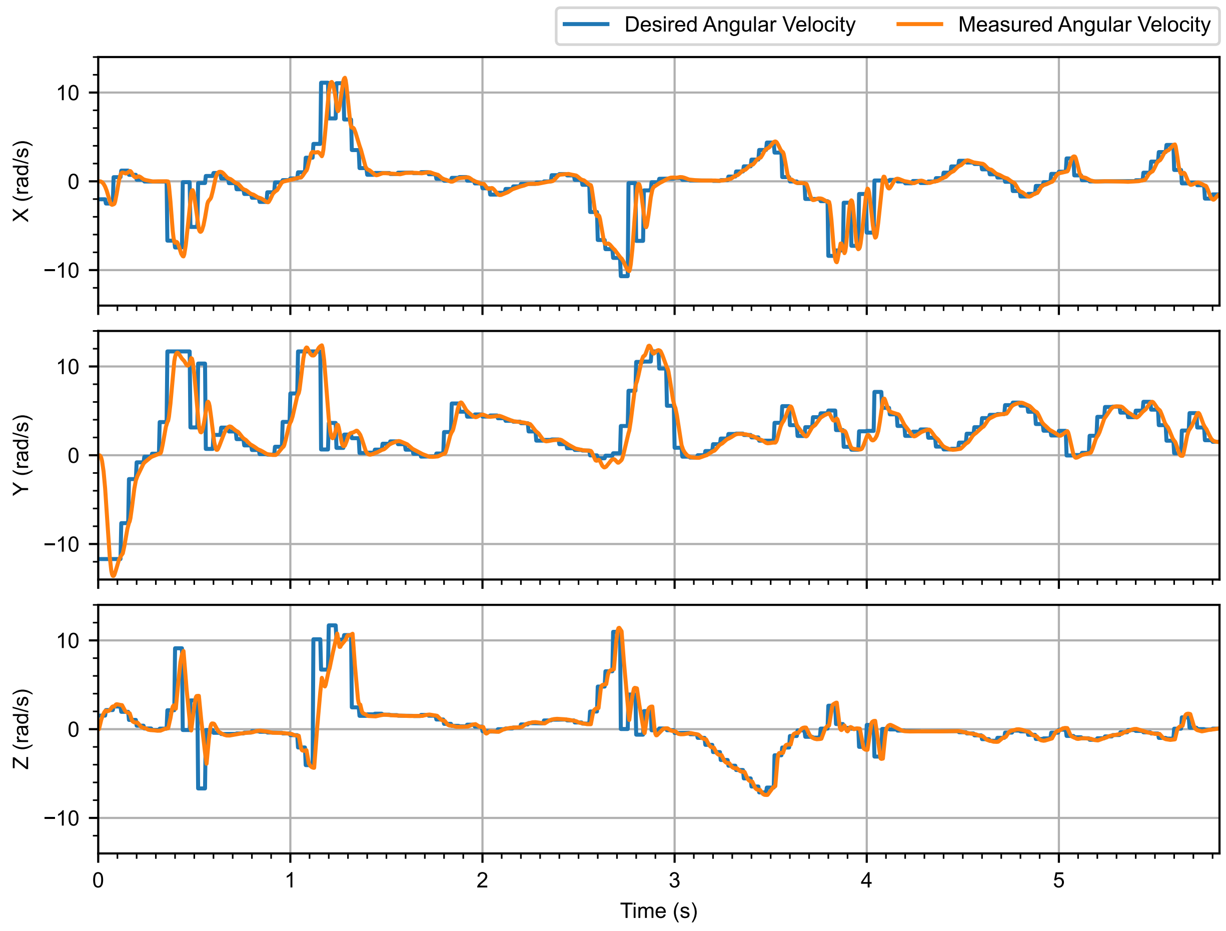


Figure 4.1: Components of desired and measured body-frame angular velocity.

4.2. Environment Steps per Second

The speed at which simulated environments can run is crucial for efficient reinforcement learning. Steps Per Second (SPS) is a key metric that indicates how many environment simulation steps can be processed per second. Higher SPS values enable faster data collection, quicker policy updates, and more extensive training within limited timeframes. Optimizing SPS directly impacts the rate of policy updates and overall training duration. Here we evaluate the performance of our drone racing environments under various configurations.

Our implementation allows multiple environments to be stepped in parallel within a single simulation step. Consequently, the total SPS is the product of the per-environment SPS and the number of environments running in parallel. We conducted experiments to analyze the total SPS, (per-environment) SPS, and GPU VRAM usage (normalized using 24 GB) at different scales of parallelization, scene complexity, and camera sensor presence (480×270). To maximize performance, the experiments were run in headless mode when cameras were disabled, and in GUI mode when cameras were enabled. The results of these experiments are presented in Figures 4.2 to 4.4. All experiments were conducted on a desktop computer equipped with an `Intel i9-13900K` CPU and an `Nvidia RTX 4090` GPU running `Ubuntu 22.04`.

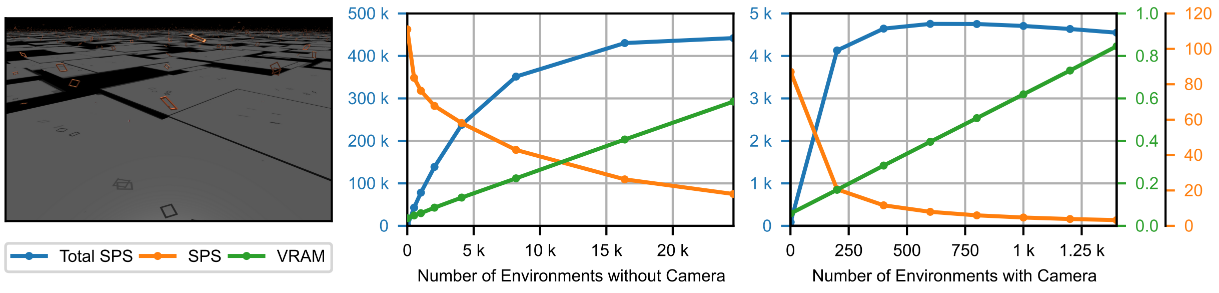


Figure 4.2: SPS and VRAM usage for different numbers of parallel environments and camera configurations, where each environment consists of 1 ground plane and 2 gates.

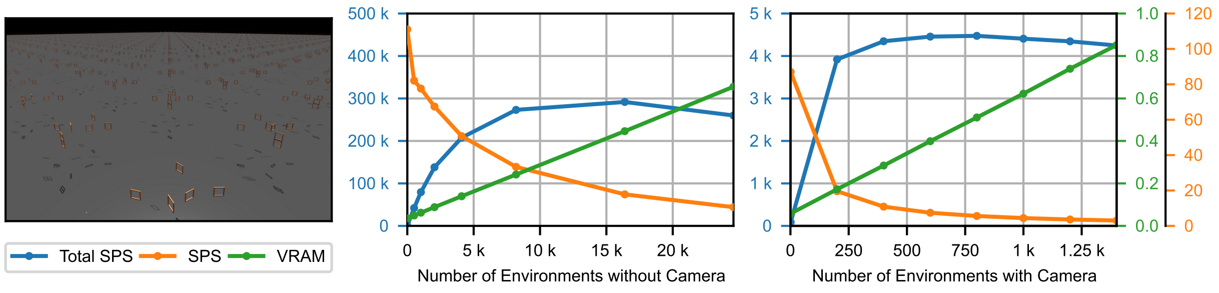


Figure 4.3: SPS and VRAM usage for different numbers of parallel environments and camera configurations, where each environment consists of 1 ground plane and 7 gates (Split-S).

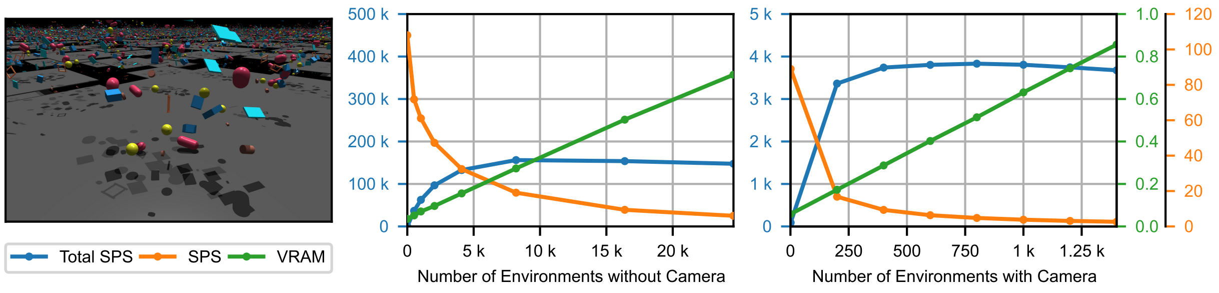


Figure 4.4: SPS and VRAM usage for different numbers of parallel environments and camera configurations, where each environment consists of 1 ground plane, 2 gates, and 20 geometries.

The number of parallel environments is a critical factor in determining overall performance. As expected, increasing the number of environments results in a decrease in per-environment SPS due to the fixed computational capacity being “distributed” across more environments. However, total SPS—calculated as the product of the number of environments and per-environment SPS—initially rises with the increase in parallel environments. This growth continues until reaching a point of diminishing returns, where the system becomes saturated.

The trends of total SPS, per-environment SPS, and VRAM usage vs. the number of environments are consistent for both camera-enabled and camera-disabled environments. However, it is important to note that enabling camera sensors introduces substantial computational overhead and significantly increases VRAM usage, limiting the number of environments that can run in parallel to the thousands. In such cases, the maximum achievable total SPS is reduced by two orders of magnitude compared to environments without cameras.

Scene complexity also plays a crucial role in determining simulation performance. As the complexity of the environment increases—through the addition of gates or geometric obstacles—the computational demands per environment rise significantly. This leads to a noticeable reduction in per-environment and total SPS, as more resources are required to simulate the additional elements.

These experiments demonstrate that factors such as scene complexity, number of parallel environments, and sensor configurations all significantly influence simulation speed. In general, environments without camera sensors can generate hundreds of thousands of interaction data points per second, enabling the rapid learning of state-based policies in a matter of minutes. While enabling cameras introduces substantial computational overhead, resulting in a slower simulation speed, it remains considerably faster than other simulators and frameworks like Flightmare, AirSim, and FlightGoggles.

5 Obstacle-Free Drone Racing

Obstacle-free drone racing eliminates the need for camera sensors typically required for obstacle avoidance, enabling us to fully leverage our environments' capacity to achieve hundreds of thousands of total steps per second, as detailed in the last section. This capability facilitates the rapid learning of state-based policies, often resulting in convergence in under 20 minutes. In this chapter, we will first demonstrate our ability to train directly on the target racing track, reaching state-of-the-art performance in a short time frame. Following this, we will showcase the effectiveness of utilizing random track segments across different parallel environments to develop a generalizable racing policy that can race on unseen tracks.

5.1. Direct Training on the Target Track

We utilize the Split-S track as our target racing track to demonstrate direct training. The track layout and environment performance are illustrated in Figure 4.3. To maximize total SPS, we spawn 16,384 parallel environments during training. Given the asynchronous nature of these environments, we set the rollout horizon in the PPO algorithm to 50 steps, equivalent to 2 seconds of simulation time, to enable frequent policy updates. The policy is represented by a five-layer MLP with hidden dimensions of 256, 128, 128, 64. For the observations, we use a 56-dimensional vector derived from the observation vector described in Part III, by simply removing the DCE vector. And the reward function is the same as that in Part III.

During training, drones are initialized with the same pose as the initial waypoint, with zero commands and velocities. The policy converges within 250 million total steps, or approximately 20 minutes of training, achieving a lap time of 5.72 s. The mean episode reward and length throughout training are presented below.

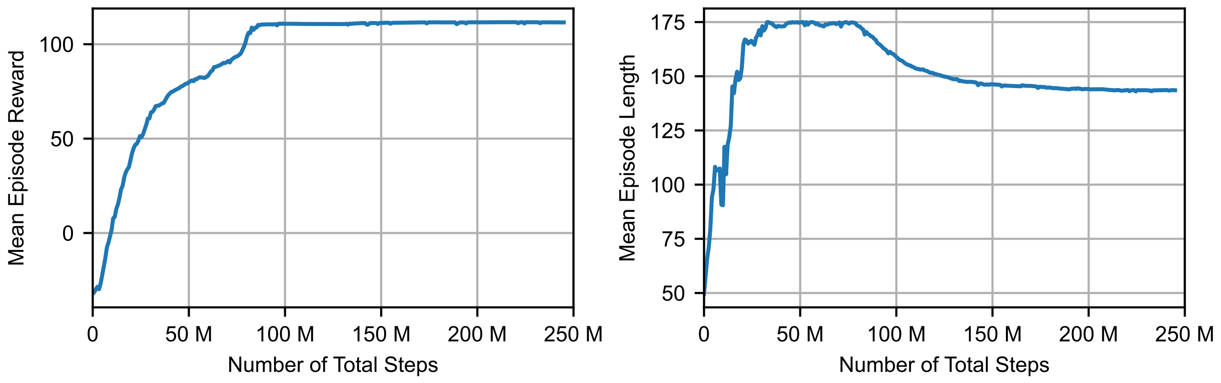


Figure 5.1: Mean episode reward and length throughout direct training on Split-S.

From these plots, we can infer what is happening at each stage of training. Initially, the policy struggles to navigate through gates or steer the drone properly, leading to quick episode failures and negative mean rewards. As training progresses, the policy learns to pass through an increasing number of gates, resulting in both higher rewards and longer episode lengths. Around 30 million steps, the episode length plateaus as timeouts occur after 175 steps. This indicates that while the policy can reliably pass through multiple gates, it is unable to complete the entire track within the timeout limit. However, as training goes on, the policy begins to improve its speed. At approximately 75 million steps, the reward starts to level off, while episode lengths begin to decrease, signaling that the policy can successfully finish the track. Following this point, the policy refines its racing skills to maximize speed and accumulate the highest discounted reward until no further improvements can be made within this particular experimental setup.

We further plot the trajectory produced by rolling out the best-reward policy on two racing tracks: the Split-S track and the Turns track, as shown in Figure 5.2. As illustrated, the policy performs well on the training track, reaching a peak speed of about 25 m/s and generating a typical drone racing trajectory, consistent with findings from other works [25, 32, 34]. However, on the Turns track, which features several straight segments and simple turns, the policy struggles to pass through even the third gate. These observations suggest that the policy trained directly on the target track exhibits strong overfitting to the training environment and fails to generalize effectively to new tracks.

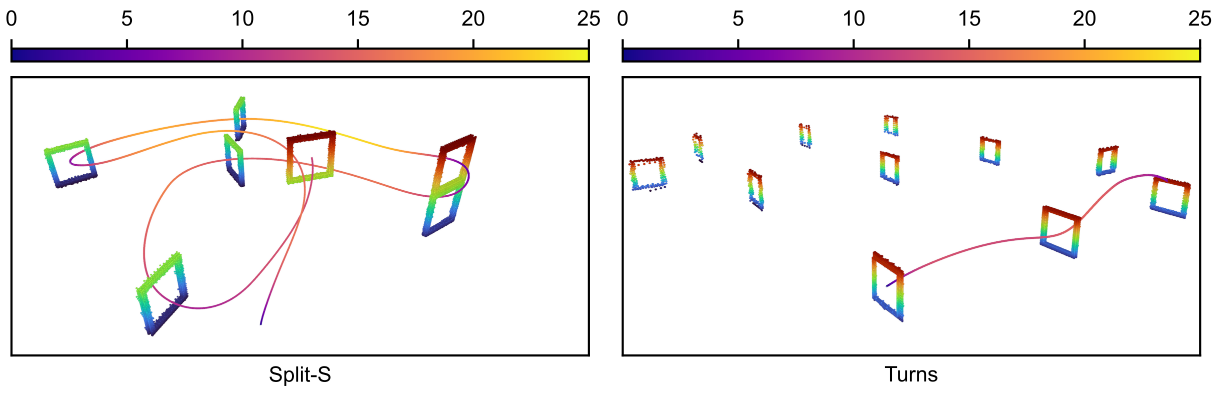


Figure 5.2: Rollout trajectory of the policy trained directly on the Split-S track. The top color bar maps trajectory color to drone speed in m/s. Gates are represented as points colored according to their heights.

5.2. Learning a Generalizable Policy

Achieving a generalizable policy is crucial for enabling agents to perform well across diverse environments rather than just in the specific conditions under which they were trained. And it is one of the biggest challenges in RL. In the context of drone racing, policies trained only on a single track may exhibit overfitting, as demonstrated in the previous section. In this section, we explore the use of domain randomization and varied training scenarios. By using short random track segments in parallel environments as shown in Figure 4.2, we aim to train the policy to adapt to unseen tracks. Figure 5.3 illustrates the mean episode reward and episode length recorded during training in this randomized setup.

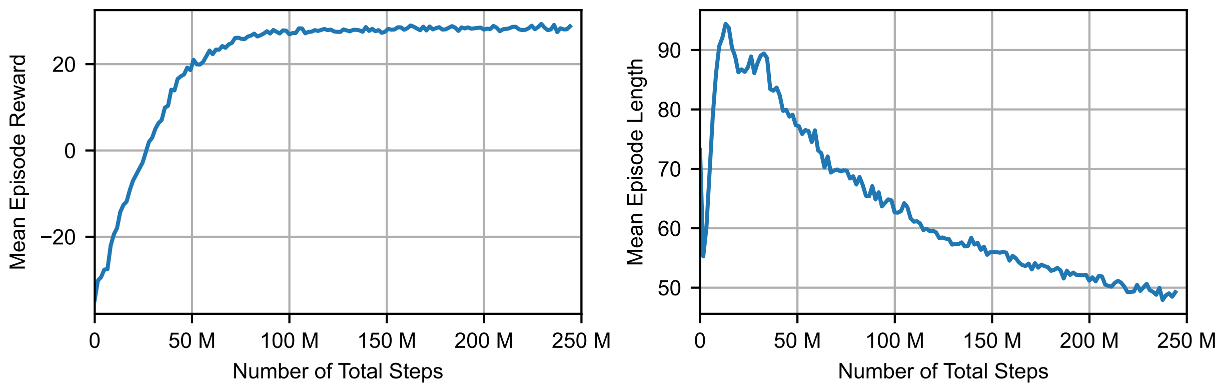


Figure 5.3: Mean episode reward and length throughout training on random different tracks.

The policy network, observation, and reward settings remain the same as in the direct training experiment. However, the tracks are randomized across environments using the waypoint generator, as described in Section 3.2, and drones are initialized with random bounded commands, velocities, and poses. Given the diversity of environments, we increased the rollout horizon to 100 steps to ensure that each agent has enough time to reach the finishing gate in a single rollout. Additionally, the simplicity of these randomized tracks allows us to achieve a higher total SPS, enabling the collection of 250 million steps in just 10 minutes. Within this time frame, the learned policy demonstrates a notable ability to generalize to unseen racing tracks.

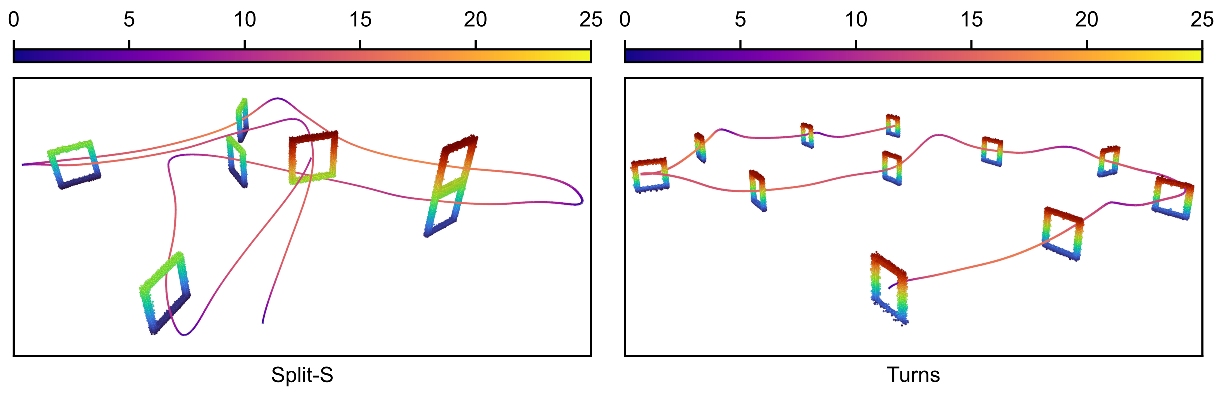


Figure 5.4: Rollout trajectory of the policy trained on random tracks. The top color bar maps trajectory color to drone speed in m/s. Gates are represented as points colored according to their heights.

Figure 5.4 shows the resulting trajectory of the generalizable policy on both the Split-S and Turns tracks. While the policy successfully completes both tracks, the trajectories are not as optimal as those achieved through direct training. This can be observed by comparing the Split-S results in Figures 5.4 and 5.2. Despite the sub-optimal performance, this policy can serve as a starting point for adaptation to specific target tracks, potentially speeding up the process of training a fully optimized policy. However, this performance gap raises an important question: how can we train a policy that not only generalizes to unseen tracks but also performs as well as a directly trained policy in terms of smoothness and lap time? We believe this is an interesting direction for future research.

Part III

Scientific Article

Learning Generalizable Policy for Obstacle-Aware Autonomous Drone Racing

Yueqian Liu

***Abstract*—Autonomous drone racing has gained attention for its potential to push the boundaries of drone navigation technologies. While much of the existing research focuses on racing in obstacle-free environments, few studies have addressed the complexities of obstacle-aware racing, and approaches presented in these studies often suffer from overfitting, with learned policies generalizing poorly to new environments. This work addresses the challenge of developing a generalizable obstacle-aware drone racing policy using deep reinforcement learning. We propose applying domain randomization on racing tracks and obstacle configurations before every rollout, combined with parallel experience collection in randomized environments to achieve the goal. The proposed randomization strategy is shown to be effective through simulated experiments where drones reach speeds of up to 70 km/h, racing in unseen cluttered environments. This study serves as a stepping stone toward learning robust policies for obstacle-aware drone racing and general-purpose drone navigation in cluttered environments. Code is available at** `https://github.com/ErcBunny/IsaacGymEnvs`.

***Index Terms*—Aerial Systems: Perception and Autonomy, Collision Avoidance, Integrated Planning and Learning, Reinforcement Learning**

I. Introduction

AUTONOMOUS drone navigation has emerged as a critical area of research, driven by the growing demand for drones in industries such as delivery, inspection, and emergency response. Drone racing, with its emphasis on minimum-time navigation, has become a benchmark task for testing advanced autonomous systems aiming to navigate at high speeds while avoiding obstacles in partially or fully unknown environments. Drone racing originally began as a competitive sport where human pilots control agile drones via radio to fly through a racing track as fast as possible while avoiding potentially present obstacles. This requires precision, quick reflexes, and expert navigation skills. In autonomous drone racing, human pilots are replaced by algorithms and artificial intelligence (AI). This introduces the challenge for algorithms and AI of matching human-level performance and adaptability.

There have been several global autonomous drone racing events, including the 2016-2019 IROS Autonomous Drone Racing (ADR) competitions [1], [2], the 2019 AlphaPilot Challenge [3], [4], the 2019 NeurIPS Game of Drones [5], and the 2022-2023 DJI RMUA UAV Challenges [6], [7]. The tracks in early competitions, such as the IROS ADR, AlphaPilot, and Game of Drones, are situated in less cluttered spaces, allowing drones to complete the tracks without considering obstacle avoidance. However, for tracks in cluttered environments, such as those in the more recent DJI RMUA Challenges, the absence of obstacle awareness could cause crashing. Additionally, in human-piloted drone racing, such as the Drone Racing League competitions, and in drone racing video games, there are plenty of tracks that require obstacle avoidance.

This paper is part of the author's M.Sc. thesis supervised by Ir. Hang Yu and Dr. Ir. Christophe De Wagter, at MAVLab TU Delft. Full thesis is available at `https://repository.tudelft.nl`.

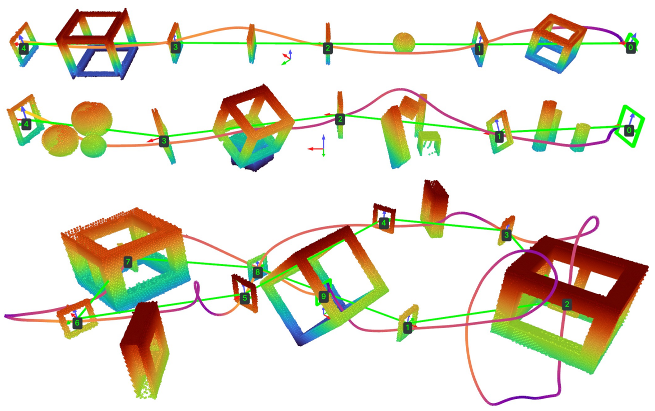


Fig. 1. Trajectories of successful rollouts of a single policy on multiple different racing tracks with obstacles designed to block flight paths.

Although autonomous drone racing has received significant attention, much of the research has been limited to obstacle-free scenarios [8]. Obstacle-free scenarios do not reflect the complexities encountered in real-world tasks where obstacle avoidance is necessary. Recognizing this, researchers have been exploring ways to integrate drone racing with obstacle avoidance through various approaches. Path-planning and optimization-based approaches can achieve short lap times [9] and strike a balance between lap times and computational efficiency [6], [10], [11], but rely on carefully designed algorithms and may experience performance degradation when model mismatches occur. Current learning-based approaches [12], [13] leverage reinforcement learning (RL) and imitation learning (IL) to train neural policies capable of making low-latency decisions that result in aggressive and collision-free trajectories. However, these policies do not generalize well to new racing tracks or different obstacle configurations.

This paper aims to enhance the generalization ability of learned policies. Specifically, the goal is to develop a single policy capable of navigating a quadcopter through various racing tracks with obstacles, without requiring additional tuning after training. Drawing inspiration from works on learning drone navigation in cluttered environments [14], [15] and generalizable obstacle-free drone racing [16], which all involve training the policy in multiple randomized environments, we propose applying the same strategy, domain randomization [17] over racing tracks, to expose the agent to a diverse set of environments. This allows the policy to learn the underlying navigation "skills" while not relying on unique observations associated with one or a few training environments. Simulated experiments verify that the resulting policy can indeed generalize to unseen racing tracks while avoiding obstacles in unseen sizes and shapes. Several successful examples are shown in Fig 1. In summary, the main contributions of this study are:

- We verify the effectiveness of applying domain randomization to encourage the learning of generalizable skills.
- We present the first generalizable neural policy for the obstacle-aware drone racing task, where the policy directly maps observations to low-level commands.
- We open-source tools and reusable modules to facilitate research and development in both obstacle-free and obstacle-aware drone racing.

II. Related Work

A. *Obstacle-Free Autonomous Drone Racing*

Optimization-based methods have been widely applied to the task of drone racing. For static obstacle-free racing tracks, time-optimal trajectories passing through gates' center waypoints can be generated using optimization with Complementary Progress Constraint (CPC) [18]. However, this approach is computationally expensive and struggles to adapt to changing track layouts. To reduce computational overhead, Model Predictive Contouring Control (MPCC) is introduced [19]. MPCC has been further extended to include an online reference path generation module to adapt to dynamic tracks and handle external disturbances [20]. A more recent study [21] demonstrates that lap times can be further reduced by exploiting the spatial potential of the gates.

Reinforcement Learning has also emerged as a promising approach for autonomous drone racing. Near-time-optimal agile flight can be achieved through state-based RL [16]. Although the learned policy results in slightly longer lap times than the CPC method, it handles variations in gate poses and generalizes to unseen tracks. Furthermore, state-based policies can serve as teacher policies within the IL framework, enabling the training of purely vision-based student policies [22]. In a follow-up study [23], the student policy is further fine-tuned using RL. These two studies show the feasibility of high-speed agile flight using AI with the same input-output modalities as human pilots.

Reinforcement learning based approaches offer several advantages over optimization-based methods. These include improved lap times and higher success rates during real-world flights, where unmodeled effects and disturbances are non-negligible [24]. However, deploying policies in real-world scenarios is challenging, requiring closing the sim-to-real gap and careful system engineering. The Swift system [25] demonstrates that, by bridging the gap via fine-tuning using data-driven residual models, AI systems powered by RL policies can achieve performance on par with human champions.

B. *Obstacle-Aware Autonomous Drone Racing*

For the task of obstacle-Aware autonomous drone racing, leveraging optimization and planning, several methods have proven effective. The teach-and-repeat framework is widely used in autonomous robot missions, and has been applied to drone racing [10]. This framework enables the drone to fly through the track while avoiding previously unseen and dynamic obstacles. In static environments, Fast-Racing [11] provides a polynomial trajectory baseline for obstacle-aware autonomous drone racing. The winning solution of the 2022 DJI RMUA UAV Challenge [6] also follows a polynomial-based trajectory but incorporates an additional online re-planning module to avoid dynamic obstacles and pass through moving gates. Moreover, Penick et al. [9] offer a sampling-based baseline aimed at finding time-optimal trajectories in cluttered environments, though this method struggles to scale with increasing environment complexity.

While RL-based methods have shown great promise in obstacle-free autonomous drone racing, achieving better lap times, disturbance rejection, and less compute latency, their application to obstacle-aware racing remains relatively sparse. To the best of our knowledge, only two studies [12], [13] have addressed this challenging task. Furthermore, they all focus on completing a single predefined racing track in minimum time, not considering generalizing to unseen scenarios. Realizing this gap, we aim to explore methods that enhance policy generalization in obstacle-aware drone racing.

C. *Vision-Involved Navigation via Deep RL*

In the aforementioned works addressing obstacle-aware drone racing with RL, the policies fail to generalize to environments different from the ones they were trained on. To overcome this limitation, research on learning general-purpose navigation offers valuable insights and guidance.

Near-perfect discrete-action indoor navigation for ground robots has been demonstrated with DD-PPO [26], in which training occurs across multiple indoor scenes to enhance generalization. The agent utilizes a policy network comprising a Convolutional Neural Network (CNN) as the encoder and a Long Short-Term Memory (LSTM) network. At first, this network is optimized as a whole using RL, which is inefficient. Follow-up works [27], [28] show that using auxiliary tasks, such as predicting depth, inverse dynamics, and remaining distance to target, results in quicker policy convergence and better overall performance.

Besides using auxiliary tasks, modular learning is another approach to achieving efficient learning for vision-involved navigation tasks. Hoeller et al. [29] propose a modular learning framework for training a quadruped robot to navigate in cluttered dynamic environments. Here network modules are learned separately: once an upstream module is learned, it is frozen while downstream modules are optimized. MAVRL [14] and Kulkarni et al. [15] also adopt a similar framework for drone navigation in clutter. All these methods utilize randomization of the training environments to promote generalization, which directly inspires our core idea for learning a generalizable policy in obstacle-aware drone racing.

III. Methodology

A. Drone Model and Waypoints

The drone is modeled as a rigid body with mass m and moment of inertia $\boldsymbol{J}$. With the body frame attached to the center of gravity, the equations of dynamics can be written as:

$$\begin{bmatrix} \dot{\boldsymbol{p}}_{\mathcal{W}} \\ \dot{\boldsymbol{q}}_{\mathcal{WB}} \\ \dot{\boldsymbol{v}}_{\mathcal{W}} \\ \dot{\boldsymbol{\omega}}_{\mathcal{B}} \\ \dot{\boldsymbol{\Omega}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_{\mathcal{W}} \\ \frac{1}{2}\boldsymbol{q}_{\mathcal{WB}} \otimes \begin{bmatrix} 0 & \boldsymbol{\omega}_{\mathcal{B}}^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}} \\ \boldsymbol{g}_{\mathcal{W}} + \frac{1}{m}\boldsymbol{R}_{\mathcal{WB}}(\boldsymbol{f}_a + \boldsymbol{f}_d) \\ \boldsymbol{J}^{-1}(\boldsymbol{\tau}_a + \boldsymbol{\tau}_d - \boldsymbol{\omega}_{\mathcal{B}} \times \boldsymbol{J}\boldsymbol{\omega}_{\mathcal{B}}) \\ k_r(\boldsymbol{\Omega}_s - \boldsymbol{\Omega}) \end{bmatrix}, \tag{1}$$

where $\mathcal{W}$, $\mathcal{B}$ denote the world frame and the drone body frame, and $\boldsymbol{p}_{\mathcal{W}}$, $\boldsymbol{q}_{\mathcal{WB}}$, $\boldsymbol{v}_{\mathcal{W}}$, $\boldsymbol{\omega}_{\mathcal{B}}$, $\boldsymbol{g}_{\mathcal{W}}$ represent drone position, attitude quaternion, linear velocity, angular velocity, and gravitational acceleration, respectively. Rotation matrix of drone attitude is $\boldsymbol{R}_{\mathcal{WB}}$. Actuator wrench, or "force and torque", $(\boldsymbol{f}_a, \boldsymbol{\tau}_a)$ and aerodynamic drag wrench $(\boldsymbol{f}_d, \boldsymbol{\tau}_d)$ make up the total wrench acting on the rigid body. Rotor spinning dynamics is considered as a first-order lag model: the derivative of rotor angular velocities in rounds per second (RPM) $\dot{\boldsymbol{\Omega}}$ is the product of the rotor constant k_r and the difference between steady-state RPM $\boldsymbol{\Omega}_s$ and current RPM $\boldsymbol{\Omega}$.

From rotor angular velocities and accelerations we can calculate the actuator wrenches:

$$\begin{bmatrix} \boldsymbol{f}_a \\ \boldsymbol{\tau}_a \end{bmatrix} = \begin{bmatrix} \sum_i \boldsymbol{f}_p(\boldsymbol{\Omega}_i) \\ \sum_i \left(\boldsymbol{\tau}_p(\boldsymbol{\Omega}_i) + \boldsymbol{r}_i \times \boldsymbol{f}_p(\boldsymbol{\Omega}_i) + \zeta_i \boldsymbol{J}_r \dot{\boldsymbol{\Omega}}_i\right) \end{bmatrix}, \tag{2}$$

where $\boldsymbol{f}_p(\boldsymbol{\Omega}_i)$ and $\boldsymbol{\tau}_p(\boldsymbol{\Omega}_i)$ represent the thrust force and torque generated by the i-th spinning propeller, respectively. They are calculated using polynomial models derived from the motor manufacturer's data. Here, $\boldsymbol{r}_i$ is the displacement of the i-th rotor relative to the drone's body frame origin, $\boldsymbol{J}_r$ denotes the moment of inertia of the rotor, which includes the propeller and the spinning parts of the motor, and ζ_i indicates the rotational direction of the i-th rotor.

The aerodynamic drag depends on the linear velocity, $\boldsymbol{v}_{\mathcal{B}} = \boldsymbol{R}_{\mathcal{WB}}^{-1}\boldsymbol{v}_{\mathcal{W}}$, and the angular velocity, $\boldsymbol{\omega}_{\mathcal{B}}$, in the body frame:

$$\begin{bmatrix} \boldsymbol{f}_d \\ \boldsymbol{\tau}_d \end{bmatrix} = -\frac{1}{2}\rho \begin{bmatrix} A_t \left(\boldsymbol{C}_0 \boldsymbol{v}_{\mathcal{B}}^2 + \boldsymbol{C}_1 \boldsymbol{v}_{\mathcal{B}}\right) \\ A_r \left(\boldsymbol{C}_2 \boldsymbol{\omega}_{\mathcal{B}}^2 + \boldsymbol{C}_3 \boldsymbol{\omega}_{\mathcal{B}}\right) \end{bmatrix}, \tag{3}$$

where ρ is the air density, A_t and A_r are the effective areas responsible for generating translational and rotational drag, respectively. Coefficients $\boldsymbol{C}_i$ for i from 0 to 3 are adjustable coefficients in the polynomial drag model.

The steady-state angular velocity is determined by the motor commands $\boldsymbol{u}_m$ using a polynomial model fitted to the motor manufacturer's data. Simulating how human operators control racing drones, we employ an angular velocity controller to translate control commands $\boldsymbol{a} \in [-1, 1]^4$ to motor commands $\boldsymbol{u}_m \in [0, 1]$ of the drone. Vector $\boldsymbol{a}$ contains 3 channels for body rates and 1 channel for the throttle, or collective thrust level. The angular velocity controller is derived from Betaflight [30] and is responsible for mapping control commands to desired angular velocity, running closed-loop control, and allocating the control to motor commands.

Apart from dynamics and control, the rigid body's collision geometry is coarsely approximated by the minimum body-frame axis-aligned bounding box of the real geometry. Regarding sensors, we assume we have access to accurate vehicle states including $\boldsymbol{p}_{\mathcal{W}}$, $\boldsymbol{q}_{\mathcal{WB}}$, $\boldsymbol{v}_{\mathcal{W}}$, and $\boldsymbol{\omega}_{\mathcal{B}}$. Furthermore, we attach a tilted depth camera to the front of the drone and set the depth sensing range to a relatively large value (e.g. 20 m) to mimic how human operators or depth estimation networks retrieve depth from monocular images.

Drone racing requires the drone to pass through gates from the correct side to the other. In the obstacle-aware scenario, there are possible additional requirements for the drone to avoid certain obstacles by following desired courses where no physical gates exist. We propose to model physical gates and course constraints as waypoints. A waypoint is defined as a finite-size rectangle plane parameterized by position $\boldsymbol{p}_{\text{wp}}$, attitude quaternion $\boldsymbol{q}_{\text{wp}}$, width and height of the valid pass-through region $(w_{\text{wp}}, h_{\text{wp}})$, and a binary parameter g_{wp}, which indicates the presence of physical bars around the waypoint. The waypoint frame, denoted by $\mathcal{G}$, is attached to the center of the rectangle, with the x-axis perpendicular to the plane, pointing towards the valid pass-through direction, and y-axis z-axis perpendicular to the sides of the rectangle. An illustration of waypoints can be found in Figure 3(a).

B. Task Formulation

We formulate obstacle-aware autonomous drone racing as a partially observable Markov decision process (POMDP) [31]. POMDP models the agent-environment interaction and internal dynamics of the environment. The agent is an AI system that decides on the action to take based on observations from the environment. The environment is everything else that takes the agent's actions, updates environment states, computes rewards, and finally outputs the observations, closing the interaction loop. Actions are taken based on the policy π that maps observations to actions, with the transition model of states, a trajectory $\boldsymbol{\tau}$ can be produced. The goal is to find the policy that maximizes the expectation of total discounted reward:

$$\max_{\pi} \mathbb{E}_{\boldsymbol{\tau}} \left[\sum_{t=0}^{\infty} \gamma^t r(t)\right], \tag{4}$$

where $\gamma \in [0, 1)$ stands for the discount factor, and $r(t)$ denotes the reward as a function of time t.

1) States: States include every piece of necessary information to define the environment configuration. This may include drone rigid body states, camera transform, internal states of the actuator model and controllers, gate poses and sizes, obstacle shapes, and poses, etc.

2) Action: As discussed in the previous drone model section, we use the control commands denoted by $\boldsymbol{a}$ as the action to simulate human operators' control commands to a radio-controlled racing drone.

3) Transition: The transition of states of our environment is deterministic and only updates states associated with the drone model. Action $\boldsymbol{a}$ is turned into actuator wrenches through the angular velocity controller and Equation (2). Together with the

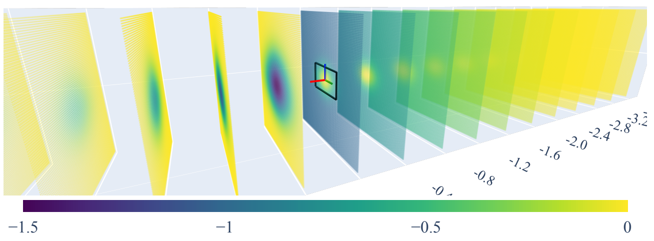


Fig. 2. Illustration of part of the target waypoint's guidance reward field. The pass-through region (outlined using black lines) and the waypoint frame (RGB-xyz) are displayed at the center. The field spans to the entire $\mathbb{R}^3$.

drag wrenches in Equation (3), drone states $\boldsymbol{p}_{\mathcal{W}}$, $\boldsymbol{q}_{\mathcal{WB}}$, $\boldsymbol{v}_{\mathcal{W}}$, and $\boldsymbol{\omega}_{\mathcal{B}}$ can be updated using Equation (1) with an integrator or a physics engine. States of obstacles and gates are fixed within an episode of the process.

4) Reward Function: The reward function is a weighted sum of reward terms including the progress reward r_{prog}, perception reward r_{prec}, command reward r_{cmd}, collision reward r_{col}, guidance reward r_{guid}, waypoint passing reward r_{wp}, timeout reward r_{time}, and finally the linear velocity reward r_{vel}. By representing weights collectively as a vector $\boldsymbol{\lambda}$, we can write the reward as:

$$r = \begin{bmatrix} r_{\text{prog}} & r_{\text{prec}} & r_{\text{cmd}} & r_{\text{col}} & r_{\text{guid}} & r_{\text{wp}} & r_{\text{time}} & r_{\text{vel}} \end{bmatrix} \boldsymbol{\lambda}. \tag{5}$$

Terms r_{prog}, r_{prec}, r_{cmd}, and r_{col} are formulated as in Swift [25]. The guidance reward is extended based on the safety reward seen in [16]. The remaining ones are additional terms proposed in this study.

To calculate the guidance reward, we first need to transform the drone position from the world frame to the target waypoint frame. Let $\boldsymbol{p}_{\mathcal{G}} = [x\ y\ z]^{\mathsf{T}}$ denote drone position in the waypoint frame, the guidance reward is $r_{\text{guid}} = -f^2(x) \cdot g(x, y, z)$, with $f(x) = \max(1 - \text{sgn}(x)x/k_0, 0)$ and $g(x, y, z)$ expanded to:

$$g(x, y, z) = \begin{cases} k_1 \exp(-\frac{y^2+z^2}{2v}), & x > 0 \\ 1 - \exp(-\frac{y^2+z^2}{2v}), & x \leq 0 \end{cases}, \tag{6}$$

where v is further expanded to:

$$v = k_2 \left(1 + f^2(x)\right) \sqrt{\frac{z^2 + y^2}{\left(z/h_{\text{wp}}\right)^2 + \left(y/w_{\text{wp}}\right)^2}}, \tag{7}$$

if $y^2 + z^2 \neq 0$. Otherwise $v = k_2 \left(1 + f^2(x)\right)$. Here k_i for i from 0 to 2 are scalar parameters, and k_2 is different for cases $x > 0$ and $x \leq 0$, i.e. for different sides of the waypoint. Figure 2 illustrates the guidance reward field induced by the target waypoint. Our formulation adapts the original "safety reward" to rectangular waypoints and additionally penalizes the behavior of approaching the gate from the wrong side.

The waypoint passing reward r_{wp} and the timeout reward r_{time} are sparse and only become non-zero at specific steps: r_{wp} turns to positive if the drone has just passed through a waypoint, and r_{time} turns to negative if the drone has not crossed the final waypoint within a time limit.

Finally, we use the linear velocity reward r_{vel} to encourage forward flight, which is beneficial for both making progress and obstacle avoidance with a limited camera field of view. It penalizes lateral and backwards velocity in the body frame $\boldsymbol{v}_{\mathcal{B}} = [v_x\ v_y\ v_z]^{\mathsf{T}}$ using negative parameters k_3 and k_4:

$$r_{\text{vel}} = k_3 v_y^2 + k_4 \left(\min(v_x, 0)\right)^2. \tag{8}$$

5) Observations: We assume all observations are noise-free and deterministic. At time t, the observations include: the depth image $\boldsymbol{d}_t \in [0, 1]^{270\times480}$, drone states $\boldsymbol{s}_t \in [-1, 1]^{18}$, the last action $\boldsymbol{a}_{t-1} \in [-1, 1]^4$, and waypoint information of the next two target waypoints $\boldsymbol{w}_t \in [-1, 1]^{34}$.

The depth image is produced by a depth camera using the pinhole model. We set resolutions to 270×480 and its horizontal FOV to 90 degrees. The transform from the ground-truth depth $\boldsymbol{d}_{\text{gt}}$ to the observed depth $\boldsymbol{d}$ is:

$$\boldsymbol{d} = \min\left(\boldsymbol{d}_{\text{gt}}/d_{\max}, \boldsymbol{1}\right), \tag{9}$$

where $d_{\max}$ denotes the maximum sensing range.

The drone states vector is defined as:

$$\boldsymbol{s} = \begin{bmatrix} \frac{(\boldsymbol{p}_{\mathcal{W}} - \boldsymbol{p}_{\mathcal{W}_0})^{\mathsf{T}}}{p_{\max}} & \boldsymbol{x}_{\mathcal{B}}^{\mathsf{T}} & \boldsymbol{y}_{\mathcal{B}}^{\mathsf{T}} & \boldsymbol{z}_{\mathcal{B}}^{\mathsf{T}} & \frac{\boldsymbol{v}_{\mathcal{W}}^{\mathsf{T}}}{v_{\max}} & \frac{\boldsymbol{\omega}_{\mathcal{B}}^{\mathsf{T}}}{\omega_{\max}} \end{bmatrix}^{\mathsf{T}}, \tag{10}$$

where $\boldsymbol{p}_{\mathcal{W}_0}$ is the initial drone position; $\boldsymbol{x}_{\mathcal{B}}$, $\boldsymbol{y}_{\mathcal{B}}$, and $\boldsymbol{z}_{\mathcal{B}}$ are column vectors of the rotation matrix $\boldsymbol{R}_{\mathcal{WB}}$. Manually adjustable parameters for maximum sensing ranges for the position, linear velocity, and angular velocity are denoted by $p_{\max}$, $v_{\max}$, and $\omega_{\max}$, respectively. This vector is further clamped to $[-1, 1]$ before returned.

We include information about two future waypoints based on the result of the gate observation experiment in [16]: including information about two future gates can improve success rate and lap times. The information vector of one waypoint, indexed i, is defined as:

$$\boldsymbol{w}_t^i = \begin{bmatrix} s_c & \min(\boldsymbol{l}^{\mathsf{T}}/l_{\max}, \boldsymbol{1}) & \boldsymbol{v}_{\text{corners}}^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}}, \tag{11}$$

with s_c being the cosine similarity between vector $\boldsymbol{p}_{\text{wp}_i} - \boldsymbol{p}_{\mathcal{WB}}$ and the x-axis of waypoint i, $\boldsymbol{l}$ being the vector containing lengths of vectors from the origin of the drone body frame to 4 waypoint corners, $l_{\max}$ being the maximum allowed value of these lengths, and $\boldsymbol{v}_{\text{corners}}$ denoting concatenated unit vectors from the drone to the corners. The dimension of $\boldsymbol{w}_t^i$ adds up to 17, so the dimension of $\boldsymbol{w}_t$, containing $\boldsymbol{w}_t^0$ and $\boldsymbol{w}_t^1$, is 34.

6) Policy: We use a neural network to represent the policy. Denoting the parameters of the neural network by $\boldsymbol{\theta}$, we can express the policy function as:

$$\boldsymbol{a}_t = \pi_{\boldsymbol{\theta}}(\boldsymbol{d}_t, \boldsymbol{s}_t, \boldsymbol{w}_t, \boldsymbol{a}_{t-1}). \tag{12}$$

The neural network consists of an image encoder module that encodes an image $\boldsymbol{d}_t$ to a 64-dimensional latent vector $\boldsymbol{z}_t$ and a multi-layer perceptron (MLP) module that maps the concatenated 120-dimensional vector $[\boldsymbol{z}_t^{\mathsf{T}}\ \boldsymbol{s}_t^{\mathsf{T}}\ \boldsymbol{w}_t^{\mathsf{T}}\ \boldsymbol{a}_{t-1}^{\mathsf{T}}]^{\mathsf{T}}$ to action $\boldsymbol{a}_t$.

We employ a pre-trained Deep Collision Encoder (DCE) [32] as the image encoder and freeze its weights during

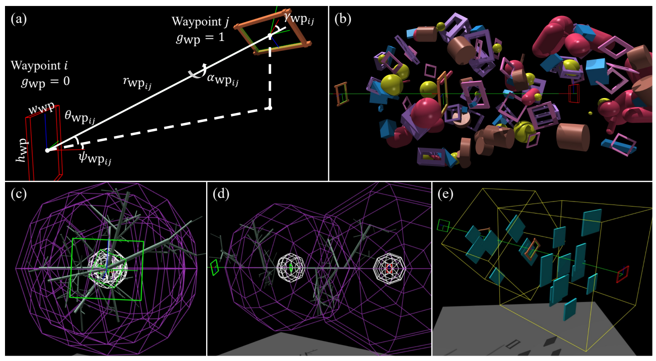


Fig. 3. Illustration of parameters describing relative waypoint poses (a) and obstacles managed by the obstacle manager (b)-(e). Sub-figure (b) shows orbital obstacles, (c) and (d) show tree-like obstacles from different views, and (e) shows wall-like obstacles between waypoints.

training. The DCE is a residual network using convolutional layers at each block, and has fully connected layers that generate the mean and variance of the latent distribution, from which the latent vector $\boldsymbol{z}$ is sampled. This sampling process makes the policy stochastic.

The MLP module consists of two major parts. The first part is composed of 4 fully connected linear layers with Exponential Linear Unit (ELU) activation in between each layer, while the second part contains separate linear layers. The output of the first part is mapped to the mean and variance of the action distribution, in addition to a scalar, also known as the "value", by 3 separate linear layers. During training, the action is sampled from the distribution characterized by mean and variance, and during evaluation and deployment, the action directly uses the mean.

C. Policy Training

Given that our policy is represented by a neural network, our goal boils down to finding the optimal parameters $\boldsymbol{\theta}$ that maximizes the expectation of accumulated rewards in Equation (4). We use the Proximal Policy Optimization (PPO) [33] reinforcement learning algorithm for this purpose. Additionally, we explore the technique of domain randomization to enable generalization to unseen environments, hoping that with enough variability encountered during training, an unseen environment would appear as just another variation, where the policy has required knowledge to finish the track. To achieve this, we have designed a waypoint generator, a random obstacle manager, and multiple training strategies.

1) Waypoint Generator and Obstacle Manager: We consider a racing track the combination of waypoints and obstacles for the obstacle-aware drone racing task. We use the waypoint generator to generate random waypoints and use the obstacle manager to put obstacles at places that effectively block the flight paths between waypoints.

We use 5 values $(\psi_{\mathrm{wp}_{ij}}, \theta_{\mathrm{wp}_{ij}}, r_{\mathrm{wp}_{ij}}, \alpha_{\mathrm{wp}_{ij}}, \gamma_{\mathrm{wp}_{ij}})$ to parameterize relative pose between waypoint i and j, as shown in Figure 3(a). Given the pose of waypoint i as position vector and rotation matrix $(\boldsymbol{p}_{\mathrm{wp}_i}, \boldsymbol{R}_{\mathrm{wp}_i})$, the pose of waypoint j can be calculated using:

$$\begin{aligned} \boldsymbol{p}_{\mathrm{wp}_j} &= r_{\mathrm{wp}_{ij}} \boldsymbol{R}_y(\theta_{\mathrm{wp}_{ij}}) \boldsymbol{R}_z(\psi_{\mathrm{wp}_{ij}}) \boldsymbol{R}_{\mathrm{wp}_i} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^{\mathsf{T}} + \boldsymbol{p}_{\mathrm{wp}_i} \\ \boldsymbol{R}_{\mathrm{wp}_j} &= \boldsymbol{R}_y(\gamma_{\mathrm{wp}_{ij}}) \boldsymbol{R}_x(\alpha_{\mathrm{wp}_{ij}}) \boldsymbol{R}_y(\theta_{\mathrm{wp}_{ij}}) \boldsymbol{R}_z(\psi_{\mathrm{wp}_{ij}}) \boldsymbol{R}_{\mathrm{wp}_i} \end{aligned} . \tag{13}$$

Although there are only 5 degrees of freedom, this parameterization allows for enough room for randomization and intuitive adjustments of the track's difficulties.

Waypoints are generated procedurally. Firstly, the Initial waypoint's roll, pitch, and yaw angles are sampled uniformly within defined bounds, and the position is set to an arbitrary value. Secondly, for subsequent waypoints, we sample the relative pose parameters uniformly within defined bounds and calculate their poses till the final waypoint using Equation (13). Thirdly, parameters $(w_{\mathrm{wp}}, h_{\mathrm{wp}}, g_{\mathrm{wp}})$ are also sampled uniformly within defined ranges for all waypoints. Lastly, we offset all waypoints' positions to fit the track within environment boundaries.

We observe that uniformly distributing obstacles in $\mathbb{R}^3$, as seen in [15], is not suitable for significantly larger environments. Uniformly distributing obstacles requires an excessive number of obstacles in the environments, which increases computational overhead and hurts simulation performance. Our obstacle manager allows for challenging the agent on obstacle avoidance using a small number of obstacles, by strategically sampling obstacle poses based on the generated waypoints. The manager supports uniformly distributing tree-like obstacles along line segments connecting waypoint centers, placing wall-like cuboids between waypoints, and finally making obstacles of various shapes orbit waypoints. Managed obstacles are illustrated in Figure 3(b) to 3(e). By anchoring obstacles to the racing track, we achieve efficient obstacle management. Furthermore, the difficulty level can controlled by specifying the number of obstacles in each group and parameters defining the shapes of the obstacles.

2) Training on Track Segments: Since full-length tracks are the combination of segments of shorter lengths, we believe that training on short track segments will allow generalization to full-length tracks, while reducing computational overhead for the waypoint generator, obstacle manager, and the physics engine. Plus, with shorter track lengths, we can fit full episodes into shorter rollout horizons, which increases policy update frequency and potentially reduces the wall-clock time required for the policy to converge. We generate waypoints and manage obstacles for short track segments containing only 4 waypoints. The task is to fly from the initial position near waypoint 0, pass through waypoint 1, and finally finish the episode by passing through waypoint 2. Waypoint 3 is generated to keep the dimensions of the waypoint information vector $\boldsymbol{w}_t$ consistent.

3) Environment Randomization: Combining the waypoint generator and obstacle manager, we can create an infinite amount of random environments for training. Aiming to provide the agent with diverse experience, our implementation of the waypoint generator and the obstacle manager supports vectorized environments, that is, for a single round of data collection (rollout), multiple different environments are randomly created using these tools. By incorporating experience collected in different environments into a single rollout dataset, we avoid overfitting the policy to a single environment from the ground up. For a small number of parallel environments, e.g. a few hundred, creating only a single set of random environments and using them for all rollouts is not enough, as this makes the policy learn an average strategy that maximizes the mean total reward for this specific set of environments. To solve this problem, we generate a new set of random environments for every single rollout, which further diversifies the total experience dataset.

4) Random Agent Initialization: The camera transform in the drone body frame and drone states are randomly initialized upon agent resets to make the policy more robust. This strategy can also be seen as an application of domain randomization. Specifically, we randomize the camera position in the yz-plane of the drone body frame, and the camera tilt angle. For drone states, we initialize the position within the obstacle-free zone of the initial waypoint to avoid spawning the drone into obstacles, other states such as linear and angular velocities, as well as the initial actions, are uniformly sampled within defined ranges. The initial attitude is either the same as the initial waypoint's attitude or the resulting attitude of firstly aligning the body x-axis with vector $\boldsymbol{p}_{\mathrm{wp}_1} - \boldsymbol{p}_{\mathrm{wp}_0}$, and secondly rotating about the vector for a random angle.

D. Implementation Details

We implement vectorized environments based on the Isaac Gym Simulator [34], which supports parallel physics simulation on GPUs and offers relatively high image rendering speed. To work with Isaac Gym, our code is highly optimized using PyTorch-based vectorized operations. The physics simulation frequency and angular velocity control frequency are set to 250 Hz, while the camera rendering frequency and policy control frequency are set to 25 Hz, that is, one environment step corresponds to 10 closed-loop physics steps. With this setting, we achieve about 3,000 total environment steps per second with camera sensors enabled. This speed is recorded on a consumer-grade desktop PC equipped with an Intel i9-13900K CPU and an Nvidia RTX 4090 GPU running the Ubuntu 22.04 operating system.

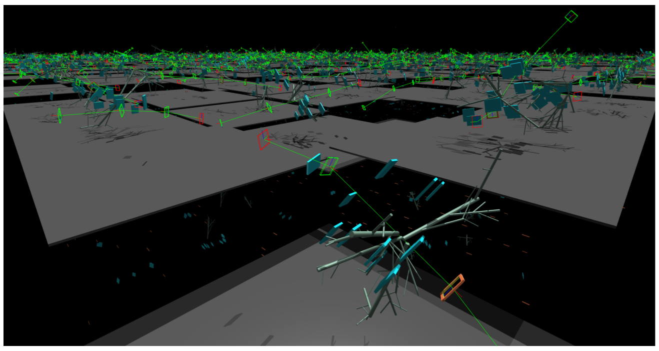


Fig. 4. Illustration of parallel environments for training. Environments are tiled up in Isaac Gym but are independent and asynchronous. Debug views of the waypoints are enabled here for visualization purposes, but are disabled during actual training.

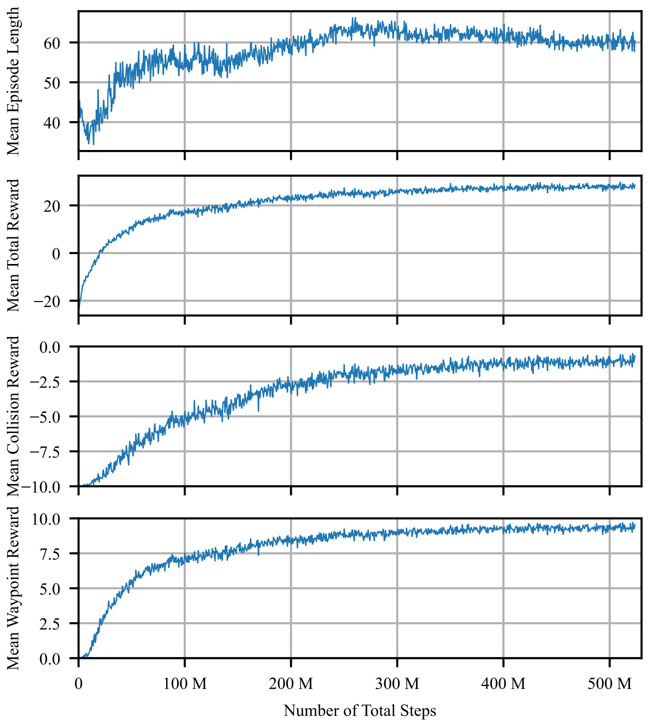


Fig. 5. Logged mean episode length in steps, mean total reward, mean collision reward, and mean waypoint reward throughout training.

Although the implemented waypoint generator and obstacle manager are capable of generating complex racing tracks with a large number of obstacles, for now, we train the policy in simpler environments. To ensure enough experience diversity within data collected through one rollout, we run 512 random environments in parallel. Every environment includes 4 tree-like obstacles taken from Aerial Gym [35], and 12 wall-like obstacles with sizes randomly specified from $(0.2, 1.5, 1.5)$ to $(0.2, 2.5, 2.5)$ in meters. For the waypoint generator, the range of waypoint sizes is set to $[1.4, 2.0]$; the initial waypoints' roll and pitch are restricted within range $[-0.2, 0.2]$, but yaw can be an arbitrary value; bounds of the relative waypoint pose parameters $(\psi_{\mathrm{wp}_{ij}}, \theta_{\mathrm{wp}_{ij}}, r_{\mathrm{wp}_{ij}}, \alpha_{\mathrm{wp}_{ij}}, \gamma_{\mathrm{wp}_{ij}})$ are $(-0.3, -0.3, 6, 0, 0)$ and $(0.3, 0.3, 18, 3.14, 0.2)$. Figure 4 shows the appearances of such environments.

We code the training loop based on the actor-critic PPO agent in RL-Games [36]. Our domain randomization happens during environment resets, so we modify the original training loop to include calling environment reset before running rollout in every training iteration. In total, we train the policy for 1,000 iterations, which corresponds to collecting experiences in 512,000 different environments for about 520 million environment steps. It takes about 50 wall-clock hours to finish all iterations. Figure 5 shows the mean episode length, total reward, and the collision and waypoint reward terms throughout training. With the collision reward set to -10 at collision, and the waypoint reward set to 5 at waypoint passing, the corresponding reward curves suggest achieving around 10% crash rate and 90% success rate at the end of training.

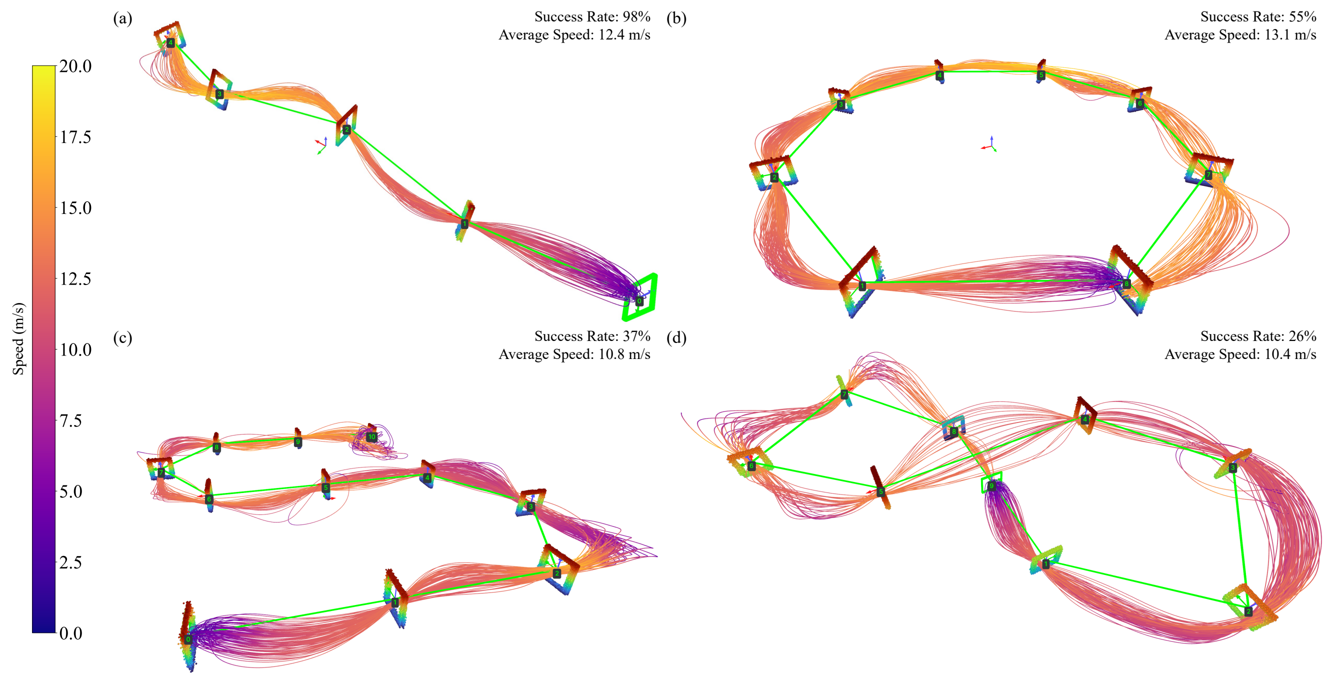


Fig. 6. Illustration of rollout trajectories of the trained policy on multiple different racing tracks without obstacles: (a) “Kebab”, (b) “Circle”, (c) “Turns”, and (d) “Wavy Eight”. Physical gate bars are represented as points colored according to their z positions in the world frame.

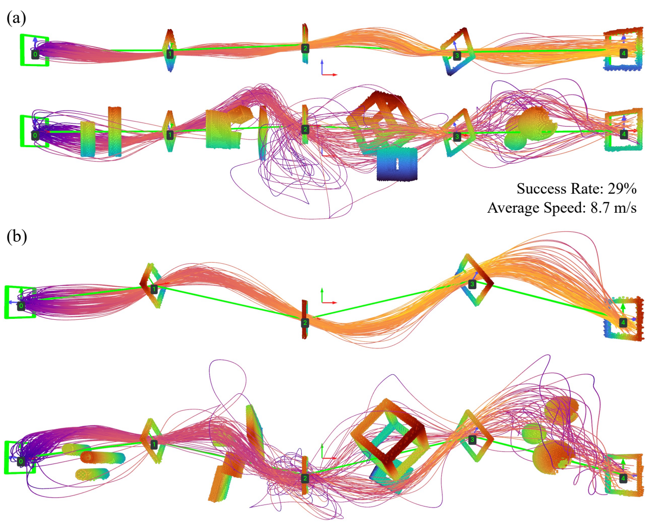


Fig. 7. Side views (a) and top-down views (b) showing distorted trajectories due to the presence of obstacles.

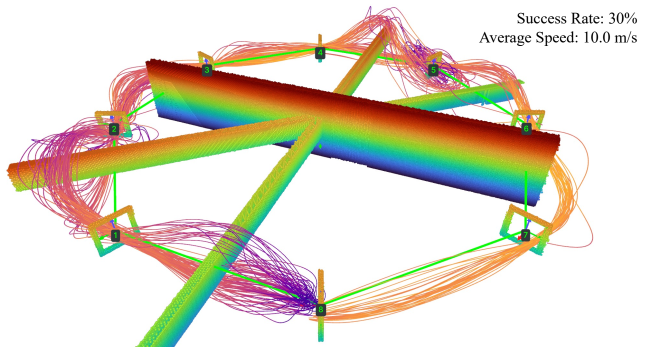


Fig. 8. Trajectories executed on the “Circle” track with additional obstacles.

IV. Experiments

A. Demonstrating Generalizable Obstacle-Aware Racing

We first demonstrate the policy’s ability to generalize to unseen waypoint placement and the number of waypoints. Four full-length tracks are designed for this experiment: “Kebab”, “Circle”, “Turns”, and “Wavy Eight”. The “Kebab” features 5 waypoints roughly in a row, representing scenarios encountered during training. The “Circle” consists of 9 waypoints uniformly distributed on a circle with a radius of 15 meters. The “Turns” has 11 waypoints positioned on a big letter “S”. Then in the “Wavy Eight”, waypoints form a figure of eight, with their z coordinates different. Since the drone’s position is part of the observation, all waypoints in the test tracks are positioned within the same environment boundaries as those used for training.

Drones are randomly initialized as in training, but we set larger ranges for initial attitude, body-frame velocities, and commands, to further “stress” the policy. Under this setup, we roll out 100 episodes for each track, log the trajectories, and calculate the success rate and the average linear speed. Results are shown in Figure 6. On track “Kebab”, the most similar to the training scenarios, the policy achieves the highest success rate. As the track gets more twists and includes more sharp turns that are not present in the training set, the success rate drops. Despite the performance drop, this experiment confirms that the policy generalizes to full-length racing tracks and completely unseen relative waypoint poses.

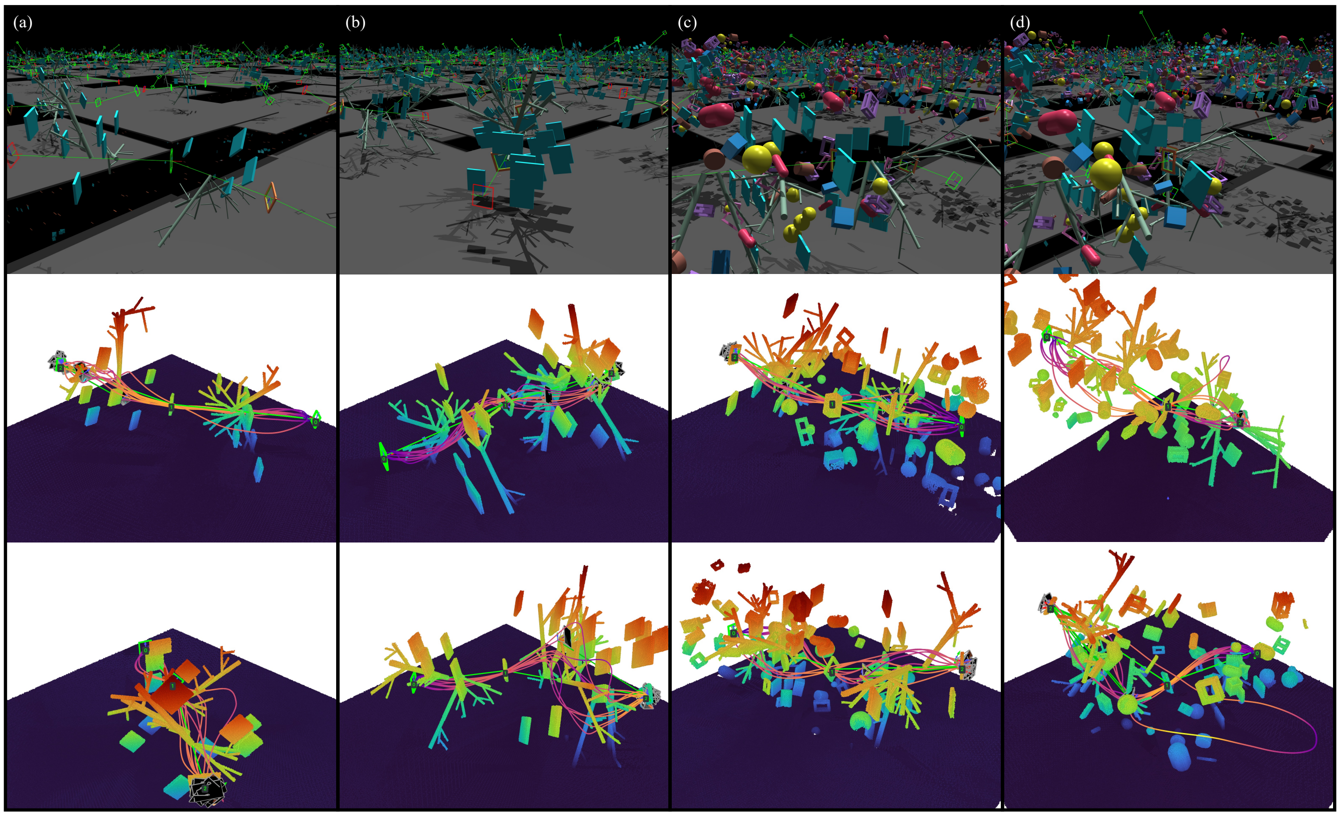


Fig. 9. Illustration of random tracks of different complexities and executed trajectories on two randomly selected tracks. Sub-figures (a) to (b) correspond to difficulty levels from 1 to 4.

To verify the generalizable obstacle avoidance ability of the policy, we deliberately place obstacles of various shapes, including ones not present in training, on track "Kebab" and "Circle" to block trajectories that might have been executed if there were no obstacles. If the policy is capable of obstacle avoidance, the resulting trajectories would be distorted. Furthermore, due to randomness in the initial drone states, we also anticipate observing non-homotopic trajectories or passing around obstacles on different sides. The results shown in Figures 7 and 8 confirm that our policy has indeed learned the generalizable ability to avoid obstacles. However in several parts of these designed tracks, the obstacle configurations have deviated too far from those in training environments, so the success rates are low.

B. Benchmarking Policy by Varying Scene Complexity

To further evaluate the capability of our policy, we assess its performance on a series of randomized tracks with varying levels of complexity. Level 1 represents the difficulty level of training environments: in each environment there are 12 wall-like obstacles and 4 tree-like obstacles, relative waypoint pose parameters $(\psi_{\mathrm{wp}_{ij}}, \theta_{\mathrm{wp}_{ij}}, r_{\mathrm{wp}_{ij}}, \alpha_{\mathrm{wp}_{ij}}, \gamma_{\mathrm{wp}_{ij}})$ are set between $(-0.3, -0.3, 6, 0, 0)$ and $(0.3, 0.3, 18, 3.14, 0.2)$. Level 2 doubles the amount of obstacles and leaves waypoint parameters unchanged. Level 3 includes 60 additional obstacles orbiting waypoint 0 and waypoint 1. Finally, level 4 sets the bounds of the waypoint parameters to $(-1, -0.4, 6, 0, 0)$ and $(1, 0.4, 18, 3.14, 0.3)$ on top of other settings in level 3.

We generate 100 random tracks per difficulty level and roll out 10 episodes per track, resulting in a total of 1000 episodes per level. Screenshots of environments in Isaac Gym, sample environments, and resulting trajectories are illustrated in Figure 9. For each trajectory, we log its termination mode, safety margin, mean and maximum values of average commands of all motors, linear speed, and angular speed. Then we can calculate the success rates and plot the data distributions of other metrics over all trajectories for all difficulty levels, as shown in Figure 10.

The success rate starts at 0.9 for level 1, consistent with the training results shown in Figure 5, but decreases as track difficulty increases, reaching around 0.4 by level 4. A similar downward trend is observed for the safety margin, indicating that the drone comes closer to obstacles on harder tracks. In terms of control effort, the maximum values of motor commands remain consistent across difficulty levels, suggesting that the policy pushes the drone to its control limits whenever possible. The mean values of commands show a clear upward trend, implying that navigating more complex tracks requires increasingly aggressive control inputs. This is also reflected in the angular speed, where both mean and maximum values increase with difficulty, as the drone must rotate more quickly to handle tighter turns and avoid obstacles. Finally, both the mean and maximum values of linear speed

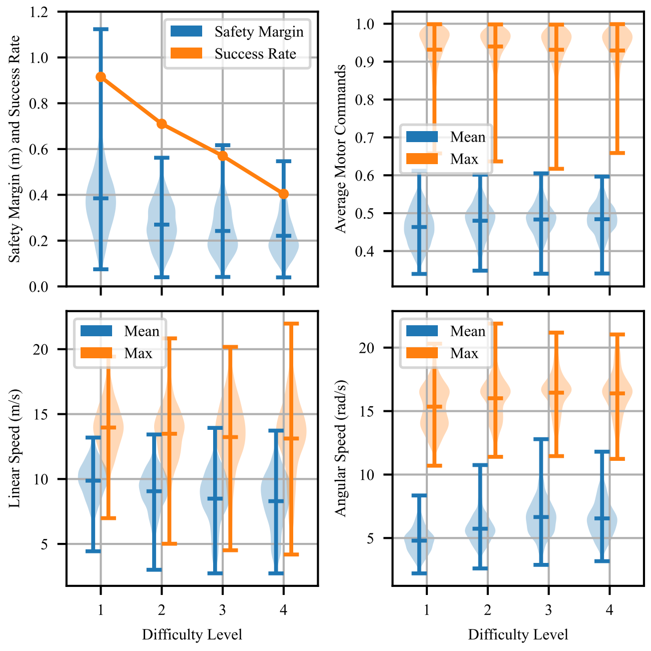


Fig. 10. Policy performance metrics across different difficulty levels. The safety margin of an episode is the minimum distance between the trajectory and the obstacle.

show a slight decrease as difficulty increases, due to the drone slowing down in response to more complex track layouts and more cluttered spaces.

These results show that the policy generalizes to unseen tracks and achieves a high success rate in similar-to-training environments. As the test set deviates from the training set, the policy can adapt to increased difficulty with higher control effort and more careful velocity management and still maintain a certain level of success rate.

C. Towards Robust Obstacle-Aware Racing Policies

We additionally evaluate the policy on four hard tracks, all characterized by shorter waypoint distances, and a higher density of obstacles. These characteristics require the drone to do sharper turns. As a result, the policy struggles to navigate, with most trajectories being unsuccessful, lowering success rates to below 0.01, as shown in Figure 11. This performance decline suggests that the current policy, trained in simple environments, lacks the robustness required to navigate more complex and constrained tracks. This limitation could be due to over-simplified environments in the training set. The training tracks have fewer obstacles, and obstacles are all distributed in a simple way, which may make the policy overfit to this specific track design. As a result, the policy struggles in new, more challenging scenarios that require advanced obstacle avoidance and tighter maneuvering.

To improve robustness, the core problem would be how to randomly create a set of training environments that better represent the complexity of real-world tracks. Once this problem is answered, we can train the policy with domain randomization to finally obtain a robust policy for obstacle-aware drone racing.

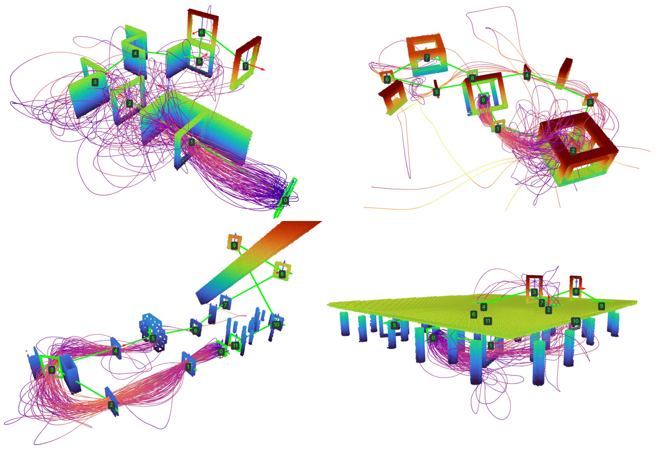


Fig. 11. Illustration of rollout trajectories on hard racing tracks where distances between waypoints are much shorter, requiring sharper turns. Most trajectories are unsuccessful.

V. Conclusion

This work presents an approach for training a generalizable obstacle-aware drone racing policy using domain randomization and deep reinforcement learning. The policy is trained on randomized short track segments, and evaluated on both hand-crafted, full-length tracks and randomized short segments across varying difficulty levels. Experiment results demonstrate that the policy generalizes well to unseen tracks and adapts to increased difficulty levels, achieving high success rates in environments closely resembling the training set. However, the policy encounters challenges in more complex, cluttered environments, where obstacle density and tighter waypoint spacing result in significant performance drops of the policy. This highlights the importance of further research into the method for generating more diverse and challenging training environments. Future work may also explore using advanced training strategies such as curriculum learning or adaptive difficulty scaling to better prepare policies for real-world drone racing challenges.

References

[1] H. Moon, J. Martinez-Carranza, T. Cieslewski, M. Faessler, D. Falanga, A. Simovic, D. Scaramuzza, S. Li, M. Ozo, C. De Wagter *et al.*, "Challenges and implemented technologies used in autonomous drone racing," *Intelligent Service Robotics*, vol. 12, pp. 137–148, 2019.

[2] E. Kaufmann, M. Gehrig, P. Foehn, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza, "Beauty and the beast: Optimal methods meet learning for drone racing," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 690–696.

[3] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar, and D. Scaramuzza, "Alphapilot: Autonomous drone racing," *Autonomous Robots*, vol. 46, no. 1, pp. 307–320, 2022.

[4] C. De Wagter, F. Paredes-Vallés, N. Sheth, and G. de Croon, "The sensing state-estimation and control behind the winning entry to the 2019 artificial intelligence robotic racing competition," *Field Robot.*, vol. 2, pp. 1263–1290, 2022.

[5] "Game of drones: A neurips 2019 competition," https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing.

[6] Q. Wang, D. Wang, C. Xu, A. Gao, and F. Gao, "Polynomial-based online planning for autonomous drone racing in dynamic environments," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 1078–1085.

[7] "Robomaster unmanned aerial vehicle intelligent perception technology competition," https://www.robomaster.com/zh-CN/robo/drone?djifrom=nav_drone.

[8] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing: A survey," *IEEE Transactions on Robotics*, 2024.

[9] R. Penicka and D. Scaramuzza, "Minimum-time quadrotor waypoint flight in cluttered environments," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 5719–5726, 2022.

[10] F. Gao, L. Wang, B. Zhou, X. Zhou, J. Pan, and S. Shen, "Teach-repeat-replan: A complete and robust system for aggressive flight in complex environments," *IEEE Transactions on Robotics*, vol. 36, no. 5, pp. 1526–1545, 2020.

[11] Z. Han, Z. Wang, N. Pan, Y. Lin, C. Xu, and F. Gao, "Fast-racing: An open-source strong baseline for se(3) planning in autonomous drone racing," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 8631–8638, 2021.

[12] R. Penicka, Y. Song, E. Kaufmann, and D. Scaramuzza, "Learning minimum-time flight in cluttered environments," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 7209–7216, 2022.

[13] Y. Song, K. Shi, R. Penicka, and D. Scaramuzza, "Learning perception-aware agile flight in cluttered environments," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 1989–1995.

[14] H. Yu, C. De Wagter, and G. C. de Croon, "Mavrl: Learn to fly in cluttered environments with varying speed," *arXiv preprint arXiv:2402.08381*, 2024.

[15] M. Kulkarni and K. Alexis, "Reinforcement learning for collision-free flight exploiting deep collision encoding," *arXiv preprint arXiv:2402.03947*, 2024.

[16] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 1205–1212.

[17] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "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)*, 2017, pp. 23–30.

[18] P. Foehn, A. Romero, and D. Scaramuzza, "Time-optimal planning for quadrotor waypoint flight," *Science robotics*, vol. 6, no. 56, p. eabh1221, 2021.

[19] A. Romero, S. Sun, P. Foehn, and D. Scaramuzza, "Model predictive contouring control for time-optimal quadrotor flight," *IEEE Transactions on Robotics*, vol. 38, no. 6, pp. 3340–3356, 2022.

[20] A. Romero, R. Penicka, and D. Scaramuzza, "Time-optimal online replanning for agile quadrotor flight," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 7730–7737, 2022.

[21] C. Qin, M. S. Michet, J. Chen, and H. H.-T. Liu, "Time-optimal gate-traversing planner for autonomous drone racing," *arXiv preprint arXiv:2309.06837*, 2023.

[22] J. Fu, Y. Song, Y. Wu, F. Yu, and D. Scaramuzza, "Learning deep sensorimotor policies for vision-based autonomous drone racing," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2023, pp. 5243–5250.

[23] J. Xing, A. Romero, L. Bauersfeld, and D. Scaramuzza, "Bootstrapping reinforcement learning with imitation for vision-based agile flight," *arXiv preprint arXiv:2403.12203*, 2024.

[24] Y. Song, A. Romero, M. Müller, V. Koltun, and D. Scaramuzza, "Reaching the limit in autonomous racing: Optimal control versus reinforcement learning," *Science Robotics*, vol. 8, no. 82, p. eadg1462, 2023.

[25] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7976, pp. 982–987, 2023.

[26] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, and D. Batra, "Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames," *arXiv preprint arXiv:1911.00357*, 2019.

[27] S. S. Desai and S. Lee, "Auxiliary tasks for efficient learning of point-goal navigation," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2021, pp. 717–725.

[28] J. Ye, D. Batra, E. Wijmans, and A. Das, "Auxiliary tasks speed up learning point goal navigation," in *Conference on Robot Learning*. PMLR, 2021, pp. 498–516.

[29] D. Hoeller, L. Wellhausen, F. Farshidian, and M. Hutter, "Learning a state representation and navigation in cluttered and dynamic environments," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 5081–5088, 2021.

[30] "Betaflight: Open source flight controller firmware," https://github.com/betaflight/betaflight.

[31] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1, pp. 99–134, 1998.

[32] M. Kulkarni and K. Alexis, "Task-driven compression for collision encoding based on depth images," in *International Symposium on Visual Computing*. Springer, 2023, pp. 259–273.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[34] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa *et al.*, "Isaac gym: High performance gpu-based physics simulation for robot learning," *arXiv preprint arXiv:2108.10470*, 2021.

[35] M. Kulkarni, T. J. L. Forgaard, and K. Alexis, "Aerial gym – isaac gym simulator for aerial robots," 2023. [Online]. Available: https://arxiv.org/abs/2305.16510

[36] D. Makoviichuk and V. Makoviychuk, "rl-games: A high-performance framework for reinforcement learning," https://github.com/Denys88/rl_games, 5 2021.

Part IV

Closure

6 Conclusion

This report details efforts devoted towards training a neural policy that can do obstacle-aware drone racing and is capable of generalizing to unseen racing tracks. At the heart of the thesis is the idea of using reinforcement learning with domain randomization over racing tracks: by randomizing the tracks and exposing our agent to diverse environments, we hope that an unseen track is just another variation of the tracks used in training, where the agent is trained to perform well. This idea is shown effective in extensive simulated experiments: the agent achieves a 90% success rate while flying at an average speed of 10 m/s in similar-to-training but not exactly seen environments. However, it is important to point out that random randomization fails if the test set is too different from the training set such that the test set can no longer be counted as a "variation". This limitation suggests that to make the policy more robust, we need better methodologies for creating the training environments, or the training set. These key results and insights have been put into the article in Part III.

Before connecting the dots, putting pieces together, and writing the final article, a significant amount of engineering effort has been made to "create the dots". A complete software stack for training drone racing policies in Isaac Gym has been developed. The software stack includes reusable modules such as the waypoint generator, procedural asset composer, racing track creator, a polynomial drone model, and a Betaflight-based angular rates controller. They can be directly used for obstacle-free drone racing tasks, which have also been demonstrated as one of the early results. Since our software leverages GPU parallelization, it reaches very high simulation speed. Using the software, we can train a policy that achieves state-of-the-art lap times in under 20 minutes using the direct training method, and we can obtain a generalizable policy by training it on 16,384 random track segments for just 10 minutes.

Let's wrap up by reflecting upon the research questions.

Research Question 1

What are existing methods in the literature that can achieve the objective in whole or partially?

Answer 1: No existing method in the literature fully achieves the desired objective. However, several learning-based approaches partially address it [32, 41, 42, 51]. This question has been answered in detail at the end of Section 2.5.

Research Question 2

If there is no existing method that achieves the objective completely, what is our proposed method?

Answer 2: Our proposed method is to apply domain randomization over environments, specifically over racing track waypoint poses and obstacle configurations, and to combine it with parallel experience collection in different randomized environments. Details are explained in the article in Part III.

Research Question 3

How does the policy trained using the proposed method perform in terms of navigation success rate, speed, and generalization ability?

Answer 3: The policy performs well in environments similar to those seen at training time, but performance decreases as the test environments get less similar to the training environments. In similar-to-training environments, the policy achieves a 90% success rate and an average linear speed of 10 m/s. In random environments with additional obstacles, the success rate drops to 40% and the average speed decreases to 7 m/s. On manually designed tracks with obstacles, the success rate may further decrease. Despite the decline in success rate, the policy's generalization ability is proven by non-zero success rates on racing tracks significantly different than training tracks.

7 Recommendations

Training Environment Generation

Training with domain randomization over tracks is effective in improving a policy's ability to generalize to unseen test tracks. However, if the test tracks are too different to be viewed as a training track variation, the policy may easily fail. This brings forward the question of how to generate a set of environments for training so that the policy is robust enough to a vast range of test cases. In the obstacle-free case, we have also observed sub-optimal performance of a policy trained on randomized tracks as compared to a policy directly trained on the target track, which also leads to the question of how to train a policy that performs as well as directly trained ones while preserving generalization ability. This question is also related to training environment generation. But there might be other factors and methodologies too as this is still an open question.

Observation Design

In this project, the depth image is clipped at a maximum range of 20 meters, normalized, and encoded via DCE. While this works in simulation, it poses limitations for real-world applications, especially when using a monocular FPV camera mounted on a drone. Future work should explore alternatives, such as integrating monocular depth estimation networks or stereo depth systems for more practical perception. Additionally, the current inclusion of the drone's normalized position in the observation limits the racing track size to the maximum observation range, which constrains performance outside this range. Investigating the removal or adjustment of these observation terms, and analyzing the resulting policy performance, could be a valuable next step.

Policy Network Architecture

The policy network used in this project is relatively simple due to time constraints. Adding a memory module, such as GRU or LSTM, could help capture temporal dependencies and improve performance in complex scenarios where the policy needs to recall past states. Moreover, the use of a stochastic policy derived from the sampled latent space of DCE adds uncertainty, which may make real-world deployment challenging. Exploring the use of the mean latent space, or deterministic policy approaches, may lead to more stable and reliable performance.

Reward Function Design

The reward function design could be improved to make learning more efficient. The guidance reward is adapted from a safety reward in [32], but the negative reward near waypoints discourages the agent from approaching them, which delays discovering the large positive reward associated with passing through them. A possible improvement would be making the reward field positive on the "inbound" sides of waypoints to guide the agent more smoothly. Additionally, there is no dense reward for obstacle avoidance, making it harder for the agent to learn this crucial behavior. Implementing a dense obstacle avoidance reward would require significant engineering, particularly for vectorized environments, but this is an important area for future exploration.

Scaling up Training

The current implementation supports running a single instance of Isaac Gym on one GPU, which limits the speed of experience collection. Vision-involved reinforcement learning typically benefits from using multiple GPUs for rendering, which can dramatically increase the steps per second achieved during training. Adapting the code to support multi-GPU rendering and training is highly recommended, as it would significantly accelerate experimental iterations. A viable approach may include porting the code to Isaac Lab and exploring cloud computing platforms like AWS or Google Cloud, which offer scalable GPU resources to support high-performance training environments.

References

[1] *2023 DRL SIM Tryouts - L.A.pocalypse*. `https://www.youtube.com/watch?v=qStacJHgtAo`. (Accessed on 04/19/2024).

[2] *2023 MultiGP Championship Track*. `https://www.multigp.com/multigp-championship/track/`. (Accessed on 04/19/2024).

[3] Hyungpil Moon et al. "Challenges and implemented technologies used in autonomous drone racing". English. In: *Intelligent service robotics* 12.2 (2019), pp. 137–148. DOI: `10.1007/s11370-018-00271-6`.

[4] Elia Kaufmann et al. "Beauty and the Beast: Optimal Methods Meet Learning for Drone Racing". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 690–696. DOI: `10.1109/ICRA.2019.8793631`.

[5] *IROS TV 2019 - Robot vs. Robot - Autonomous Drone Racing - YouTube*. `https://www.youtube.com/watch?v=w2ttHVn7Rio&list=PL9Hnb9qlvGkRgubMuiq2jd_5OY3NeCfOw&index=19`. (Accessed on 03/07/2024).

[6] Philipp Foehn et al. "Alphapilot: Autonomous drone racing". In: *Autonomous Robots* 46.1 (2022), pp. 307–320.

[7] Christophe De Wagter et al. "The sensing state-estimation and control behind the winning entry to the 2019 artificial intelligence robotic racing competition". In: *Field Robot.* 2 (2022), pp. 1263–1290.

[8] *Game of Drones at NeurIPS 2019: Simulation-based drone-racing competition built on AirSim - Microsoft Research*. `https://www.microsoft.com/en-us/research/blog/game-of-drones-at-neurips-2019-simulation-based-drone-racing-competition-built-on-airsim/?OCID=msr_blog_gameofdrones_neurips_fb`. (Accessed on 03/07/2024).

[9] Qianhao Wang et al. "Polynomial-Based Online Planning for Autonomous Drone Racing in Dynamic Environments". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2023, pp. 1078–1085. DOI: `10.1109/IROS55552.2023.10342456`.

[10] *RoboMaster | Unmanned Aerial Vehicle Intelligent Perception Technology Competition*. `https://www.robomaster.com/zh-CN/robo/drone?djifrom=nav_drone`. (Accessed on 03/07/2024).

[11] Drew Hanover et al. "Autonomous drone racing: A survey". In: *IEEE Transactions on Robotics* (2024).

[12] L Oyuki Rojas-Perez et al. "On-board processing for autonomous drone racing: An overview". In: *Integration* 80 (2021), pp. 46–59.

[13] *Github: microsoft/AirSim-NeurIPS2019-Drone-Racing: Drone Racing @ NeurIPS 2019, built on Microsoft AirSim*. `https://github.com/microsoft/AirSim-NeurIPS2019-Drone-Racing`. (Accessed on 03/08/2024).

[14] *DJI RMUA Challenge 2023 Documents*. `https://www.robomaster.com/zh-CN/resource/pages/announcement/1644`. (Accessed on 03/08/2024).

[15] Sunggoo Jung et al. "A direct visual servoing-based framework for the 2016 IROS Autonomous Drone Racing Challenge". In: *Journal of Field Robotics* 35.1 (2018), pp. 146–166.

[16] *Estimation and Control of Autonomous Racing Drone*. `https://drum.lib.umd.edu/items/6d03ea1e-3a60-43ab-92aa-4805bc498226`. (Accessed on 03/09/2024).

[17] Raul Mur-Artal et al. "ORB-SLAM: a versatile and accurate monocular SLAM system". In: *IEEE transactions on robotics* 31.5 (2015), pp. 1147–1163.

[18] Davide Falanga et al. “PAMPC: Perception-aware model predictive control for quadrotors”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 1–8.

[19] Shuo Li et al. “Visual model-predictive localization for computationally efficient autonomous racing of a 72-g drone”. In: *Journal of Field Robotics* 37.4 (2020), pp. 667–692.

[20] Michael Bloesch et al. “Robust visual inertial odometry using a direct EKF-based approach”. In: *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2015, pp. 298–304.

[21] *Game of Drones: Winning Teams’ Reports*. `https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/winning_teams_reports.html`. (Accessed on 03/10/2024).

[22] Charbel Toumieh et al. “High-speed planning in unknown environments for multirotors considering drag”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 7844–7850.

[23] *Report for Game of Drones: A NeurIPS 2019 Competition*. `https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/_files/Sangyun.pdf`. (Accessed on 03/10/2024).

[24] Sang-Yun Shin et al. “Reward-driven U-Net training for obstacle avoidance drone”. In: *Expert Systems with Applications* 143 (2020), p. 113064.

[25] Philipp Foehn et al. “Time-optimal planning for quadrotor waypoint flight”. In: *Science robotics* 6.56 (2021), eabh1221.

[26] Angel Romero et al. “Model predictive contouring control for time-optimal quadrotor flight”. In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3340–3356.

[27] Angel Romero et al. “Time-optimal online replanning for agile quadrotor flight”. In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 7730–7737.

[28] Chao Qin et al. “Time-Optimal Gate-Traversing Planner for Autonomous Drone Racing”. In: *arXiv preprint arXiv:2309.06837* (2023).

[29] Elia Kaufmann et al. “Deep drone racing: Learning agile flight in dynamic environments”. In: *Conference on Robot Learning*. PMLR. 2018, pp. 133–145.

[30] Antonio Loquercio et al. “Deep drone racing: From simulation to reality with domain randomization”. In: *IEEE Transactions on Robotics* 36.1 (2019), pp. 1–14.

[31] *AGILEFLIGHT*. `https://cordis.europa.eu/project/id/864042`. (Accessed on 03/10/2024).

[32] Yunlong 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.

[33] Jiawei Fu et al. “Learning deep sensorimotor policies for vision-based autonomous drone racing”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 5243–5250.

[34] Elia Kaufmann et al. “Champion-level drone racing using deep reinforcement learning”. In: *Nature* 620.7976 (2023), pp. 982–987.

[35] Jiaxu Xing et al. “Bootstrapping Reinforcement Learning with Imitation for Vision-Based Agile Flight”. In: *arXiv preprint arXiv:2403.12203* (2024).

[36] Yunlong Song et al. “Reaching the limit in autonomous racing: Optimal control versus reinforcement learning”. In: *Science Robotics* 8.82 (2023), eadg1462.

[37] Jesus Tordesillas et al. “MADER: Trajectory planner in multiagent and dynamic environments”. In: *IEEE Transactions on Robotics* 38.1 (2021), pp. 463–476.

[38] Xin Zhou et al. “Swarm of micro flying robots in the wild”. In: *Science Robotics* 7.66 (2022), eabm5954.

[39] Wenyi Liu et al. “Integrated Planning and Control for Quadrotor Navigation in Presence of Suddenly Appearing Objects and Disturbances”. In: *IEEE Robotics and Automation Letters* (2023).

[40] Yunfan Ren et al. “Bubble planner: Planning high-speed smooth quadrotor trajectories using receding corridors”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2022, pp. 6332–6339.

[41] Hang Yu et al. “MAVRL: Learn to Fly in Cluttered Environments with Varying Speed”. In: *arXiv preprint arXiv:2402.08381* (2024).

[42] Mihir Kulkarni et al. “Reinforcement Learning for Collision-free Flight Exploiting Deep Collision Encoding”. In: *arXiv preprint arXiv:2402.03947* (2024).

[43] Fei Gao et al. “Teach-repeat-replan: A complete and robust system for aggressive flight in complex environments”. In: *IEEE Transactions on Robotics* 36.5 (2020), pp. 1526–1545.

[44] Stephen J Wright. “Coordinate descent algorithms”. In: *Mathematical programming* 151.1 (2015), pp. 3–34.

[45] Taeyoung Lee et al. “Geometric tracking control of a quadrotor UAV on SE(3)”. In: *49th IEEE conference on decision and control (CDC)*. IEEE. 2010, pp. 5420–5425.

[46] Zhichao Han et al. “Fast-Racing: An Open-Source Strong Baseline for SE(3) Planning in Autonomous Drone Racing”. In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 8631–8638.

[47] Zhepei Wang et al. “Geometrically Constrained Trajectory Optimization for Multicopters”. In: *IEEE Transactions on Robotics* 38.5 (2022), pp. 3259–3278. DOI: `10.1109/TRO.2022.3160022`.

[48] Robert Penicka et al. “Minimum-time quadrotor waypoint flight in cluttered environments”. In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 5719–5726.

[49] Robert Penicka et al. “Learning minimum-time flight in cluttered environments”. In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 7209–7216.

[50] *Betaflight: Open Source Flight Controller Firmware*. `https://github.com/betaflight/betaflight`. (Accessed on 03/12/2024).

[51] Yunlong Song et al. “Learning perception-aware agile flight in cluttered environments”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 1989–1995.

[52] Yingjian Wang et al. “Autonomous flights in dynamic environments with onboard vision”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2021, pp. 1966–1973.

[53] Erik Wijmans et al. “DD-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames”. In: *arXiv preprint arXiv:1911.00357* (2019).

[54] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[55] Saurabh Satish Desai et al. “Auxiliary tasks for efficient learning of point-goal navigation”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2021, pp. 717–725.

[56] Joel Ye et al. “Auxiliary tasks speed up learning point goal navigation”. In: *Conference on Robot Learning*. PMLR. 2021, pp. 498–516.

[57] David Hoeller et al. “Learning a state representation and navigation in cluttered and dynamic environments”. In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 5081–5088.

[58] Mihir Kulkarni et al. “Task-driven compression for collision encoding based on depth images”. In: *International Symposium on Visual Computing*. Springer. 2023, pp. 259–273.

[59] Guangyu Zhao et al. *Learning Speed Adaptation for Flight in Clutter*. 2024. arXiv: `2403.04586 [cs.RO]`. URL: `https://arxiv.org/abs/2403.04586`.

[60] Nathan Koenig et al. “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. Ieee. 2004, pp. 2149–2154.

[61] Fadri Furrer et al. “Rotors—a modular gazebo mav simulator framework”. In: *Robot Operating System (ROS) The Complete Reference (Volume 1)* (2016), pp. 595–625.

[62] Lorenz Meier et al. “PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms”. In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2015, pp. 6235–6240.

[63] Shital Shah et al. “Airsim: High-fidelity visual and physical simulation for autonomous vehicles”. In: *Field and Service Robotics: Results of the 11th International Conference*. Springer. 2018, pp. 621–635.

[64] Winter Guerra et al. “Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 6941–6948.

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

[66] *Isaac Sim - Robotics Simulation and Synthetic Data | NVIDIA Developer*. `https://developer.nvidia.com/isaac-sim`. (Accessed on 03/14/2024).

[67] Marcelo Jacinto et al. “Pegasus Simulator: An Isaac Sim Framework for Multiple Aerial Vehicles Simulation”. In: *arXiv preprint arXiv:2307.05263* (2023).

[68] Mayank Mittal et al. “Orbit: A unified simulation framework for interactive robot learning environments”. In: *IEEE Robotics and Automation Letters* (2023).

[69] *Orbit vs OmniIsaacGymEnvs - Omniverse / Isaac Sim - NVIDIA Developer Forums*. `https://forums.developer.nvidia.com/t/orbit-vs-omniisaacgymenvs/251329/3`. (Accessed on 03/14/2024).

[70] Botian Xu et al. “Omnidrones: An efficient and flexible platform for reinforcement learning in drone control”. In: *IEEE Robotics and Automation Letters* (2024).

[71] *Development Roadmap — OmniDrones 0.1 documentation*. `https://omnidrones.readthedocs.io/en/latest/roadmap.html`. (Accessed on 03/14/2024).

[72] Andrew Szot et al. *Habitat 2.0: Training Home Assistants to Rearrange their Habitat*. 2022. arXiv: `2106.14405 [cs.LG]`. URL: `https://arxiv.org/abs/2106.14405`.

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

[74] Leonard Bauersfeld et al. “Neurobem: Hybrid aerodynamic quadrotor model”. In: *arXiv preprint arXiv:2106.08015* (2021).

[75] Leonard Bauersfeld et al. “Range, endurance, and optimal speed estimates for multicopters”. In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 2953–2960.

[76] Gerald Tesauro et al. “Temporal difference learning and TD-Gammon”. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[77] Katsunari Shibata et al. “Acquisition of box pushing by direct-vision-based reinforcement learning”. In: *SICE 2003 Annual Conference (IEEE Cat. No. 03TH8734)*. Vol. 3. IEEE. 2003, pp. 2322–2327.

[78] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[79] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[80] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144.

[81] Julian Schrittwieser et al. “Mastering atari, go, chess and shogi by planning with a learned model”. In: *Nature* 588.7839 (2020), pp. 604–609.

[82] Christopher Berner et al. “Dota 2 with large scale deep reinforcement learning”. In: *arXiv preprint arXiv:1912.06680* (2019).

[83] Peter R Wurman et al. “Outracing champion Gran Turismo drivers with deep reinforcement learning”. In: *Nature* 602.7896 (2022), pp. 223–228.

[84] *Introducing ChatGPT*. `https://openai.com/blog/chatgpt`. (Accessed on 04/24/2024).

[85] Ilge Akkaya et al. “Solving rubik’s cube with a robot hand”. In: *arXiv preprint arXiv:1910.07113* (2019).

[86] *Embodied AI Workshop*. `https://embodied-ai.org/`. (Accessed on 04/22/2024).

[87] Marc G Bellemare et al. “Autonomous navigation of stratospheric balloons using reinforcement learning”. In: *Nature* 588.7836 (2020), pp. 77–82.

[88] Killian Dally et al. “Soft actor-critic deep reinforcement learning for fault tolerant flight control”. In: *AIAA Scitech 2022 Forum*. 2022, p. 2078.

[89] Richard S Sutton et al. *Reinforcement learning: An introduction*. MIT press, 2018.

[90] Richard Bellman. “Dynamic programming”. In: *science* 153.3731 (1966), pp. 34–37.

[91] *Kinds of RL Algorithms — Spinning Up documentation*. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`. (Accessed on 04/23/2024).

[92] David Ha et al. “Recurrent World Models Facilitate Policy Evolution”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[93] Sébastien Racanière et al. “Imagination-Augmented Agents for Deep Reinforcement Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper_files/paper/2017/file/9e82757e9a1c12cb710ad680db11f6f1-Paper.pdf`.

[94] Anusha Nagabandi et al. “Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. Brisbane, Australia: IEEE Press, 2018, pp. 7559–7566. DOI: `10.1109/ICRA.2018.8463189`. URL: `https://doi.org/10.1109/ICRA.2018.8463189`.

[95] Vladimir Feinberg et al. *Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning*. 2018. arXiv: `1803.00101 [cs.LG]`.

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

[97] John Schulman et al. “Trust region policy optimization”. In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[98] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[99] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: `1602.01783 [cs.LG]`.

[100] Will Dabney et al. “Distributional reinforcement learning with quantile regression”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[101] Marc G Bellemare et al. “A distributional perspective on reinforcement learning”. In: *International conference on machine learning*. PMLR. 2017, pp. 449–458.

[102] Marcin Andrychowicz et al. “Hindsight experience replay”. In: *Advances in neural information processing systems* 30 (2017).

[103] Csaba Szepesvári. *Algorithms for reinforcement learning*. Springer nature, 2022.

[104] David Silver et al. “Deterministic policy gradient algorithms”. In: *International conference on machine learning*. Pmlr. 2014, pp. 387–395.

[105] Scott Fujimoto et al. “Addressing function approximation error in actor-critic methods”. In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[106] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[107] Fei Xia et al. “Gibson env: real-world perception for embodied agents”. In: *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE. 2018.

[108] Angel Chang et al. “Matterport3D: Learning from RGB-D Data in Indoor Environments”. In: *International Conference on 3D Vision (3DV)* (2017).

[109] Mihir Kulkarni et al. *Aerial Gym – Isaac Gym Simulator for Aerial Robots*. 2023. arXiv: 2305.16510 [cs.RO]. URL: https://arxiv.org/abs/2305.16510.

[110] *Rate Calculator | Betaflight*. https://betaflight.com/docs/wiki/guides/current/rate-calculator. (Accessed on 09/04/2024).

[111] Denys Makoviichuk et al. *rl-games: A High-performance Framework for Reinforcement Learning*. https://github.com/Denys88/rl_games. May 2021.

[112] Antonin Raffin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[113] Shengyi Huang et al. “CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms”. In: *Journal of Machine Learning Research* 23.274 (2022), pp. 1–18. URL: http://jmlr.org/papers/v23/21-1342.html.

[114] Antonio Serrano-Muñoz et al. “skrl: modular and flexible library for reinforcement learning”. In: *J. Mach. Learn. Res.* 24.1 (Mar. 2024).

[115] Albert Bou et al. *TorchRL: A data-driven decision-making library for PyTorch*. 2023. arXiv: 2306.00577 [cs.LG]. URL: https://arxiv.org/abs/2306.00577.

[116] Xue Bin Peng et al. “AMP: adversarial motion priors for stylized physics-based character control”. In: *ACM Transactions on Graphics* 40.4 (July 2021), pp. 1–20. DOI: 10.1145/3450626.3459670. URL: http://dx.doi.org/10.1145/3450626.3459670.

A
Appendix

```python
@torch.jit.script
def _compute_mixing_script(
    mix_table: torch.Tensor,
    throttle_boost_gain: float,
    thrust_linearization_throttle_compensation: float,
    thrust_linearization_gain: float,
    output_idle: float,
    pid_sum: torch.Tensor,
    cmd_t: torch.Tensor,
    throttle_low_freq_component: torch.Tensor,
):
    # find desired motor command from RPY PID, shape (num_envs, num_rotors)
    rpy_u = torch.matmul(mix_table[:, 1:], pid_sum.T).T

    # u range for each environment, shape (num_envs, )
    rpy_u_max = torch.max(rpy_u, 1).values
    rpy_u_min = torch.min(rpy_u, 1).values
    rpy_u_range = rpy_u_max - rpy_u_min

    # normalization factor
    norm_factor = 1 / rpy_u_range  # (num_envs, )
    norm_factor.clamp_(max=1.0)

    # mixer adjustment
    rpy_u_normalized = norm_factor.view(-1, 1) * rpy_u
    rpy_u_normalized_max = norm_factor * rpy_u_max
    rpy_u_normalized_min = norm_factor * rpy_u_min

    # throttle boost
    throttle_high_freq_component = cmd_t - throttle_low_freq_component
    throttle = cmd_t + throttle_boost_gain * throttle_high_freq_component
    throttle.clamp_(min=0.0, max=1.0)

    # thrust linearization step 1
    throttle /= 1 + thrust_linearization_throttle_compensation * torch.pow(
        1 - throttle, 2
    )

    # constrain throttle so it won't clip any outputs
    throttle.clamp_(min=-rpy_u_normalized_min, max=(1 - rpy_u_normalized_max))

    # synthesize output
    u_rpy_t = rpy_u_normalized + throttle.view(-1, 1)

    # thrust linearization step 2
    u_rpy_t *= 1 + thrust_linearization_gain * torch.pow(1 - u_rpy_t, 2)

    # calculate final u based on idle
    u = output_idle + (1 - output_idle) * u_rpy_t

    return u
```

Listing A.1: Python code for the mixing function.

```python
def _define_waypoints() -> List[Waypoint]:
    waypoints = [
        Waypoint(
            index=0, xyz=[-5.0, 4.75, 1.0], rpy=[0.0, 0.0, -90.0],
            length_y=1.0, length_z=1.0, gate=False,
        ),
        Waypoint(
            index=1, xyz=[-0.5, -1.0, 3.25], rpy=[0.0, 0.0, -18.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=2, xyz=[9.6, 6.25, 1.1], rpy=[0.0, 0.0, 0.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=3, xyz=[9.5, -3.8, 1.1], rpy=[0.0, 0.0, 226.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=4, xyz=[-4.5, -5.1, 3.25], rpy=[0.0, 0.0, 180.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=5, xyz=[-4.5, -5.1, 1.2], rpy=[0.0, 0.0, 0.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=6, xyz=[4.9, -0.5, 1.1], rpy=[0.0, 0.0, 79.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=7, xyz=[-2.0, 6.6, 1.1], rpy=[0.0, 0.0, 208.0],
            length_y=1.7, length_z=1.7, gate=True,
        ),
        Waypoint(
            index=8, xyz=[-0.5, -1.0, 3.25], rpy=[0.0, 0.0, -18.0],
            length_y=1.7, length_z=1.7, gate=False,
        ),
    ]

    return waypoints
```

Listing A.2: Code for defining waypoints of the Split-S racing track.

```python
def _define_obstacles() -> Tuple[List[urdfpy.Link], List[List[float]]]:
    obstacle_links = []
    obstacle_origins = []

    obstacle_links.append(cuboid_link("obstacle_0", [4.0, 0.1, 2.0]))
    obstacle_origins.append([-4.0, 2.0, 1.0, 0.0, 0.0, 0.0])

    obstacle_links.append(cuboid_link("obstacle_1", [0.1, 2.0, 2.0]))
    obstacle_origins.append([-2.0, 3.0, 1.0, 0.0, 0.0, 0.0])

    obstacle_links.append(cuboid_link("obstacle_2", [0.1, 2.0, 2.0]))
    obstacle_origins.append([2.0, 4.0, 1.0, 0.0, 0.0, 0.0])

    obstacle_links.append(cuboid_link("obstacle_3", [2.0, 0.1, 2.0]))
    obstacle_origins.append([3.0, 0.0, 1.0, 0.0, 0.0, 0.0])

    obstacle_links.append(cuboid_link("obstacle_4", [2.0, 0.1, 2.0]))
    obstacle_origins.append([5.0, -2.0, 1.0, 0.0, 0.0, 0.0])

    obstacle_links.append(cuboid_link("obstacle_5", [0.1, 2.0, 2.0]))
    obstacle_origins.append([1.0, -2.0, 1.0, 0.0, 0.0, 0.0])

    return obstacle_links, obstacle_origins
```

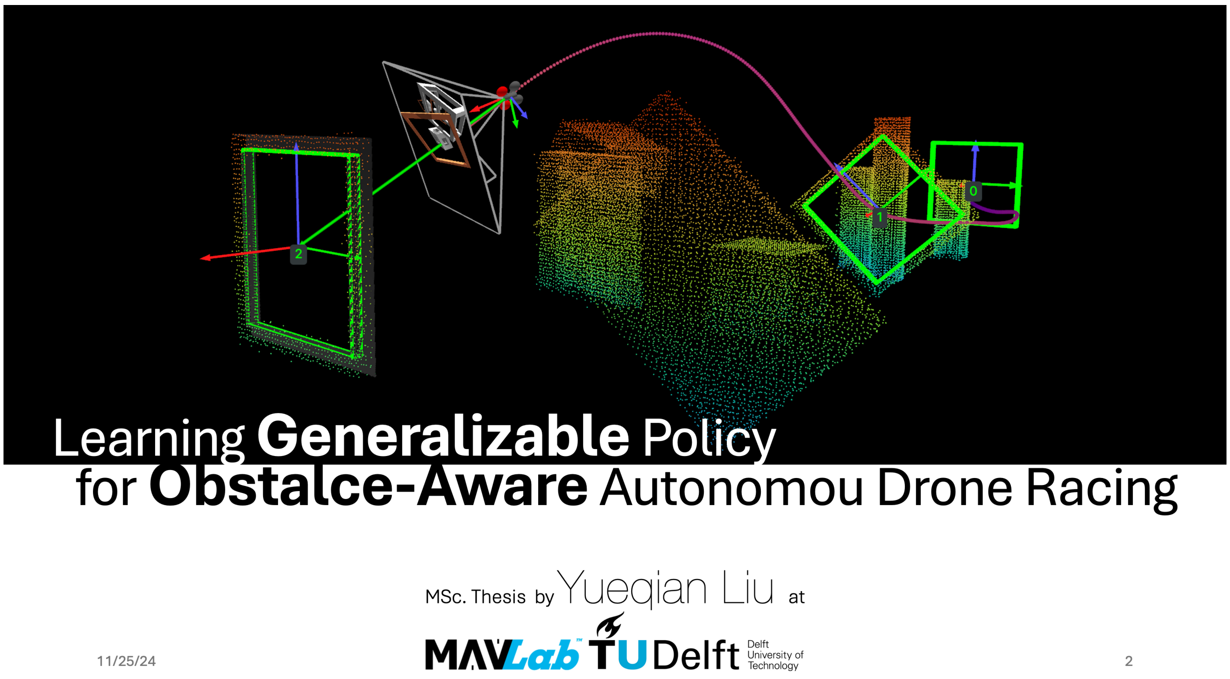
Listing A.3: Code for defining obstacles of the Walls racing track.

```python
# algos_torch.model_builder.NetworkBuilder.__init__
self.network_factory.register_builder(
    'actor_critic',
    lambda **kwargs: network_builder.A2CBuilder()
)
self.network_factory.register_builder(
    'resnet_actor_critic',
    lambda **kwargs: network_builder.A2CResnetBuilder()
)
self.network_factory.register_builder(
    'rnd_curiosity',
    lambda **kwargs: network_builder.RNDCuriosityBuilder()
)
self.network_factory.register_builder(
    'soft_actor_critic',
    lambda **kwargs: network_builder.SACBuilder()
)

# algos_torch.model_builder.ModelBuilder.__init__
self.model_factory.register_builder(
    'discrete_a2c',
    lambda network, **kwargs: models.ModelA2C(network)
)
self.model_factory.register_builder(
    'multi_discrete_a2c',
    lambda network, **kwargs: models.ModelA2CMultiDiscrete(network)
)
self.model_factory.register_builder(
    'continuous_a2c',
    lambda network, **kwargs: models.ModelA2CContinuous(network)
)
self.model_factory.register_builder(
    'continuous_a2c_logstd',
    lambda network, **kwargs: models.ModelA2CContinuousLogStd(network)
)
self.model_factory.register_builder(
    'soft_actor_critic',
    lambda network, **kwargs: models.ModelSACContinuous(network)
)
self.model_factory.register_builder(
    'central_value',
    lambda network, **kwargs: models.ModelCentralValue(network)
)

# isaacgymenvs.train.launch_rlg_hydra.build_runner
model_builder.register_model(
    'continuous_amp',
    lambda network, **kwargs: amp_models.ModelAMPContinuous(network),
)
model_builder.register_network(
    'amp',
    lambda **kwargs: amp_network_builder.AMPBuilder()
)
```

Listing A.4: Registrations of models and network builders.



Learning **Generalizable** Policy
for **Obstalce-Aware** Autonomou Drone Racing

MSc. Thesis by Yueqian Liu at

MAVLab TUDelft Delft University of Technology

11/25/24

2

Figure A.1: A video presentation is available at: `https://www.youtube.com/watch?v=f3pSJzbFjsM`.