CIS 5190: Applied Machine Learning Fall 2024

RL-based High-Speed Drone Racing

Team Members: Tony Jie Wang, Jerry (Runxuan) Wang *Supervisor: Prof. Antonio Loquercio*

Project Repository & Video Result

1 Summary

Reinforcement Learning (RL)-based drone racing has emerged as a popular research topic in robotics, showcasing significant advancements in autonomous control under high-speed, dynamic conditions. The Robotics and Perception Group (RPG) at the University of Zurich(UZH) has led groundbreaking work [1], [2], demonstrating the practical success of Sim2Real RL approaches in real-world.

Inspired by Prof. Antonio Loquercio's seminar [3], we aim to reproduce a simulated drone racing track and train a policy that enables agents to fly through the track as efficiently and quickly as possible. We utilize OmniDrones [4], a new simulation platform built on NVIDIA IsaacSim and IsaacLab [5], [6].

Our reward design focuses on three key objectives: minimizing the drone's distance to the target gates, promoting progression through the racetrack, and ensuring flight stability. We trained our policy using the Proximal Policy Optimization (PPO) algorithm [7], [8] across 250 parallel environments, accumulating a total of 150,000,000 frames per experiment to achieve sufficient exploration and learning.

As a result, the trained policy successfully enables our agents to complete a 127-meter ellipse track in 16 seconds, achieving an average speed of 7.94 m/s, with a success rate of 40.7%.

2 Core Components

Unlike supervised learning, reinforcement learning focus more on environment design, including the physical simulation environment and the observation space and rewards fuctions.

The first step of our development pipeline is building a simulation environment for data collection. For our task, this involves building racetracks consisting of rectangular gates. Then, we moved on to reward design, which aims to guide the agent through our racing task. Finally, we evaluate our results quantitatively.

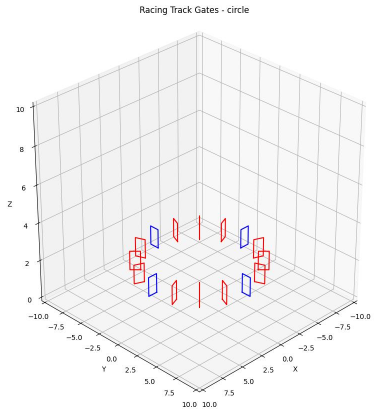
2.1 Environment Design

2.1.1 Simulation Environment Construction

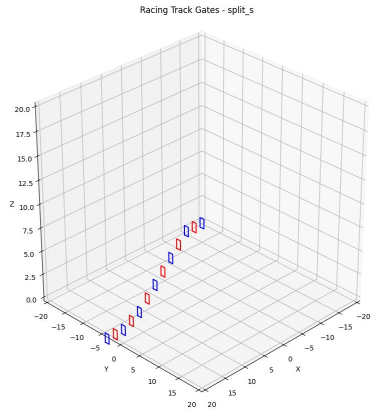
To create an effective training environment for RL-based drone racing, we designed a curriculum of racetracks with varying levels of complexity: **Simple, Median, and Hard** tasks. Each task has more challenging gate configurations progressively.

1. **Simple Tasks** We use these simple track to test our reward function step by step.
    - **Figure-8 Line**: A changing *lemniscate* requiring the drone to descend and turn back mid-flight, testing basic control and smooth transitions.
    - **Ellipse Line**: A smooth changing elliptical track, introducing curved trajectories and requiring basic trajectory-following skills.
    - **One Gate:** A single gate placed at a fixed location to test fundamental gate traversal.
    - **Two Gates:** Two sequential gates positioned linearly, challenging the drone to maintain directionality and alignment.

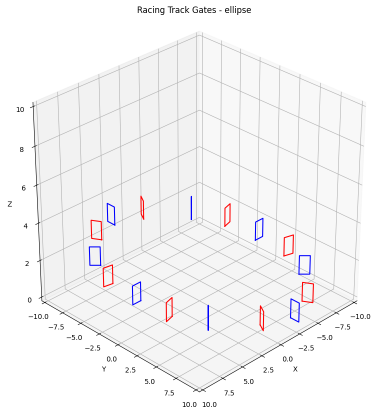
2. **Median Tasks** Median tasks introduce more dynamic and continuous motion, challenging the agent's ability to handle curved trajectories and maintain stability over longer distances.
    - Circular Track: A continuous circular layout requiring the agent to navigate a smooth, consistent trajectory while maintaining velocity.
    - Elliptical Track: An extended version of the circular track with varying curvature and longer trajectories, it includes more sharp return and long distance.
3. **Hard Tasks** Hard tasks emulate real-world drone racing challenges, requiring precise control, aggressive maneuvers, and the ability to navigate complex spatial configurations.
    - Split-S Maneuver: A demanding task combining a steep descent with a sharp directional turn, requiring the agent to optimize both speed and stability.
    - UZH Track: A replica of the UZH RPG team's championship-level racetrack, featuring multiple gates with varying orientations and complex paths.



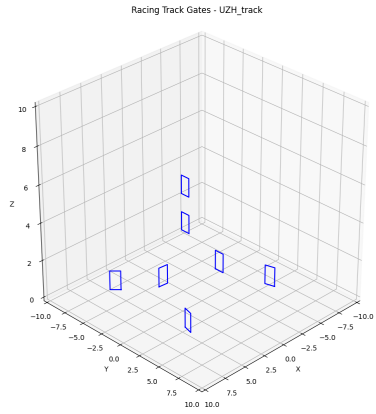
(a) Circle Track



(b) S-tunnel Track



(c) Ellipse Track



(d) UZH Track

Figure 1: Visualization of different race tracks used in our experiments. Note Split S is part of UZH Track.

Each track is defined by its position(x,y,z), orientation (Euler angles), and visibility(Boolean).

The gate configurations are stored in YAML files and are parsed in the environment setup scripts.

2.1.2 Observation Space

Unlike drone racing works such as [9] that requires Sim2Real transfer, our project is trained completely using simulation data, which contains privileged information that are difficult to obtain in real world. Our observation space is a combination of drone states, relative positions, and optional time encoding, resulting in a total dimension of **[32, 1, 33]**. Each component is described below:

Table 1: Observation Space Components

Component	**Description**	**Dimension**	**Index Range**
`drone_state`	Low-level drone state information	[32, 1, 20]	0:20
Rotation (Quaternion)	Drone's orientation	4	0:4
Linear Velocity	Drone's linear velocity in (x, y, z)	3	4:7
Angular Velocity	Drone's angular velocity in (row, yaw, pitch)	3	7:10
Heading Vector	Drone's heading vector	3	10:13
Up Vector	Drone's up direction vector	3	13:16
Motor Outputs	Thrust applied to the four rotors	4	16:20
`target_drone_rpos`	Relative position to the target	[32, 1, 3]	20:23
`gate_drone_rpos`	Relative position to the closest gate	[32, 1, 3]	23:26
`time_encoding`	Time encoding vector	[32, 1, 4]	29:33

As we learned in a blog[10] recommended by Prof. Antonio, the observation space should include only relevant information that the agent needs. So we remove the drone position data, normalize continuous observations to similar scales, let the agent has sufficient information for effective decision-making[7].

2.1.3 Action Space

The action space for our drone racing task is defined as a **4-dimensional continuous vector** representing the thrust applied to each of the drone's four rotors directly. Specifically:

$$a = [T_1, T_2, T_3, T_4] \quad \text{where} \quad T_i \in [0, 1]$$

Meanwhile, the drone robot is operated by **LeePositionController**[11], including a position/velocity controller, an attitude controller, and a rate controller[4].

To maintain stability and realism, thrust values are clipped within the specified range during training and execution.

2.1.4 Reward

The high-level idea of our reward design is to motivate the drones flying through all gates at their maximum speed, while maintaining stability and avoiding collisions with the gates. More concretely, each agent has two targets along each subsection of the track: the next gate and the target point slightly in front of the gate. Our idea is to let the drone track the target point, and use the gate to guide the drone through the gate plane within the gate's frame.

Our reward contains 6 sub-components: **position reward, gate reward, up reward, spin reward, effort reward and progress reward.** The **position reward** is an exponential function of the distance between the drone and the target point. The reward increases exponentially as the drone gets closer to the next target point. The **gate reward** motivates the drone to maintain a path that aligns with the center of each gate, and gives a reward of 1 if the drone passes the gate successfully. It penalizes if the drone's path deviates from the gate center. The **progress reward**[9] is the difference between the drone-gate euclidean

distance at the last timestamp and the current timestamp. This reward encourages the drone to move toward the next gate instead of hovering at one spot. The **up reward** motivates the drone to maintain an upright orientation. The **spin reward** discourages the drone from spinning. The **effort reward** minimizes the overall energy consumption of the drone.

The progress reward is a vital component of our reward function. We noticed that without this reward, each drone tend to hover at its current target position instead of moving on to the next one. The progress reward can be viewed as the rate at which the drone approaches the next gate, which also encourages the drone from flying at its maximum speed.

$$R = \lambda_1 R_{\text{pos}} + \lambda_2 R_{\text{gate}} + \lambda_3 R_{\text{gate}}(R_{\text{up}} + R_{\text{spin}}) + \lambda_4 R_{\text{effort}} + \lambda_5 R_{\text{progress}}$$

$$R_{\text{pos}} = e^{-\delta_{\text{target}}}$$

$$R_{\text{gate}} = \begin{cases} \sum_{y,z} (0.4 - \delta_{\text{gc}}) e^{-\delta_{\text{gc}}}, & \text{if } \delta_{\text{gp}} > 0 \\ 1, & \text{otherwise} \end{cases}$$

$$R_{\text{progress}} = \lambda_1 \left[\delta_{t-1}^{\text{gate}} - \delta_t^{\text{gate}} \right]$$

$$R_{\text{up}} = 0.5 \left(\frac{\mathbf{u}_z + 1}{2} \right)^2$$

$$R_{\text{spin}} = \frac{0.5}{1.0 + (\nu_{\text{drone},z})^2}$$

$$R_{\text{effort}} = e^{-\varepsilon}$$

where:

- δ_{target}: distance from the drone to the target
- δ_{gc}: distance from the drone to the gate center
- δ_{gp}: distance from the drone to the gate plane
- $\mathbf{u}_z$: unit vector of the drone's "up" direction
- $\nu_{\text{drone},z}$: angular velocity around the drone's z-axis
- ε: drone's effort/energy consumption

2.2 Trajectory-based Model

In addition to RL approach, we adopted trajectory-based methods inspired by [2] as a baseline benchmark to evaluate our agent's success rate on the racetrack. These precomputed trajectories serve as a reference.

Given the irregular geometric distribution of complex racetracks, we generated trajectories using linear optimization techniques. However, as highlighted in [2], these reference trajectories are not necessarily *time-optimal* or dynamically feasible for high-speed drone racing. As a result, we excluded the trajectory reward from the final RL model.

Trajectory Usage While the generated trajectories were not used for reward, they played an important role in our environment setup:

- **Track Boundary**: Trajectories are used to define the valid race boundaries. If an agent deviates more than 5 meters from the reference trajectory, the episode is terminated early to encourage consistent exploration within the racetrack.
- **Benchmarking**: The precomputed trajectories can run average of 5m/s and 60% success rate in UZH track.

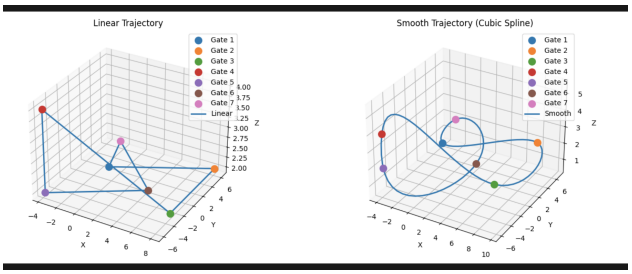


Figure 2: Comparison of trajectory mapping methods used for benchmarking.

2.3 Policy Training

We trained our policy using Proximal Policy Optimization (PPO), a popular policy gradient method. The nature of our task poses challenges for PPO due to the high-dimensional search space. To ensure efficient training, we employ a parallel training scheme similar to [2]. We utilized IsaacSim's parallel training capability to train 250 environments in parallel. This significantly increases training speed and creates a diverse set of environment interactions. Training 150,000,000 episodes takes around 6 hours on two NVIDIA A6000 GPUs.

2.4 Evaluation and Model Performance

We evaluated our policy on an ellipse racetrack with a total length of 127 meters. From 250 policy rollouts, the average task completion time is 16 seconds, giving an average velocity of 7.94 m/s, overall success rate of 40.711%.

The following are experiment record using *Weights & Biases* (wandb).

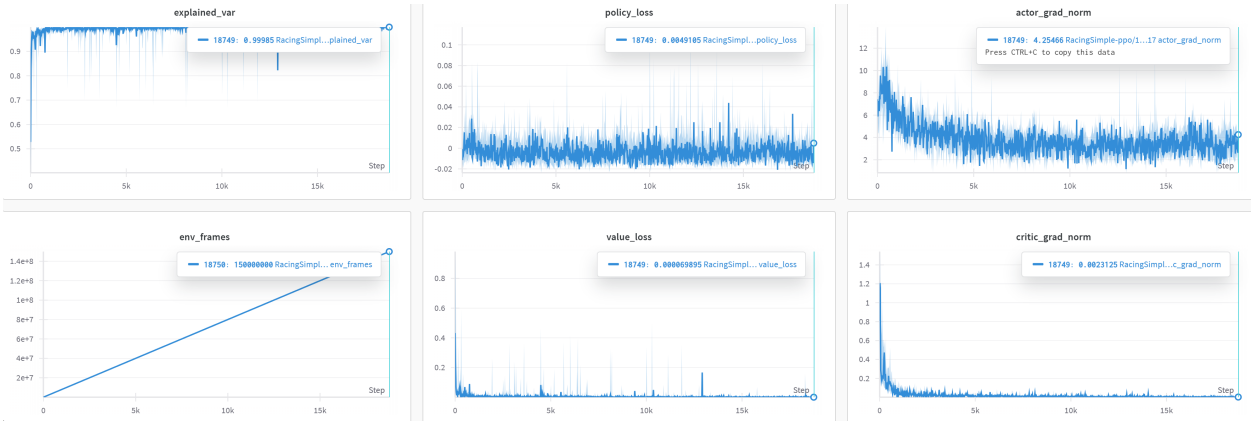


Figure 3: PPO Policy Graident

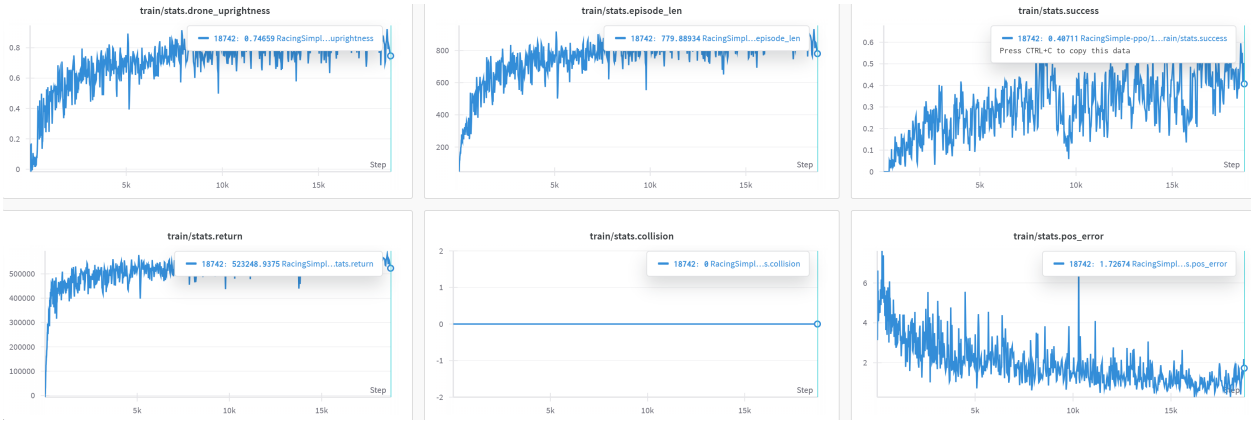


Figure 4: Train Result, with 40.711% success rate of complishing the track

3 Exploratory Questions

3.1 Reward Hacking

Initially, we observed that the reward function caused the drone to hover near the starting point or gate center. Upon further analysis, we realized this behavior was caused by overly harsh penalties for boundary violations, while the upright reward let agents tend to stay hover. As a result, the drone avoided exploration and instead remained in regions.

To address this, we improve the progress reward by changing the target position slightly behind the gates, which incentivizes the drone to move forward along the racetrack. While the time encoding adds a sense of urgency for agents. Furthermore, we assigned a significantly higher reward for successfully crossing a gate. This adjustment shifted the agent's behavior from hovering to actively progressing through the track.

3.2 Domain Randomization

Domain randomization (DR) theoretically enhances robustness and Sim2Real transfer. However, in drone racing, achieving maximum velocity on specific tracks often requires a deterministic and over-fitted model, making DR challenging to implement effectively.

For racing tasks, gate positions and initial drone states can be randomized within reasonable bounds. Careful tuning of these ranges is crucial—excessive randomization makes learning infeasible, while overly narrow ranges lack sufficient variability for robustness. As suggested by Weng [12], guided domain randomization could be explored to optimize these ranges based on task performance.

In our experiments, models trained on an elliptical track failed to generalize to circular tracks, with the complex UZH track proving even harder to adapt. This highlights the need for broader training episodes and environments to enable generalization across diverse tracks.

3.3 Simulator Trade-off

Initially, we decide to use **RLTools** [13], a DeepRL library featuring for fast training. However, the efficiency is traded off by complexity: RLTools is purely written in C++, without any outside library. We spent a lot of effort trying to work with it, the training result could be found in Appendix C. As for other simulator like **PyBullet** and **Flightmare**, they also share similar problem in development. Considering we have less than one month to work on the final project, we decide to move on OmniDrones, a more modern simulator written in Python, tailored for Drone RL training. Meanwhile, the mutli-thread, parrellel training feature of IssacSim greatly helps our development, relavant statistics is attached in Appendix D.

4 Team Contributions

Jerry's main role in the project was implementing the gate transition logic and designing the reward function. Tony primarily worked on constructing the racetrack and managing the simulation training environment. Both team members contributed equally to writing the final report.

4.1 Jerry:

- Initial ellipse track construction
- Implemented gate transition logic and through gate detection
- Reward function design & reward tuning

4.2 Tony:

- Training Environment setup & Race Track layouts
- Implemented track visualization logic and trajectory-based model
- Observation Space Design & experiments record

5 Acknowledgments

We would like to extend our sincere gratitude to Botian Xu, the author of OmniDrones [4], for his warm-hearted support in setting up the simulation environment and providing insightful discussion on reward function design.

We are also grateful to Kaitian Chao, who worked on a similar drone racing project using PyBullet in MEAM 5170. He gave many helpful advice on trajectory generation and comparisons with MPC-based control.

Finally, we express our heartfelt thanks to Prof. Dinesh Jayaraman and the members of the PAL Lab for providing access to a high-performance workstation equipped with two NVIDIA A6000 GPUs. The availability of sufficient computing power greatly accelerated the development and training processes of our project.

6 Suggestions for future iterations of this project

As far as we know, it has been many projects involved RL-based Drone Racing across the GRASP Lab and Penn Engineering Schools. The most challenging part may be reward design and racetrack engineering, which are often under-documented in original research publications. We have open sourced our code in this repo. For anyone is interested in continue research or extend on this project, welcome to contact us via *tonyw@seas.upenn.edu, runxuan@seas.upenn.edu.*

We look forward to seeing further advancements and innovations in this exciting field.

References

[1] E. Kaufmann, L. Beld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7887, pp. 737–742, 2023. DOI: 10.1038/s41586-023-06419-4. [Online]. Available: https://www.nature.com/articles/s41586-023-06419-4.

[2] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, *Autonomous drone racing with deep reinforcement learning*, 2021. arXiv: 2103.08624 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2103.08624.

[3] A. Loquercio. "Fall 2024 grasp on robotics: Antonio loquercio." Accessed: 2024-12-15. (2024), [Online]. Available: https://www.grasp.upenn.edu/events/fall-2024-grasp-on-robotics-antonio-loquercio/.

[4] B. Xu, F. Gao, C. Yu, R. Zhang, Y. Wu, and Y. Wang, *Omnidrones: An efficient and flexible platform for reinforcement learning in drone control*, 2023. arXiv: 2309.12825 [cs.RO].

[5] M. Mittal, C. Yu, Q. Yu, *et al.*, "Orbit: A unified simulation framework for interactive robot learning environments," *IEEE Robotics and Automation Letters*, vol. 8, no. 6, pp. 3740–3747, 2023. DOI: 10.1109/LRA.2023.3270034.

[6] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, *Gpu-accelerated robotic simulation for distributed reinforcement learning*, 2018. arXiv: 1810.05762 [cs.RO]. [Online]. Available: https://arxiv.org/abs/1810.05762.

[7] OpenAI. "Proximal policy optimization." Accessed: 2024-12-15. (2024), [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[8] T. Simonini. "Proximal policy optimization (ppo)." Accessed: 2024-12-15. (2022), [Online]. Available: https://huggingface.co/blog/deep-rl-ppo.

[9] 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, p. 982, 2023. DOI: 10.1038/s41586-023-06419-4.

[10] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, "The 37 implementation details of proximal policy optimization," in *ICLR Blog Track*, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/, 2022. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[11] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "Robot operating system (ros): The complete reference (volume 1)," in A. Koubaa, Ed. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625, ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[12] L. Weng, "Domain randomization for sim2real transfer," *lilianweng.github.io*, 2019. [Online]. Available: https://lilianweng.github.io/posts/2019-05-05-domain-randomization/.

[13] J. Eschmann, D. Albani, and G. Loianno, "Rltools: A fast, portable deep reinforcement learning library for continuous control," *Journal of Machine Learning Research*, vol. 25, no. 301, pp. 1–19, 2024. [Online]. Available: http://jmlr.org/papers/v25/24-0248.html.