

Drone Race Project Instructions

ESE 6510: Physical Intelligence

October 2025

1 Introduction

In this project, you will use NVIDIA Isaac Lab to train a drone racing policy! You will write the PPO algorithm into the rsl_rl learning library, construct an observation space, shape rewards, and define an episode reset strategy.

1.1 Isaac Lab/ Sim Installation

NVIDIA Isaac Lab is an open-source robot learning framework built on the Isaac Sim simulator. We will be using Isaac Sim 4.5 and an Isaac Lab fork. The following installation instructions **will not** conflict with existing Isaac Sim copies, and renaming an existing Isaac Lab directory will suffice to avoid path conflicts.

1.1.1 Isaac Sim Pip Install

- Create a new virtual environment using Conda:

```
conda create -n env_isaacclab python=3.10
conda activate env_isaacclab
```

- Ensure the latest pip is installed:

```
pip install --upgrade pip                # Linux
python -m pip install --upgrade pip      # Windows
```

- Install the CUDA-enabled PyTorch 2.7.0 build for CUDA 12.8:

```
pip install torch==2.7.0 torchvision==0.22.0 --index-url https://download.pytorch.org/whl/cu128
```

- Pip install Isaac Sim:

```
pip install "isaacsim[all,extscache]==4.5.0" --extra-index-url https://pypi.nvidia.com
```

- Test the Isaac Sim install. It can take upwards of 10 minutes for all dependencies to cache when launched for the first time.

```
isaacsim isaacsim.exp.full.kit
```

1.1.2 Isaac Lab Install

- Enter your **home** directory, then git clone our class fork of Isaac Lab:

```
git clone git@github.com:vineetpasumarti/IsaacLab.git
```

- Install dependencies using **apt** (*Linux only, not necessary for Windows*):

```
sudo apt install cmake build-essential          # Linux (not req. for Windows)
```

- Enter the IsaacLab directory and run the install command to iterate over all extensions in source. The 'none' argument skips installing the robot learning libraries for this assignment (*you will implement PPO from a local copy of rsl_rl in the project directory*)

```
cd IsaacLab
./isaacclab.sh --install none                    # Linux
isaacclab.bat --install none                     # Windows
```

- Test the Isaac Lab installation:

```
./isaacclab.sh -p scripts/tutorials/00_sim/create_empty.py    # Linux
isaacclab.bat -p scripts\tutorials\00_sim\create_empty.py     # Windows
```

1.2 Accessing Project Repo

The project repository contains the code for the drone racing environment, training, and evaluation. The repo also includes a custom copy of the rsl_rl robot learning library where you will implement Proximal Policy Optimization.

- Enter your **home** directory, then git clone the project repository. It is critical that the project repo and the Isaac Lab directory are at the same level.

```
git@github.com:Jirl-upenn/ese651_project.git
```

- In order to train, we can call the following command from terminal:

```
python scripts/rsl_rl/train_race.py \
    --task Isaac-Quadcopter-Race-v0 \
    --num_envs 8192 \
    --max_iterations 1000 \
    --headless
    --logger wandb
```

- In order to play, we can call the following example command from terminal:

```
python scripts/rsl_rl/play_race.py \
    --task Isaac-Quadcopter-Race-v0 \
    --num_envs 1 \
    --load_run [YYYY-MM-DD_XX-XX-XX] \ # The run directory is in logs/rsl_rl/quadcopter_direct/
    --checkpoint best_model.pt \
    --headless \
    --video \
    --video_length 800
```

- Neither command will work until PPO is implemented as per the next section.

2 PPO

For this project, you will use a local copy of the `rsl_rl` robot learning library, an open-source reinforcement learning library optimized for GPU-based training. You will find our class' local copy at `src/third_parties/rsl_rl_local`. Before you begin, you should create a Weights and Biases (wandb) account. This is a free cloud platform to monitor your neural network training, and you will rely heavily on it to gain insight on your drone's performance.

1. Your objective is to implement the Proximal Policy Optimization (PPO) RL algorithm by writing the `update()` method marked as `#TODO` in `src/third_parties/rsl_rl_local/rsl_rl/algorithms/ppo.py`.
2. Additionally, consider exploring different ways to compute the advantage at `compute_returns()` in `src/third_parties/rsl_rl_local/rsl_rl/storage/rollout_storage.py`. *Optional*

Since our local `rsl_rl` library is modified, directly copying from the current `rsl_rl` repository on Github will not work (*and is a breach of academic integrity*). Reading the `rsl_rl` repository to understand how PPO is implemented is okay, and can be useful to learn best practices for writing GPU-optimized code.

Once your PPO implementation is complete, you can immediately run a training with the command from Sec 1.2. This will produce a policy where the drone hovers near the zeroth gate. You should be able to match the final timestep of `best_policy.pt` and your wandb training logs with the following, respectively:

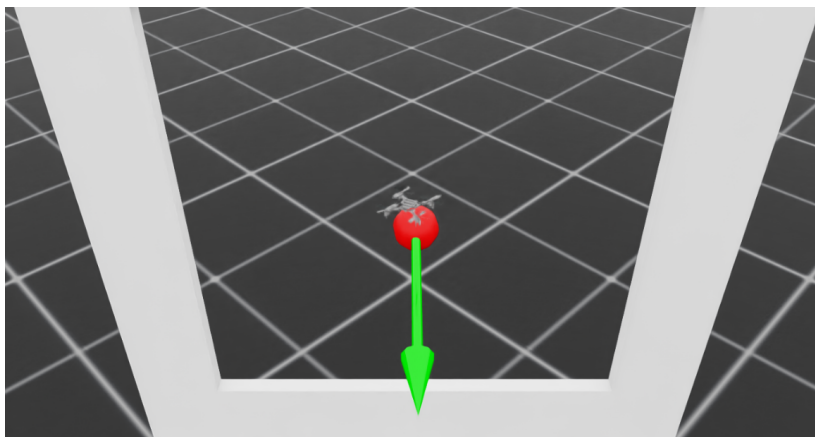


Figure 1: After implementing PPO, your first trained policy should produce a drone hovering near the zeroth gate without triggering the gate-pass condition.

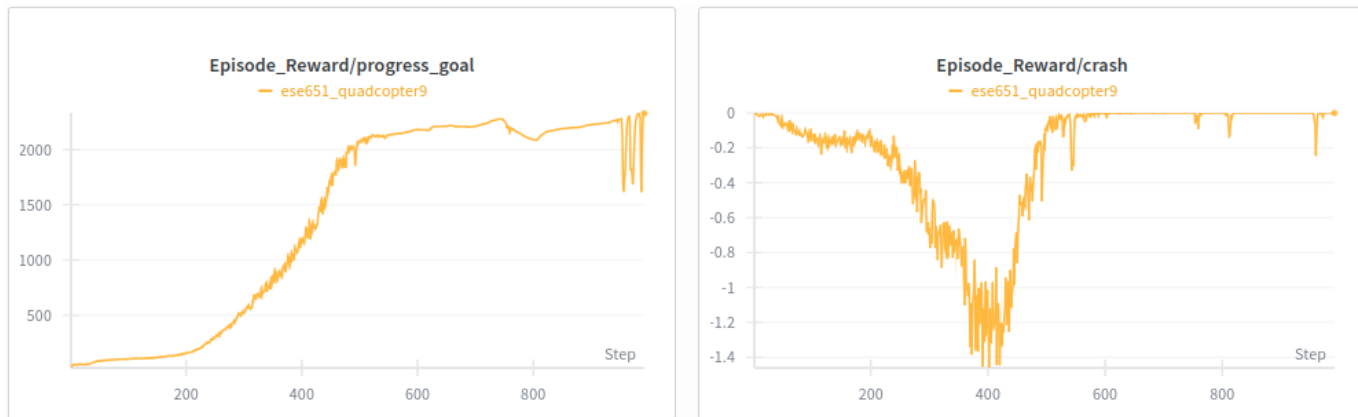


Figure 2: The episode reward plots in wandb should appear as such when training the hovering drone policy.

3 Strategy

In this section, you will complete the code marked as `#TODO` in `src/isaac_quad_sim2real/tasks/race/config/crazyflie/quadcopter_strategies.py` in order to implement a reward structure in `get_rewards()`, observation space in `get_observations()`, and drone reset strategy in `reset_idx()` for effective training. All three methods contain example code that will run and produce a simple hovering-to-zeroth-gate policy. They require significant rework to produce a strong drone racing policy. You are encouraged to further tune hyperparameters in `src/isaac_quad_sim2real/tasks/race/config/crazyflie/agents/rsl_rl_ppo_cfg.py`.

1. Design and implement a reward structure in `get_rewards()` that encourages the drone to race through gates with minimal lap time. Your implementation should:
 - Implement logic to detect when a gate is successfully traversed
 - Calculate meaningful progress metrics that reward forward movement through the course
 - Detect and penalize crashes using contact sensor data
 - Compute per-timestep rewards by multiplying your reward components with the corresponding reward scales defined in `train_race.py`

The provided example code only produces a policy that hovers near the zeroth gate and does not race. You must significantly modify or replace this code!
2. Create an observation space `get_observations()` that provides the policy with sufficient information for navigation and control. Your implementation should:
 - Extract relevant drone state information from the simulation
 - Be careful with frame transformations—decide whether observations should be in world, body, or gate-relative frames
 - Concatenate all observation tensors into a single observation vector
3. Implement a reset strategy `reset_idx()` that determines initial drone states when episodes begin. Your implementation should:
 - Define initial drone positions relative to waypoints/gates
 - Set appropriate initial orientations
 - Consider adding randomization to initial states to improve policy robustness.

3.1 Evaluation

Your drone racing policy will be evaluated through time-trials on the same track that you trained on. The primary metric is **time to complete 3 laps**. You can view your policy’s performance on a live leaderboard which we will release on Ed. In order to mimic the sim2real gap, the TAs will alter the drone dynamics in `quadcopter_env.py` so a selection of the following parameters will be sampled from anywhere within these ranges:

```
# TWR
self._twr_min = self.cfg.thrust_to_weight * 0.95
self._twr_max = self.cfg.thrust_to_weight * 1.05

# Aerodynamics
self._k_aero_xy_min = self.cfg.k_aero_xy * 0.5
self._k_aero_xy_max = self.cfg.k_aero_xy * 2.0
self._k_aero_z_min = self.cfg.k_aero_z * 0.5
self._k_aero_z_max = self.cfg.k_aero_z * 2.0

# PID gains
self._kp_omega_rp_min = self.cfg.kp_omega_rp * 0.85
self._kp_omega_rp_max = self.cfg.kp_omega_rp * 1.15
self._ki_omega_rp_min = self.cfg.ki_omega_rp * 0.85
self._ki_omega_rp_max = self.cfg.ki_omega_rp * 1.15
```

```
self._kd_omega_rp_min = self.cfg.kd_omega_rp * 0.7
self._kd_omega_rp_max = self.cfg.kd_omega_rp * 1.3

self._kp_omega_y_min = self.cfg.kp_omega_y * 0.85
self._kp_omega_y_max = self.cfg.kp_omega_y * 1.15
self._ki_omega_y_min = self.cfg.ki_omega_y * 0.85
self._ki_omega_y_max = self.cfg.ki_omega_y * 1.15
self._kd_omega_y_min = self.cfg.kd_omega_y * 0.7
self._kd_omega_y_max = self.cfg.kd_omega_y * 1.3
```

Consider strategies like domain randomization and adaptation so your policy can still succeed despite a dynamics mismatch. The evaluation environment will be held constant for all students.