Vision-Guided Outdoor Obstacle Evasion for UAVs via Reinforcement Learning

Shiladitya Dutta

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-132
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-132.html

May 23, 2025

Vision-Guided Outdoor Obstacle Evasion for UAVs via Reinforcement Learning

by Shiladitya Dutta

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Avideh Zakhor
Research Advisor

5/23/25
(Date)

* * * * * * *

Professor Ahmed Alaa
Second Reader

5/23/2025
(Date)

Vision-Guided Outdoor Obstacle Evasion for UAVs via Reinforcement Learning

Shiladitya Dutta [1]

May 22, 2025

[1]Department of Electrical Engineering and Computer Science, College of Engineering, University of California Berkeley

Abstract

Although quadcopters boast impressive traversal capabilities enabled by their omnidirectional maneuverability, the need for continuous navigation in complex environments impedes their application in GNSS and telemetry-denied scenarios. To this end, we propose a novel sensorimotor policy that uses stereo-vision depth and visual-inertial odometry (VIO) to autonomously navigate through obstacles in an unknown environment to reach a goal point. The policy is comprised of a pre-trained autoencoder as the perception head followed by a planning and control LSTM network which outputs velocity commands that can be followed by an off-the-shelf commercial drone. We employ reinforcement learning and privileged learning paradigms to train the policy in simulation via Flightmare using a simplified environment model. Training follows a two-stage process: (1) supervised initialization using optimal trajectories generated by a global motion planner, and (2) curriculum-based fine-tuning to improve policy robustness and generalization. To bridge the sim-to-real gap, we employ domain randomization and reward shaping to create a policy that is both robust to noise and domain shift. In actual outdoor experiments, our approach achieves successful zero-shot transfer to both a drone platform (DJI M300) and environments with obstacles that were never encountered during training.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Remote-controlled quadcopters are popular within commercial and enterprise markets for their unparalleled mobility in a small, cheap form factor. However, they have several key drawbacks stemming from their control process including bounds to where they can traverse due to wireless connection limitations and the need for constant operator navigation. While autopilot for following preset simple paths is a common feature, in obstacle-rich, dynamic or unknown environments autonomous navigation still is not fully realized. This impedes their utilization in GNSS and telemetry denied applications such as sub-canopy forest flight, underground mapping, war zones, and industrial inspection.

As such, an open area of research is autonomous navigation in unknown environments using only onboard computation and sensors, typically in the form of vision and depth based systems. While traditional methods decompose the navigation task into discrete planning, perception and control units, new end-to-end learning-based methods using privileged information are promising [18] [31]. These involve creating an expert policy with full environmental and state data to generate optimal paths, then distilling it into a student policy using supervised learning. However, these methods typically need accurate models of the drone so that they can use body rates or trajectories followed by a model predictive controller (MPC) to achieve agile flight which cannot be used as-is across different drone models without further tuning [18].

This paper seeks to train a policy that can be deployed onto actual drones of different sizes and classes in a zero-shot fashion without any modification. We accomplish this by pairing a deep Reinforcement Learning (RL) paradigm and privileged learning to train an end-to-end model that outputs reference velocity commands which can be followed by off-the-shelf consumer drones through built-in APIs. We take a modular approach to constructing this network where a pre-trained AutoEncoder acts as a perception head by mapping the depth image data to a low-dimensional latent space. We then freeze the perception component and train an LSTM planning/control network first with optimal trajectories as a supervisory backbone for the reward function before further fine-tuning on a curriculum environment. To cross the Sim2Real gap, we employ domain randomization and policy smoothing to ensure successful transfer on varying environments and platforms. Domain randomization teaches the policy to be robust to sensor noise and trains the LSTM to account for differing system

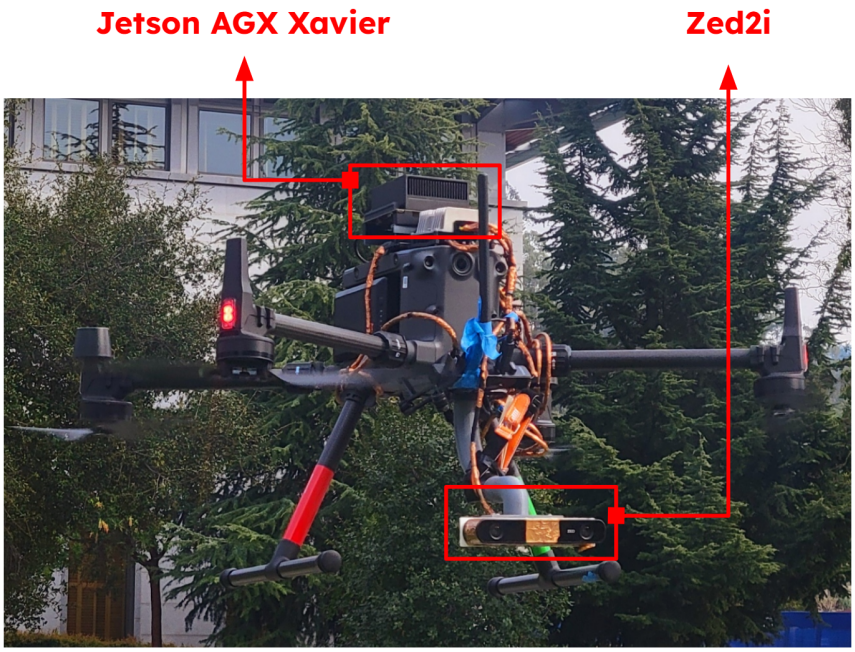

Figure 1.1: Actual testbed is a DJI M300 with an attached Zed2i sensor for depth estimation & VIO and a Jetson AGX Xavier for compute.

dynamics such as latency, inertia, etc. across time-steps [8]. Meanwhile, reward shaping prevents large action fluctuations that would degrade the drone's stability and controllability.

The result is a system that can navigate through obstacles to a goal in an outdoor environment using stereo-vision depth and visual inertial odometry (VIO). As depicted in Figure 1.1, we deploy this system onto a hardware testbed consisting of a DJI M300 with an attached Zed2i sensor for depth estimation and a Jetson AGX Xavier for compute. With real-world experiments covering a total of 650m in field trials, the policy overcomes Sim2Real gaps such as scenery changes, sensor noise, background clutter and crosswinds demonstrating successful collision-free navigation in new environments and on an off-the-shelf drone platform –the DJI M300 RTK– that was never modeled during training.

The outline of this thesis is as follows. In Chapter 2, we give an overview of related works and techniques. In Chapter 3, we give a detailed description of the methods used in our system. In Chapter 4, we summarize the implementation details of our actual drone testbed and the results from testing our system both in simulation and the real-world. Finally, in Chapter 5 we discuss future avenues to expand this work.

Chapter 2

RELATED WORK

Autonomous drone navigation broadly falls into three categories: navigation in known environments, classical navigation in unknown environments, and learning-based navigation is unknown environments. In Section 2.1, we describe methods for navigating in known environments, such as drone racing, where the map is available in advance. In Section 2.2, we give an overview of classical techniques for exploring unknown environments using traditional mapping and planning algorithms. Lastly, in Section 2.3, we discuss learning-based, end-to-end sensorimotor approaches for navigation in unknown settings.

2.1 Navigation Methods in Known Environments

There has been extensive research on methods for collision-free drone flight in cluttered environments. An established subset of this field approaches the task from a global planning perspective where perfect information about the environment is known (e.g. a 3D point cloud). These techniques focus on pre-computing dynamically feasible trajectories while optimizing for constraints such as minimizing snap [16] [21]. A related task, in terms of goals though not necessarily methods, is autonomous racing where the drones are raced through a course with known waypoints. State-of-the-art methods employ learning approaches such as Deep RL over classical pre-computed paths for their superior ability to handle noisy perception and dynamics. [12]

2.2 Classical Navigation Methods in Unknown Environments

For navigation in unknown environments, traditional methods can be separated into 3 components: perception, planning, and control. A common strategy is to use a reactive planner which searches through a set of feasible trajectories to generate a local plan. These can be split into three approaches: building a map from past observations, directly using past

observations, and memoryless methods which only use the current observation. Much of the research in this area focuses on optimizing the trajectory search space and collision-checking processes. Examples include bitwise trajectory elimination [32], sampling-based schemes to generate a free-space skeleton [10], and rectangular pyramid partitioning for efficient collision checking [3]. Other advancements include safe-stopping trajectory planning [15] and integrating semantic SLAM [17]. Despite these optimizations, end-to-end neural policies remain architecturally more efficient, as their inference is significantly simpler than tasks such as map fusion. Furthermore, hardware acceleration—leveraging NVIDIA DLA Cores or Mythic AMP—further enhances inference speed. In addition, many classical approaches –when integrated together– react adversely to compounding errors in state estimation, system latency, and sensor noise. As such, learning-based approaches replacing some or all of this 3-part stack have become a focal point of research in recent years [9]. In particular, end-to-end sensorimotor policies that directly map vision to action have proven to be effective in agile high-speed scenarios [7].

2.3 Learning-based Navigation Methods in Unknown Environments

The architecture of these end-to-end networks can be similarly broken up into a perception section and a planning/control section. For the perception network, common practices are to employ convolution layers or an image autoencoder pretrained on a separate image dataset, though [1] suggests that Vision Image Transformers have better OOD generalizability and performance. Other approaches use neural monocular depth estimation to eliminate the need for stereo cameras [35][37] and to train the autoencoder with collision meshes so that the perception network outputs can be collision-aware [13]. For control, many monocular approaches either have the perception network directly predict a steering angle and collision probability [19] or follow a state machine [20]. For end-to-end sensorimotor approaches, while fully connected networks are commonly used for the planning/control portion, some studies have proposed using Long Short-Term Memory (LSTM) [8] or attention layers [28] for their ability to implicitly compensate for sim2real factors and remembering partially observed environments over multiple time-steps. A diverse range of action outputs have been explored such as generating local trajectories in a receding horizon [18], outputting body rates [34], and directly outputting motor commands [6].

To train these networks, either reinforcement learning or imitation learning is typically used. Deep RL methods employ a drone dynamics simulator such as Flightmare [30] or AerialGym [14] to train a policy, though some works also integrate real-world data as part of the training loop to account for the sim2real gap in flight dynamics [11][5]. With imitation learning, a privileged expert provides a supervisory signal to a learned student policy, usually employing a distillation approach where the mean squared error between the student and teacher output is optimized. For the expert, some works use traditional global planning methods [18] while others use a Deep RL agent trained with privileged information and a perception-aware reward [31].

In learning-to-dodge works, imitation learning is popular for policies deployed in the real-world whereas direct RL is more common in pure-simulation works. As noted by [1], a key challenge in using RL for learning-to-dodge is balancing training variability with sensitive hyperparameter tuning and the exploration-exploitation tradeoff while ensuring the policy remains robust to domain shifts. Various works, particularly related to drone control, have proposed solutions to this problem. CAPS [22] proposes smoothing the learned state-to-action mapping of policies by introducing regularization terms for spatial and temporal smoothness. Meanwhile, [4] trains a policy with the Robust Markov Decision Process (RMDP) framework improving performance across domain shifts in simulation. Lastly, [36] trains a drone-agnostic low-level controller that can adapt to quadcopters across varying dimensions and sizes by combining privileged and reinforcement learning.

Our work is closely related to [18] and [31] in that they both aim to achieve real-world vision-based flight in cluttered outdoor environments using a learning-based approach. [31] focuses on optimizing flight in a *known* environment whereas we focus on flight in unknown environments; also [31] uses RL to train a perception-aware expert that is distilled for the policy rather than using RL to directly train the policy itself. Our approach differs from [18] in that: (a) [18] does not employ any RL techniques, and (b) [18] uses an MPC tuned to their custom-built first-person view (FPV) drone to follow outputted polynomial trajectories whereas we aim to use reference velocity commands that are drone-agnostic. To train the deployed policy, [18] and [31] use imitation learning to match the expert exactly. On the other hand, we use the relative position of the optimal vs. rollout trajectory to act as a component of the reward function in RL.

Chapter 3

METHODOLOGY

In this chapter, we give a detailed description of the methods used in our system. We first provide an overview of our approach in Section 3.1. We then discuss the RL setup including the state-action space, input processing network, actor-critic LSTMs, and reward function in Section 3.2. We then describe the training environments and the simulation loop used to run the learning components within them in Section 3.3. Next, we provide an overview of the 3 phases to train the learned policy in Section 3.4. Then, we expound upon the various measures to cross the Sim2Real gap by making the policy robust and smooth in Section 3.5. In Section 3.6, we detail optimization details for training and inference. Lastly, in Section 3.7 we describe various flaws that occurred in training and how we curtailed them.

3.1 Summary of Approach

Our approach –illustrated in Figure 3.1– trains a neural network policy to autonomously navigate a drone to a target while evading obstacles. As an input, the policy takes in a depth image and state estimates including distance to goal, relative heading to goal, and attitude from the onboard sensors. The policy then outputs vertical, horizontal, and angular velocity commands for the drone to follow. We construct this policy in two stages. We first pre-train an autoencoder, leveraging its encoder to embed depth images into a low-dimensional latent representation. We then freeze the image encoder while training an LSTM actor and state processing network in simulation using Proximal Policy Optimization (PPO). The simulator we use is Flighmare which a Unity-based quadrotor simulation framework [30]. We begin by generating a randomized training environment and computing optimal trajectories using a global motion planner [16]. We then train the policy in this environment where the optimal trajectories are a component of the reward function. Finally, we introduce a more complex curriculum environment and fine-tune the policy without optimal trajectory rewards, enhancing the policy's generalization.

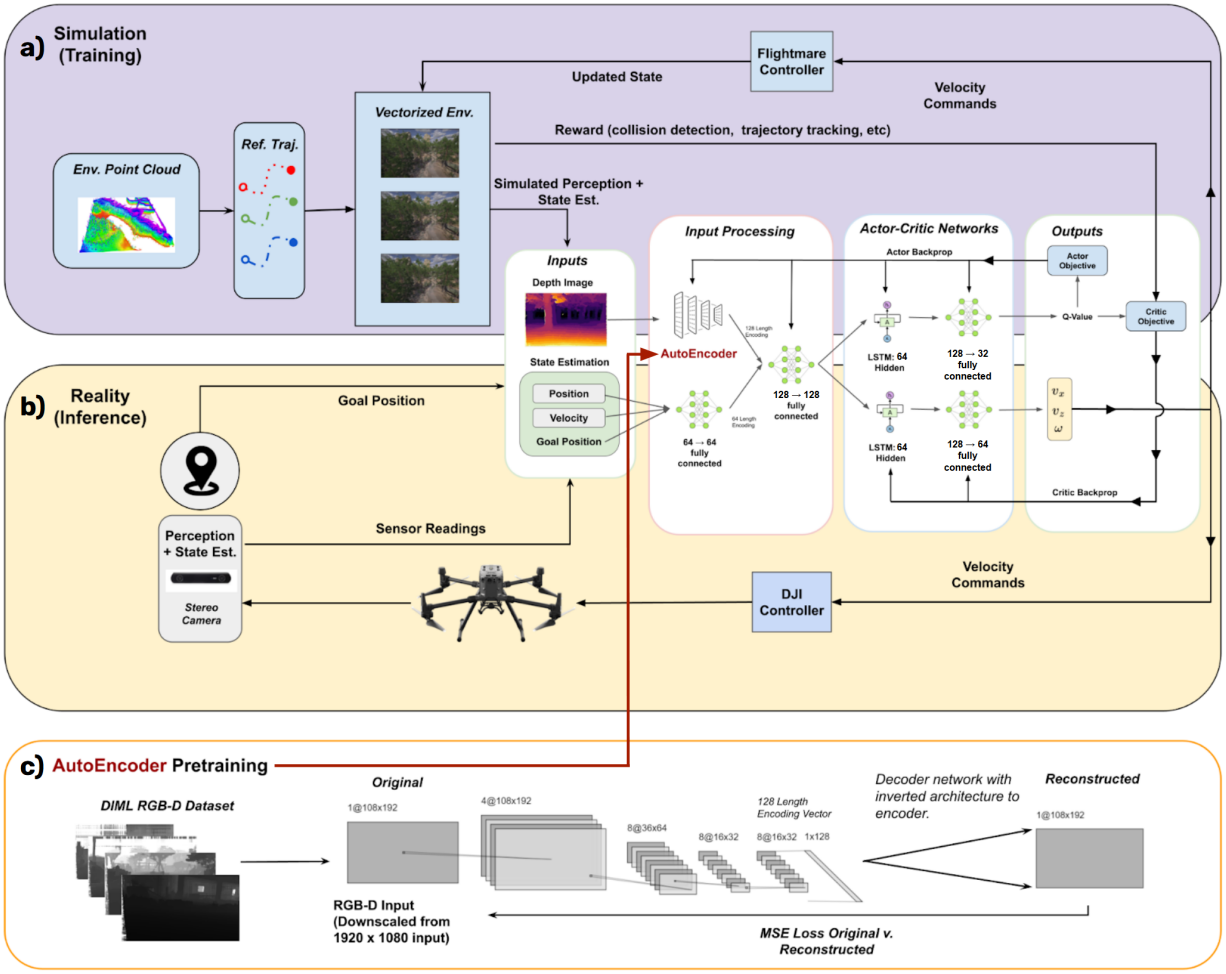

Figure 3.1: Overview of System. In (c), we first pre-train an auto-encoder to process depth images which is used as the perception head of the policy in (a) and (b). We then generate environments and calculate optimal trajectories to be used as a supervisory backbone. To train the input processing network and actor-critic networks in (a), we use PPO for optimization and Flightmare for rendering/dynamics simulation. When deploying the policy in reality with (b), we interface with the drone using the high-level DJI Controller and collect state estimates + depth image using the stereo RGB-D camera.

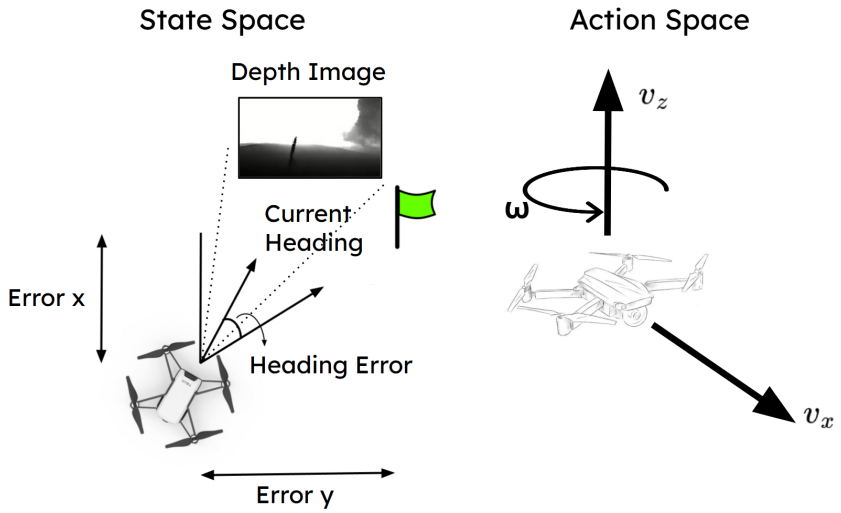

Figure 3.2: The state space consists of various measures of drone position relative to goal position alongside the depth image. The action space consists of 3 reference velocities: vertical, horizontal, and angular yaw. Based on a modified figure from [26].

3.2 Model Architecture

State and Action Space We define the drone's position at time t as $\vec{p_t} = \{p_{t,x}, p_{t,y}, p_{t,z}\}$ for x, y, and z respectively. Likewise, the goal position is $\vec{g} = \{g_x, g_y, g_z\}$. As seen on the left of Figure 3.2, the state vector at time t is $\vec{s_t} = \{g_x - p_{t,x}, g_y - p_{t,y}, g_z - p_{t,z}, \phi_t, \psi_t, \omega_t, \Delta\psi, z_t\}$ where ϕ_t, ψ_t, and ω_t are current roll, pitch, and yaw rates respectively in rad/s; $\Delta\psi$ is the difference between heading towards goal and current heading in radians; and z_t is the flattened 192×108 depth image. As seen on the right of Figure 3.2, the action at time t $\vec{a_t} = \{v_{t,x}, v_{t,z}, a_{\omega,t}\}$ consists of horizontal velocity, vertical velocity, and yaw angular velocity. These velocity commands can be directly inputted or translated for the built-in APIs of many off-the-shelf systems such as DJI or AR Parrot drones. We do not include a command for lateral y-velocity to ensure that the drone is flying in the same direction that the camera is pointed in which is necessary for safe flight due to the single camera's limited FOV.

AutoEncoder Pretraining We pre-train a denoising autoencoder using the process shown in Figure 3.1(c). The encoder portion of this network acts as a perception head for the policy by extracting a low dimensional encoding $\mathbb{R}_{128}$ from the high-dimensional depth image input $\mathbb{R}_{192\times108}$. This aids with policy function convergence by reducing the parameters that need to be trained, allowing us to tune the perception module separate from the RL simulation loop. The autoencoder is trained by feeding an image into the encoder network which compresses the image into a low-dimensional embedding before having a decoder network reconstruct the image just from the embedding. Examples of this process showcasing original versus reconstructed depth images can be found in Figure 3.3. For the training dataset, we use the DIML/CVL RGB-D dataset (`https://dimlrgbd.github.io/` which comprises 2 million RGB-D images captured across indoor and outdoor scenes. Indoor data was collected using Microsoft Kinect v2 sensors across various environments such as offices, homes, and exhibition centers in South Korea. Outdoor data was acquired using handheld

Table 3.1: The reward components, expression, and coefficients. *only used with privileged learning. Functions h and d are defined in the reward function subsection of Section 3.2.

Term	**Expression**	**Weight**
Survival	$-\lambda_1$	$\lambda_1 = 10^{-3}$
Distance to Goal	$\lambda_2 \left(1 - \frac{\|\vec{g} - \vec{p_t}\|_2}{\|\vec{g} - \vec{p_0}\|_2}\right)$	$\lambda_2 = 10^{-3}$
Heading Error	$-\lambda_3 h(\psi_t, \zeta)$	$\lambda_3 = 1/3000$
Z-position Error	$-\lambda_4 (g_z - p_{t,z})^2$	$\lambda_4 = 10^{-3}$
ω Magnitude	$-\lambda_5 a_{t,\omega}^2$	$\lambda_5 = 1/25$
v_z Direction	$\lambda_6 \begin{cases} -1.0, & \text{if } v_{t,z}(g_z - p_{t,z}) < 0 \\ 0.03, & \text{otherwise} \end{cases}$	$\lambda_6 = 1/5000$
Velocity Towards Goal	$\lambda_7 (v_{t,x} \cos(h(\psi_t, \zeta_t)))$	$\lambda_7 = 1/8000$
Acceleration	$-\vec{\lambda_8} \cdot (\vec{a_t} - \vec{a_{t-1}})$	$\vec{\lambda_8} = \begin{bmatrix} 1/20000 \\ 1/15000 \\ 1/20000 \end{bmatrix}$
Yaw Jerk	$-\lambda_9 \|(a_{t,\omega} - a_{t-1,\omega}) - (a_{t-1,\omega} - a_{t-2,\omega})\|$	$\lambda_9 = 10^{-3}$
Obstacle Proximity	$-\lambda_{10} \text{ReLU}\left(1 - \frac{\min_{o \in O} d(\vec{p_t}, o)}{3}\right)$	$\lambda_{10} = 1/3000$
Trajectory Proximity*	$\lambda_{11} \text{ReLU}\left(1 - \frac{\min_{\vec{r} \in R} \|\vec{p_t} - \vec{r}\|_2}{5}\right)$	$\lambda_{11} = 1/2000$

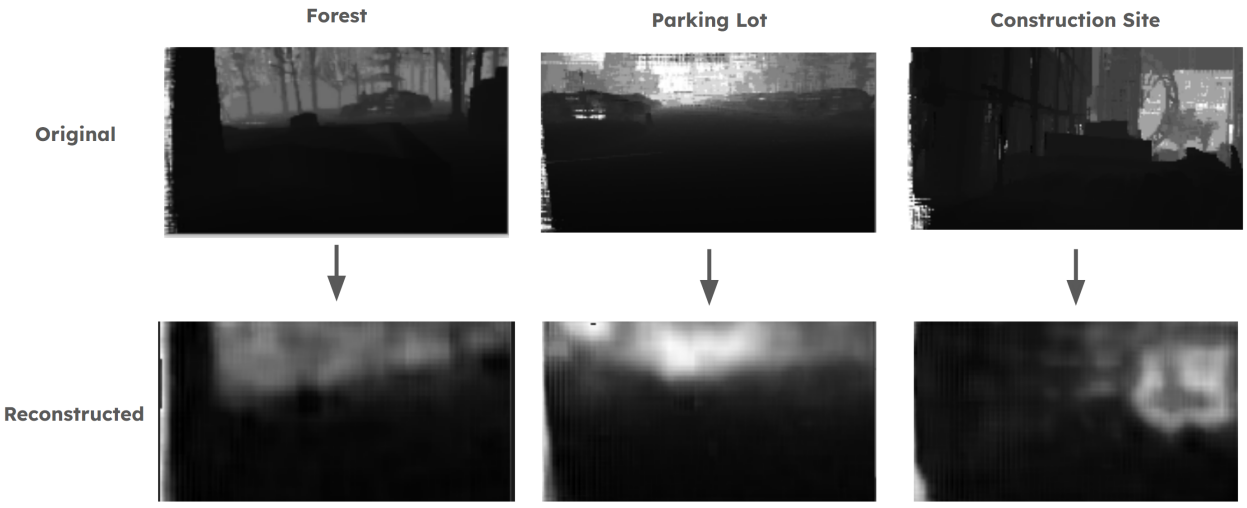

Figure 3.3: Pictures of the original images and images after being reconstructed by the autoencoder. Example depth images from 3 environments are shown: forest, construction site, and parking lot.

stereo cameras, including ZED and built-in stereo cameras, covering scenes such as parks, roads, and buildings. The depth maps were generated via stereo matching, accompanied by per-pixel confidence maps to assess depth estimation accuracy. The reasons to use this dataset are (a) it incorporates a wide variety of real-world environments and (b) it uses a Zed camera (same as ours) to obtain depth estimations.

We use a subset of 4096 images from this work in order to train the autoencoder alongside injecting Gaussian noise into the training image input which is scaled by per-pixel depth confidence score which are provided by the dataset. This approach helps with robustness by training the encoder on both a wide-range of environment types and for the noise characteristics of stereo depth measurements. We use only a small subset of the DIML dataset, as testing on the hold-out set showed that the compact perception network achieves sufficient performance even with limited training data. We train the dataset using an ADAM optimizer with a weight decay of 1e-7 and a learning rate that decays linearly from 1e-3 to 5e-5 over 200 epochs with a standard MSE reconstruction loss. We downsample the initial depth input of 1920×1080 to 192×108 using interpolation and scale the values to be between 0 and 1. Our encoder network layers are Conv, ReLU, Conv, ReLU, Max Pool, Conv, ReLU, Max Pool, and Linear to 128 unit encoding. Our decoder layers are Linear, Upsample, ReLU, Conv, Upsample, Conv, Upsample, ReLU, Conv, ReLU. All convolutional layers have a size of 3×3 and Upsample/Pool are 2×2.

Feature Extraction As visualized in the Input Processing portion of Figure 3.1(a), to extract features from the observations we feed the first 7 state values $s_{t,1:7}$ (distance to goal in x, y, and z; roll, pitch, and yaw rates; difference between current heading and heading to goal) through two fully-connected layers before being concatenated with the depth image encoding and passed through two more fully-connected layers. This yields a $\mathbb{R}_{256}$ state or feature vector which is fed into the actor-critic module of the policy.

Actor-Critic Networks and Policy Optimizer As seen in the Actor-Critic portion of Figure 3.1(a), both the actor and critic networks take in the feature/state vector and pass it through a LSTM layer with 64-dimensional hidden and memory units. The actor/critic networks then pass the LSTM outputs through two fully-connected layers (128, 32 neurons in actor and 128, 64 neurons in critic) to obtain the action $\pi(s_t)$ and Q-value $Q^\pi(s_t, a_t)$. Note, that the policy outputs range from $[-1, 1]$ and are multiplied by $v_{x,\max}, v_{z,\max}, \omega_{\max}$ to obtain the action $a_t = \{v_{t,x}, v_{t,z}, \omega_t\}$. The reason a recurrent policy was chosen rather than a feedforward architecture was because previous empirical testing with the drone in [26] revealed that it was unable to handle multiple obstacles since its policy was only computed based on the currently perceived state, meaning that as soon as an obstacle passed out of view the drone would "forget" it. Furthermore, previous research has demonstrated the efficacy of recurrent policies in both general tasks and drone navigation. In particular, [8] showed how an RNN could be useful to cross the sim-to-real gap. Specifically, it showed how RNNs could implicitly calculate time-series factors such as control latency and store them in their hidden units when trained across variable environment settings.

We considered multiple possibilities for recurrent architectures, including LSTMs, attention-based models, and transformer variants. [33] explicitly used LSTMs in both the policy and critic networks to help a drone navigate a large 3D simulated environment. Their results demonstrated that the memory capability of LSTMs enabled the drone to backtrack from dead ends and retain useful information from prior exploration. More recent work on memory-based reinforcement learning [28] leveraged a combination of temporal attention and LSTMs to enable navigation in complex mazes with dynamic obstacles. Their approach, which employed temporal attention for the Q-function to avoid the quadratic cost of standard attention, showed strong performance in challenging tasks. Additionally, we considered using Gated Transformer-XL (GTrXL), which addresses the stability issues that traditionally hinder transformers in reinforcement learning [24]. By incorporating gating layers into the transformer architecture, GTrXL achieved superior consistency and outperformed LSTMs in memory-intensive tasks such as Numpad. Ultimately, we chose to use LSTMs since we deemed that more complex memory architectures would have limited benefits given the simple nature of the task.

When designing the policy architecture, we had to decide on: shared or separate actor-critic LSTM layers, hidden unit size, and deep versus shallow networks. For the first one, we had to decide between training the LSTM to be shared between the policy function and Q-value function or to be separated. Some papers such as [33] use separate functions while others such as [28] had them shared. In theory, a shared network is more in line with the inductive bias we are trying to achieve where the LSTM is encoding core information about the state that is useful to Q-function and policy, therefore getting supervision from both. As such, we chose the LSTM layer to be common between the actor and critic. For hidden unit size, we had to choose between 64, 128, and 256 units for the hidden unit and memory unit size. The primary concern was training stability. From an on-drone computational standpoint, GPU utilization was not saturated, so increasing the size of the LSTM layer would not significantly impact performance since the number of matrix multiplication operations would stay the same. Ultimately, after empirical testing we determined that 64 units was able to achieve more stable convergence, likely due to the small representational capacity.

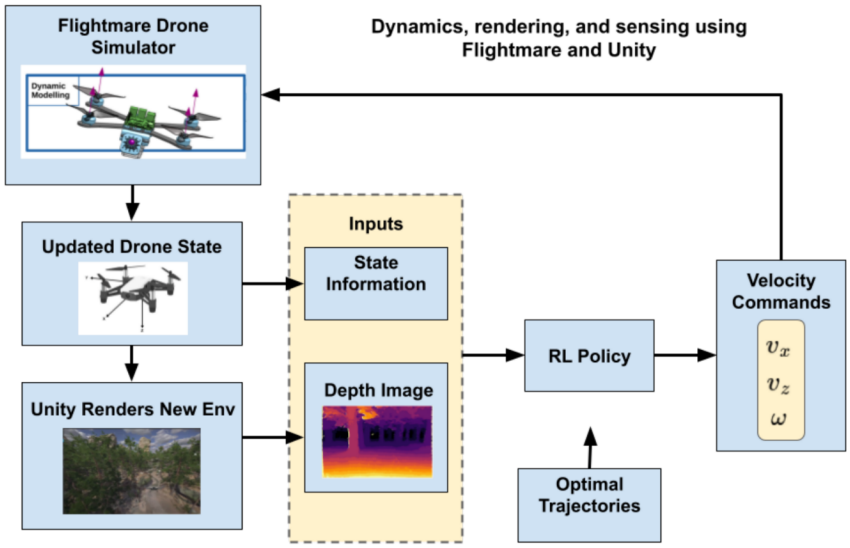

Figure 3.4: Unity renders the depth camera's PoV based on the drone state, the RL policy takes as input the image + state and outputs velocity commands, and the drone dynamics simulator uses the velocity commands to update the state.

Lastly, we had to choose between wider $512 \rightarrow 512$ layers or $128 \rightarrow 64$ layers for the fully connected network that led to the Q-function and policy outputs after the LSTM layer as shown in the actor-critic portion of Figure 3.1. Like above, this was primarily about training stability rather than on-drone computational concerns. Ultimately, we chose 64 unit hidden size and $128 \rightarrow 64$ for the feedforward sizes since the smaller networks were able to achieve more stable convergence while having substantially smaller parameter counts.

During training, we use Proximal Policy Optimization (PPO2) [27] to optimize the actor-critic and state mixing networks while freezing the encoder. PPO2 is an implementation of PPO that was designed for vectorized processing on GPUs. PPO is a popular RL algorithm that operates within an actor-critic framework, where the actor updates the policy to select actions and the critic evaluates these actions to guide learning. PPO balances performance and stability by limiting the extent of policy updates during training by employing a clipped surrogate objective function to prevent large deviations from the current policy. This enhances learning stability and sample efficiency by preventing over-exploitation of the learned critic by the actor. PPO is widely used in many robotics works due to its stability, particularly in regards to its learning rate.

Reward Function We define the beginning position of the drone as $\vec{p}_0$ and the desired heading of the drone as $\zeta = \arctan((g_x - p_{t,x})/(g_y - p_{t,y}))$. We use $o \in O$ to represent the set of all obstacles in the environment and $\vec{r} \in R$ to represent the x,y,z positions of points on the optimal trajectory. Note that the optimal trajectory generation algorithm can't be used on the actual drone since it is a global motion planner that needs perfect knowledge of the whole environment. We define function $d(p, o)$ to be the closest distance between an obstacle o and position $\vec{p}$. We define $h(a_1, a_2)$ to represent the difference between two yaw angles a_1 and a_2 when accounting for phase unwrapping. The reward for a state transition

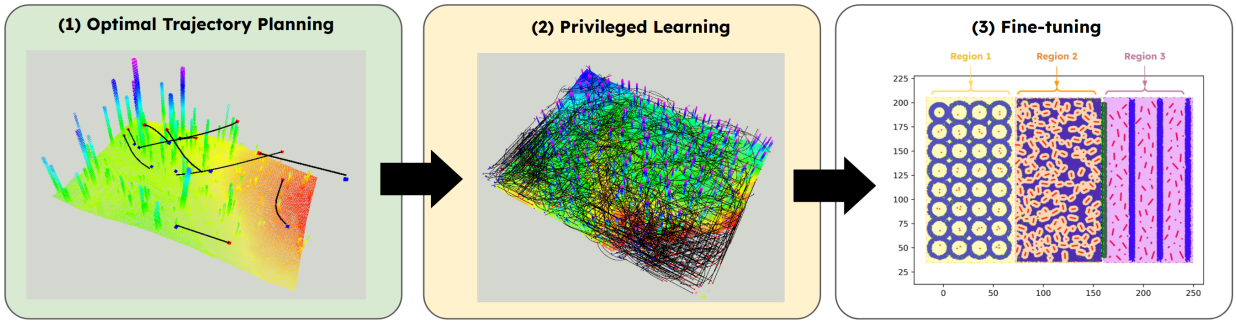

Figure 3.5: An overview of the 3 stages of the training pipeline. In (1), we generate trajectories using a motion planner with perfect environment knowledge (3D Point Cloud). Then in (2), we train an initial policy in the pillar environment with the optimal trajectories [16] as supervisory signal. Finally in (3), we fine-tune the policy in the mixed curriculum environment without any optimal trajectories.

$R(s_{t+1}, s_t, a_t)$ is calculated by adding the reward terms using the definition shown in Table 3.1. The parameters $\lambda_{1...11} \geq 0$ are empirically chosen weights. Intuitively, these rewards terms reflect either proximity to a desired state (e.g. distance to goal) or whether the agent is approaching a desired state (e.g. velocity towards goal).

In the bullet points list, we will describe the purpose of each reward term detailed in Table 3.1.

- *Survival*: a penalty for the drone surviving to discourage hovering in place.
- *Distance to Goal*: a reward for how close the drone is to the goal point.
- *Heading Error*: a penalty for the difference between the drone's current heading and the heading to the goal.
- *Z-position Error*: a penalty for how far the current height is from the goal height.
- ω *Magnitude*: a penalty for the drone turning to encourage straight line flight.
- v_z *Direction*: a penalty if the drone's vertical velocity is pointing away from the goal height and reward if it is pointing towards the goal height.
- *Velocity Towards Goal*: a reward or penalty that scales linearly with the drone's velocity in the direction of the goal.
- *Acceleration*: a penalty applied for any change in action from the previous action to discourage any horizontal, vertical, or angular acceleration.
- *Yaw Jerk*: a penalty applied for angular jerk $d\omega/dt^2$ to discourage jerky movement.
- *Obstacle Proximity*: a penalty applied if the closest obstacle to the drone is less than 3m away and the penalty scales up linearly the closer this obstacle is.

- *Trajectory Proximity*: a reward applied if the closest point in the optimal trajectory is less than 5m away from the drone and the reward scales up linearly the closer the trajectory point is.

There are four possible terminal rewards/states: reached goal (2.0), out of bounds (0.16), crashed (0.08), and timeout (-1.0). Reached goal is triggered if the drone reaches within 1.0m of the goal i.e. $\|\vec{g} - \vec{p_t}\|_2 \leq 1.0$. Crashed is triggered if the drone is within 1.25m of an obstacle i.e. $min_{o \in O} d(\vec{p_t}, o) \leq 1.25$. Timeout is if the drone is still flying for more than a maximum amount of time of 40 seconds i.e. $t_{max} \leq t, t_{max} = 480$. Lastly, the goal and start points form the diagonal corners of a rectangle in the simulation xy frame and if the drone flies more than 8m outside this rectangle then it is considered to be out-of-bounds.

3.3 Simulation and Environments

Simulation Setup We use the open-source Flightmare platform, specifically from the DodgeDrone ICRA 2022 Competition, which contains both a high-fidelity physics engine for quadrotor dynamics simulation and a graphics engine built on Unity that handles camera rendering [29]. The simulation cycle represented in Figure 3.4 is as follows: the current drone state is used by the Unity module to render the depth image view which is fed alongside the state information into the RL policy which then outputs velocity commands. The velocity commands are processed by Flightmare's built-in low-level controller to issue rotor outputs, which the dynamics modeling engine then uses to calculate the next drone state after a Δt timestep. The simulated dynamics are based on a 0.752kg Kingfisher drone. The state is fed back into Unity and this loop continues at a rate of 12 iterations per simulated second ($\Delta t = 0.085$). During training, an environment with 150 independently simulated drones is used to collect rollouts in parallel.

Environment Generation We generate two environment types: privileged learning and curriculum environments. The privileged learning environment consists of randomly placed pillars with randomized start/end locations throughout. Pillars were used because they simplify distance calculations compared to more complex 3D geometries. The curriculum environment as depicted in the fine-tuning portion of Figure 3.5 consist of three regions in increasing order of difficulty. Region 1 of Figure 3.6 is a grid of clusters where a random number of obstacles between 1 and 3 are placed within an inner 3m circle and start/end points are placed at varying 6-8m radii in an outer circle. Region 2 of Figure 3.6 consists of variable-length 2-4m walls placed at random angles with start/end points being placed randomly throughout at a maximum of 40m away from each other. Region 3 of Figure 3.6 has variable-length 2-6m walls being placed in rectangular sub-region with start/end positions being placed on either side of each sub-region. It is arranged so that the drone's trajectory passes orthogonal to the walls' orientations (+/- some angle) and therefore is forced to make larger path adjustments to avoid them. This is done by limiting the angle of the walls to be +/- 45 degrees with respect to the y-axis while having the generated start/end points have a maximum distance in the y-axis of 5m. As seen in Region 3 of Figure 3.6, the trajectories are aligned closely to the x-direction and the walls are aligned closely to the y-direction. For

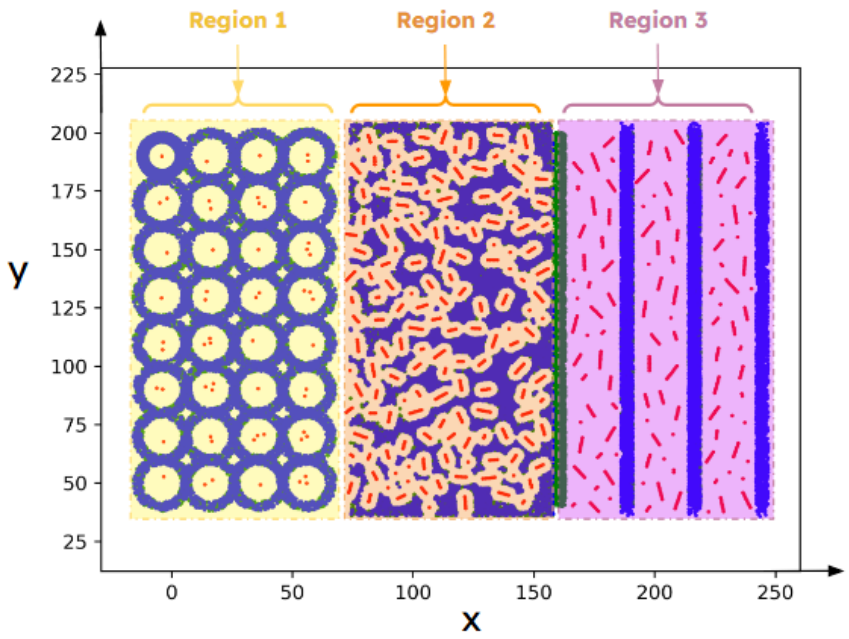

Figure 3.6: The red dots symbolize pillar/wall positions and the green/blue regions symbolize training start/end locations. Axes units are meters.

both Region 2 and 3, the walls have a minimum distance of 6m from each other and each wall consists of individual pillar segments.

3.4 Training

We train our policy using the 3 steps visualized in Figure 3.5: trajectory planning, privileged learning, and fine-tuning.

Optimal Trajectory Planning Given the privileged learning environment, we use Unity to convert the environment mesh into a 3D point cloud with a resolution of 0.15m. The point cloud is fed into a hierarchical motion planner that generates a set of dynamically feasible optimal trajectories for the given start/goal pairs. The motion planner we use is based on "Search-Based Motion Planning for Aggressive Flight in SE(3)" [16] which introduces a trajectory planning method tailored for quadrotors executing aggressive maneuvers in cluttered environments. The authors model the quadrotor as an ellipsoid, enabling the planner to consider the vehicle's orientation (roll and pitch) during flight, which allows navigation through gaps smaller than its diameter by adjusting its attitude. To efficiently generate dynamically feasible trajectories, the approach employs motion primitives—short-duration control inputs derived from optimal control solutions—that are assembled using a graph search algorithm. To mitigate the computational complexity inherent in high-dimensional planning, the method first conducts a lower-dimensional search to obtain a coarse path, which then serves as a heuristic to guide the refinement of a full SE(3) trajectory. We cannot use [16] directly since it is a global motion planner which needs perfect environment knowledge

to operate and as such isn't suitable for unknown environments.

Privileged Learning with Trajectories We teach the policy the basics of optimal time flight behavior in a simple environment of spaced-out pillars. This is aided via privileged learning whereby if the drone's position at each time step is near the optimal trajectory then it is given a reward. This differs from distillation methods since we are not trying to emulate the trajectories exactly, but rather they are a guide that helps the policy converge by giving it intermediate rewards. Furthermore, the position-based reward rather than action-based reward means that the supervisory signal acts more on the planning behavior rather than the control.

Fine-tuning in Curriculum Environment After a base network is trained, we further train the policy in the more complex curriculum environment –illustrated in Figure 3.5– to teach the policy navigation strategies, to make it robust to potential environment variations, and to prevent memorization. Over the course of fine-tuning, we start off in the easiest region (1) and gradually mix in start/goal pairs from the harder regions (2/3) as training progresses. The reason we do not use optimal trajectories during fine-tuning is that the space of optimal trajectories is multi-modal e.g. equally valid to go right or left around a single obstacle. While this effect is negligible with the simpler environments of the privileged learning step, in the more complex curriculum environment it is exacerbated since the number of near-optimal trajectories is large, implying that rewards for adhering to a particular trajectory is a poor signal that actually impedes learning.

3.5 Bridging Sim2Real

To close the Sim2Real gap, we focus on improving the policy's noise robustness and action smoothness. The reason for the former is intuitive – in the real-world there is noise across the state and action space: noise in the depth image, drift in the IMU/VIO measurements, randomness in latency, errors in rotor control, etc. The intuition behind striving for smoother policies is twofold. Firstly, while jerky movements may be optimal in simulation with an idealized model of flight dynamics, in the real-world they strain the rotors and degrade control authority. Secondly, jerky movement profiles do not generalize well to quadrotors with different characteristics (moments of inertia, thrust maps, body drags, etc.), thereby impeding zero-shot transfer.

Reward Shaping A key problem with using reference velocities as an output is that we can not enforce trajectory smoothness via methods such as interpolation. Thus, to discourage high-frequency jerky commands, we apply penalties to yaw rate magnitude, acceleration, and jerk as seen in the ω Magnitude, Acceleration, and Yaw Jerk terms in Table 3.1. To discourage low-frequency oscillations which lead to wavy motion, we apply a reward if the drone is angled towards the goal and if the its trajectory matches the smooth optimal trajectory as seen in the Heading Error and Trajectory Proximity terms in Table 3.1.

Domain Randomization We apply a small amount of normal noise ($\pm 2\%$) to positional state measurements and a 5×5 Gaussian Blur with a randomized σ in the interval [0.1, 0.7] to the depth input image. We have found empirically that Gaussian Blur degradation

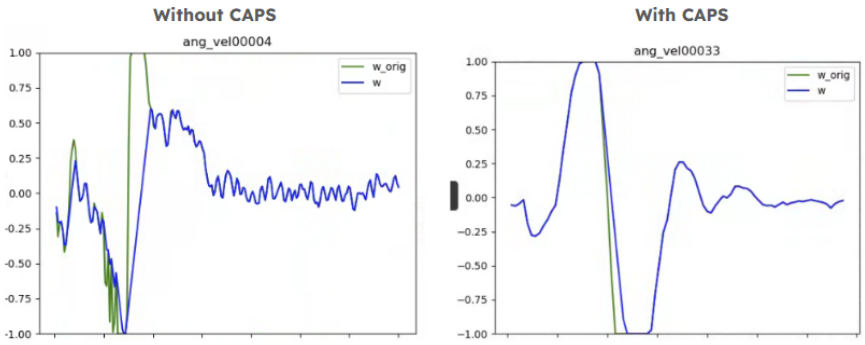

Figure 3.7: In the same scenario, the policy trained without CAPS on the left has a substantially more jittery angular acceleration than the policy trained with CAPS.

most closely approximates the type of noise encountered in real-world outdoor testing where the long range (>20m) and variable lighting lead to large variations in measurements near edges if the disparity between the foreground and background depth is large. For the action space, we apply normal noise ($\pm 5\%$) to the drone's simulated movement at each time step to emulate inaccuracies in control output and outdoor factors such as wind. In addition, we also varied the flight dynamics parameters such as mass and moments of inertia by ($\pm 10\%$) to simulate different drone testbed characteristics. We also add a lag factor to approximate system delays such as forward pass processing time and publish-poll time differences between ROS nodes.

Policy Smoothness The complex nonlinear nature of Deep RL networks means that they suffer from volatile behavior, particularly if being used for continuous control. In particular, The high local Lipschitz constant of actor networks means that there are large action fluctuations in response to only slight state variations. This characteristic leads to jerky action outputs between consecutive time-steps and large behavioral divergences in response to only slight amounts of noise. Consequently, many approaches have the network predict multiple future waypoints, which are then interpolated using a B-spline to ensure smooth and continuous trajectories. However, this approach is both more expensive due to the larger action space and harder to generalize across different drones since the policy doesn't directly output velocity commands. Thus, to train smooth and robust policies, we use Conditioning for Action Policy Smoothness (CAPS) regularization [22]. CAPS operates by adding loss terms for both temporal smoothness (similarity between consecutive actions: $\|\pi(\vec{s_t}) - \pi(\vec{s_{t-1}})\|_2$) and spatial smoothness (similarity in actions between similar states: $\|\pi(\vec{s_t}) - \pi(\vec{s_t'})\|_2, \vec{s_t'} \sim N(\vec{s_t}, \sigma)$). Using this method increases the smoothness of the policy's state-to-action mapping, thereby curtailing volatile characteristics that degrade control integrity. This is illustrated in Figure 3.7, which compares a policy trained without CAPS to one trained with CAPS. The policy without CAPS exhibits significantly more jittery angular acceleration in the same scenario, resulting in higher jerk and increased motor strain.

3.6 Training and Inference Implementation

We use the StableBaselines3 RL framework due to its relatively simple API and broad range of support. In particular, we used sb3-contrib which is a branch of StableBaselines3 which expands support to recurrent policies such as LSTMs. This new branch was necessary since the optimal sampling procedure differs from feedforward policies to recurrent policies. In feedforward policies, it is possible to sample individual experiences (state, action, next state, reward) independently since the feedforward policy runs each step in isolation. However, for LSTMs and RNNs, they take input from past states and thus their policies are conditional on past actions. As a result, rather than individual experiences being able to be sampled to form a batch, the experiences in a batch should be as contiguous as possible to ensure correct training. This turned into a major issue since this form of sampling meant that batch sizes had a tendency to vary depending on the average trajectory length. As such, a reasonable balance had to be struck between the batch size and maximum trajectory length to prevent the batches from growing larger than the available VRAM and the training had to be halted due to overflow. However, at the same time, it is necessary that the batch sizes be as large as possible since this allowed for more stable convergence. This is because large batches implicitly enable a form of curriculum training whereby easy experiences succeed enabling slow learning of difficult setups. In particular, we found that small batch sizes led to stalled convergence when domain randomization was present. To resolve this, we modified the sb3-contrib implementation to support Pytorch's DistributedDataParallelism (DDP) feature. DDP is a wrapper designed to facilitate efficient training of deep learning models across multiple GPUs and machines. It operates by launching separate processes for each GPU, each maintaining its own model replica and optimizer. During training, DDP synchronizes gradients across these processes using collective communication operations, ensuring consistent model updates and improved training performance. Modifying StableBaselines3 with DDP doubled our effective VRAM across two GPUs and set our batch size to 10000 without risking running out of VRAM. Training was performed using two Quadro RTX 8000 GPUs, each equipped with 48 GB of VRAM.

For on-drone inference, we originally wrote the processes in Python and orchestrated them via Robot Operating System (ROS). ROS is an open-source framework for developing and managing robot software, providing tools, libraries, and communication infrastructure to build modular and reusable robotics applications. It works off a microservices architecture where individual processes that may represent hardware abstraction or algorithms publish and poll from other processes allowing integration of sensors, actuators, and control algorithms. There were a few limitations to this implementation. The first is that our processes were too granular and small leading to many of individual processes all running in parallel as visualized in Figure 3.8. This was further exacerbated by the fact that Python does not support true multi-threading and parallel processing. This meant that the individual processes were not able to listen to shared ports in parallel, leading to massive lag and variance in the response time. To fix this, we rewrote the processes in C++ and merged most of the individual processes together leading to only three remaining core processes: /image_pub to represent the Zed2i, /policy to manage the policy, and /vehicle_node to connect to the M300. This allowed for significantly less overhead from process management as well as less variance in response times

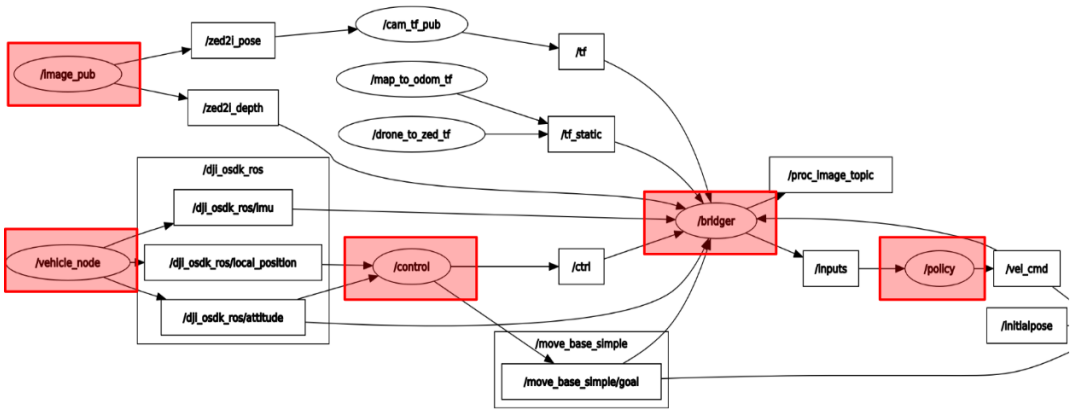

Figure 3.8: The ROS graph that was originally used with the 4 major nodes: vehicle_node for the drone commands, image_pub for the Zed2i, policy which runs the policy, and bridger which connects all the containers.

originating from discrepancy in publish/poll rates between connected nodes. Furthermore, instead of running the policy directly from the saved PyTorch model file, we pre-compiled the model using TensorRT [23]. TensorRT is NVIDIA's high-performance deep learning inference library that optimizes and accelerates trained neural networks for deployment on GPUs. It supports operations like layer fusion, precision calibration (e.g. FP16/INT8), and dynamic tensor memory management to maximize throughput and minimize latency. All of these allowed us to lower the latency for our system from 35ms to 1.5ms. The major caveat is that the depth image computation from the Zed2i occurs onboard the Jetson and has a latency of 45ms, leading to an end-to-end latency of 50ms. This computation is done with Zed's Neural Depth Engine that uses neural networks to produce higher-fidelity depth maps that have enhanced accuracy in complex environments.

3.7 Correcting Learned Behaviors

During the project, several undesirable behaviors emerged in the learned policy that required correction. First, the policy was highly sensitive to noise in the depth image input with minor variations significantly degrading performance. Second, it exhibited a directional bias, consistently turning in one direction, which impaired its ability to navigate effectively around obstacles when the opposite direction was more favorable. Lastly, the policy produced noisy action commands, leading to rapid oscillations rather than smooth stable trajectories. This section outlines the methods used to address each of these issues.

The first major problem that needed to be overcome was sensitivity to depth image noise. To solve this, as explained in the domain randomization section, we contemplated adding noise to the autoencoder training to help close the sim-to-real gap. While adding in noise such as salt-and-pepper and Gaussian is obvious, we designed the noise to be realistic to the characteristics of the Zed camera we are using. Our first attempt was to scale the noise with

the Zed's published uncertainty curve which is based on distance from the camera. However, this form of noise is not reflective of the true cause of error in stereo depth estimation which is content-based factors such as lighting and scene geometry (e.g. smooth, lots of edges, etc.). As such, while pre-training the autoencoder, we multiplied the per-pixel confidence values from the DIML dataset by Gaussian noise. These confidence images are reflective of the uncertainty that stereo cameras have about edges. This way, the noise added to each pixel during pre-training was scaled by the confidence values. The second measure that helped with this issue was adding CAPS regularization during training of the policy. This regularization helped to drop the Lipschitz constant of the network, thereby creating a policy that was smoother in response to minor noise in the input. These forms of regularization and denoising pre-training helped our policy to be substantially more resistant to both noise and blurring that could show up in the real-world.

The second major problem to overcome was that the policy had a tendency to memorize turning in only one direction—for example, always avoiding to the right. This occurred particularly often in "forest" environments, which consisted of individual pillars scattered at random intervals. This behavior likely emerged because turning in a single direction is an easy-to-learn strategy that often works, as it is generally possible to evade any given pillar from either side. Once the policy learned this behavior, it was very difficult for it to "unlearn" it, since doing so would require a major shift involving multiple intermediate policy variations. In some cases, the policy would occasionally turn in both directions but still showed a strong preference for one side. These behaviors posed a major issue in scenarios such as walls, where the drone needed to make a large turn in a specific direction. In short, the policy consistently fell into the local minimum of turning in only one direction rather than learning the globally optimal strategy of bidirectional avoidance. To address this, we generated environments in which the drone was required to turn in both directions to reach specific goals. The core issue was that the policy began to learn a single-direction turning strategy and then stopped converging, instead of learning bidirectional maneuverability. To counter this, we designed curriculum environments that gradually taught the policy to maneuver in both directions. We then implemented simple curriculum logic that introduced wider wall structures over time, forcing the drone to execute larger avoidance maneuvers. This approach was further supported by increasing the batch size using PyTorch DDP, as described in the previous section, which was critical for enabling convergence in more complex environments. Ultimately, the environment design needed to strike a balance between introducing sufficient variability, promoting bidirectional behavior, and remaining simple enough to allow for stable convergence. Examples of previous versions of the curriculum environments are shown in Figure 3.9 which we experimented on before settling on the configuration in Figure 3.6.

The final challenge was the tendency of the policy to produce rapidly oscillating action commands. These oscillations manifested in several forms, including high-frequency high-amplitude, low-frequency low-amplitude, and low-frequency high-amplitude patterns, as visualized in Figure 3.11. While one might assume this is caused by input noise leading to large output deviations due to a lack of policy smoothness, the issue also appeared consistently in simulation. This was a serious problem for two main reasons. First, rapid oscillations in the real world could impose significant strain on the drone's motors and potentially damage the hardware. Second, such oscillations introduced unpredictable control dynamics that

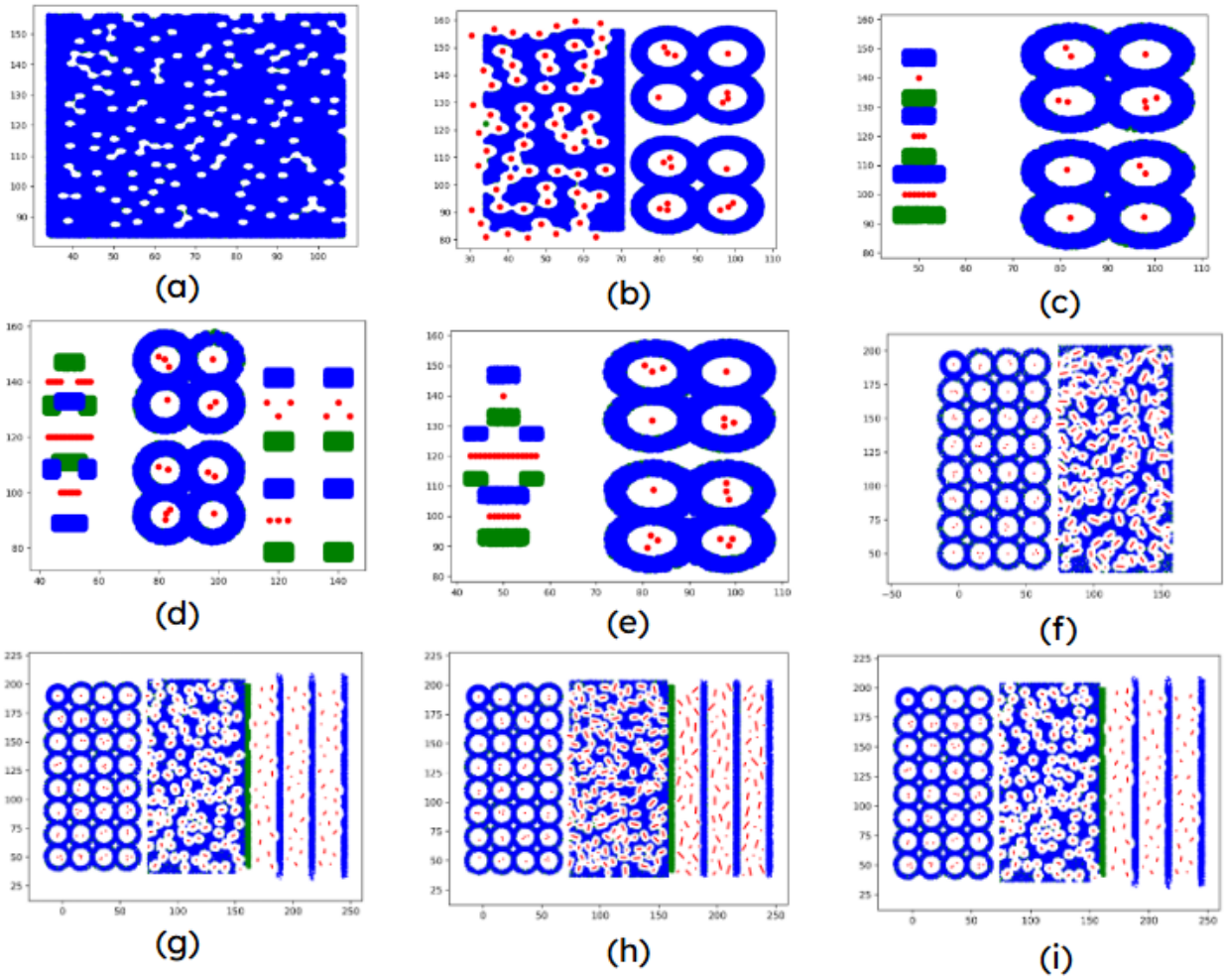

Figure 3.9: 9 pictures depicting different iterations of training environments that were tested before settling on the final configuration depicted in Figure 3.6. Originally it was a simple forest of pillar, however more environmental variations were added in response to learned behavior seen in testing. Meanwhile, tuning ensured that the environments were not too hard for the policy to reach convergence on.

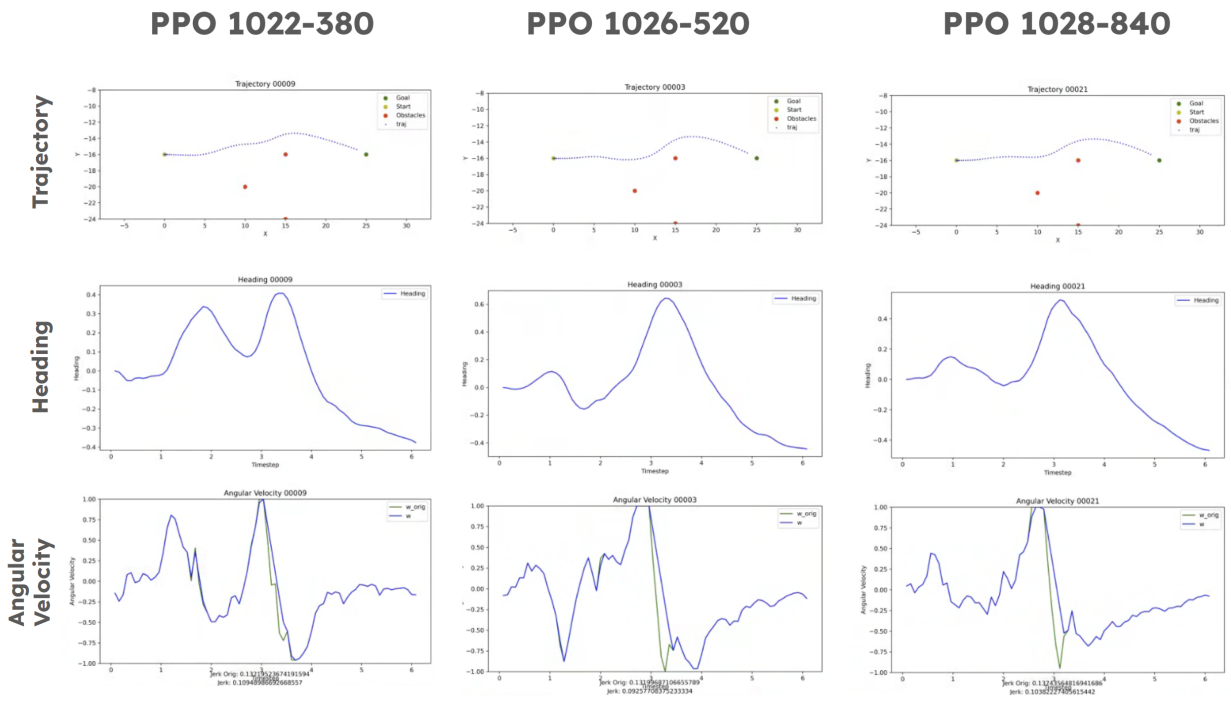

Figure 3.10: A set of trajectories, headings, and angular velocities for three policy variants in the same task setup: 1022-380, 1026-520, and 1028-840. The primary difference between these policies is the weight assigned to the yaw rate penalty in the reward function with 1/100, 1/50, and 1/25 respectively. As can be seen with the angular velocity charts, the 1028 policy is significantly smoother.

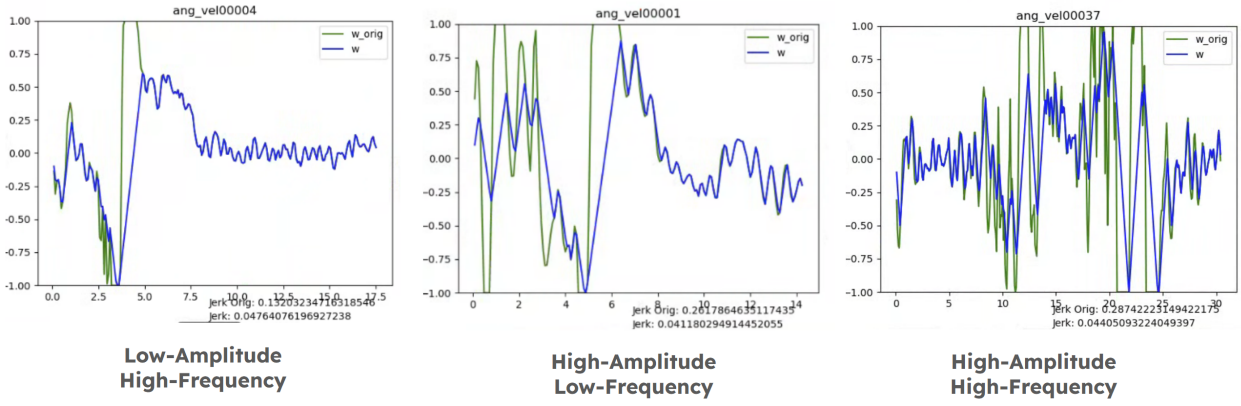

Figure 3.11: A set of angular velocity charts showing oscillating policies in simulation. The blue line is the outputted angular velocity and the green line is the angular velocity after being smooth via an exponential moving average. At the bottom is the average jerk –the derivative of acceleration– of both the original and smoothed outputs.

were not captured by the simplified dynamics modeled in Flightmare, thereby exacerbating the sim-to-real gap. This issue persisted throughout the project, and numerous solutions were attempted. The first approach involved adding several reward terms, such as penalties for yaw rate, angular acceleration, and jerk. We then implemented angular acceleration capping, where the change in commanded angular velocity between consecutive time steps was clipped. Next, we explored smoothing techniques like momentum, where the current angular velocity command was set as an exponential moving average of previous commands. We also experimented with CAPS regularization to directly encourage temporal smoothness by adding a loss term targeting this behavior, thus avoiding indirect supervision through the reward and critic functions. Another idea considered was shifting from direct velocity outputs to predicted future waypoints, which would be converted into a smooth trajectory via B-spline fitting. However, we ultimately avoided this approach because it would prevent the policy from being truly drone-agnostic, as it would no longer output raw velocity commands. Instead, the fix turned out to be deceptively simple: significantly increasing the yaw rate penalty. This penalty, which discourages nonzero angular velocities, was originally set to 1/5000 to promote direct straight line flight to the goal and reduce rotor load. Initial attempts to increase this value beyond 1/1000 resulted in policy non-convergence. However, by substantially increasing the batch size, we were able to raise the yaw rate penalty to values above 1/100 without destabilizing training. As shown in Figure 3.10, we present the position, heading, and angular velocity profiles for three policies in an identical scenario using yaw rate penalty coefficients of 1/100, 1/50, and 1/25. The results indicate that the policy became significantly smoother between 1/100 and 1/50, with diminishing improvements from 1/50 to 1/25.

In this section, we discussed three major issues in training that needed to be addressed. While different strategies were used to tackle each challenge, the key breakthrough that enabled these strategies to succeed was the implementation of PyTorch DDP and a substantial increase in batch size. This is likely because larger batch sizes improve RL convergence by providing more stable and accurate gradient estimates, reducing the variance in policy and value updates. This stability enables more consistent learning signals, particularly in high-dimensional or noisy environments, leading to faster and smoother convergence. As a result, the increased batch sizes allowed the policy to converge across a wider range of learning setups, enabling the successful application of strategies that may not have been effective under smaller batch conditions.

Chapter 4

EXPERIMENTS

This chapter outlines the experiments conducted to validate our design. In Section 4.1, we detail how we evaluated the policy in simulation. In Section 4.2, we give implementation specifics for how we setup the DJI M300 testbed onto which we deployed our policy for real-world testing. In Section 4.3, we describe our real-world experiments and show our results. Lastly, in Section 4.4, we conclude with a discussion of the results and key takeaways.

4.1 Simulation Evaluation

We first conducted a simulation study to analyze the policy's capabilities in a more elaborate environment than can be set up in the real world. To do so, we conducted 4 experiments across varying maximum speeds ($v_{x,max}$) of 1.0m/s - 4.0m/s. In each experiment, we measure the policy's performance over 1000 runs in an obstacle course randomly generated with the same region types as the curriculum environment, thereby capturing a variety of situations. Example trajectories are shown in Figure 4.1. The success rate shown in Table 4.1 stays relatively constant across 1.0 - 3.0m/s, however it drops off at 4.0m/s due to the lower agility and decreased necessary reaction time at higher speeds. Furthermore, the example trajectories in Figure 4.1 demonstrate how the policy is capable of maneuvering bidirectionally and making large turns to avoid wider wall obstacles. These results informed our decision to run the real-world experiments at $v_{x,max}$ =3.0m/s to balance high speed with relative reliability.

Table 4.1: Simulation outcomes across maximum speeds

Speed ($v_{x,max}$)	Success	Crash	Timeout	Out of Bounds
1.0 m/s	992	8	0	0
2.0 m/s	985	13	1	1
3.0 m/s	989	10	1	0
4.0 m/s	967	28	5	2

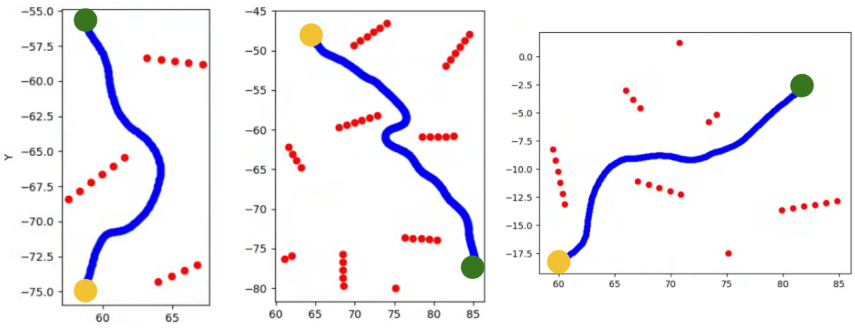

Figure 4.1: Example simulation trials. Yellow is the start and green is the goal. As can be seen in the examples, the drone is capable of making wide turns and complex movements in constrained areas.

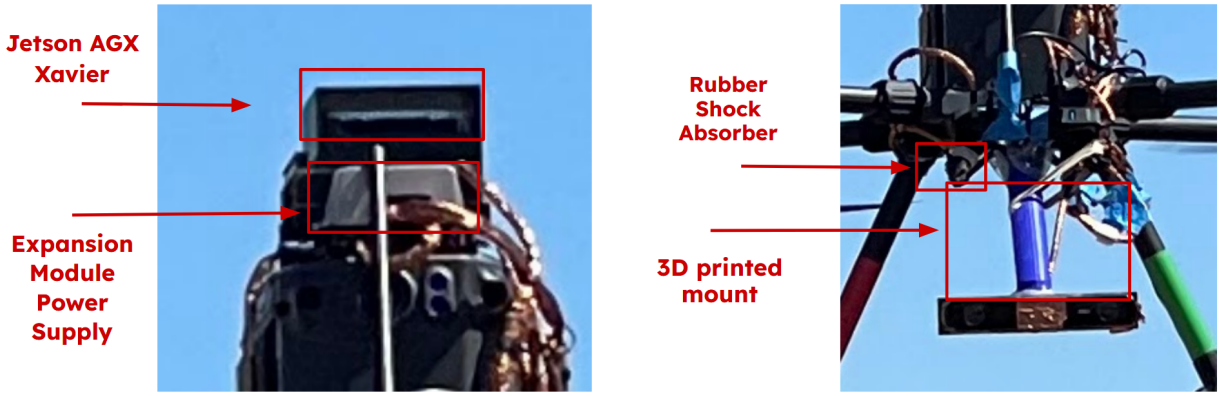

Figure 4.2: The Jetson compute module is mounted on top of the M300 alongside an expansion module power supply that converts power from the drone for use by the Jetson. The 3D-printed camera mount that holds the Zed 2i camera is attached to the M300 drone by a gimbal mount with rubber dampeners.

4.2 Flight System Overview

All real-world tests are conducted on a DJI Matrice 300 RTK (6.3 kg, 0.81 m span) with additional mounted hardware, as shown in Figure 1.1. The DJI M300 is an enterprise-grade drone primarily designed for aerial surveying and inspection, offering a maximum flight time of 55 minutes and support for heavy payloads. The M300 was selected because it was one of the few DJI drones at the time with Onboard Software Development Kit (OSDK) support, which was necessary to programmatically command the drone via ROS. By connecting the Jetson to the drone using a USB-A to USB-C cable, the OSDK can stream telemetry data such as altitude, IMU, and accelerometer readings, while simultaneously accepting control commands such as yaw rate and x/y velocity. In addition, the drone can also publish GNSS-derived position and attitude which we use for post-flight analysis and during GPS mode. Even though, by default, the M300 includes automatic obstacle avoidance, this was disabled for our purposes.

A Zed2i device with a stereo camera and built-in IMU is used to collect 672×376 resolution depth images and state estimates augmented by Visual-Inertial Odometry (VIO). As depicted in Figure 4.2 we mount the camera using a 3D-printed bracket attached to the undercarriage gimbal, which is stabilized with a rubber shock absorber. The mount extends several inches forward to minimize motor occlusion in the camera's field of view. However, this introduces torque on the rubber absorbers, necessitating additional reinforcement to maintain camera stability. The camera operates in neural depth mode, which is a mode provided by Zed that uses a neural network for the disparity calculation, thereby providing improved depth estimation accuracy and robustness to noise. In the depth image, we filter out measurements beyond 20 m (Zed2i's effective range) and below 0.2 m (to exclude rotor blades), before downscaling the resolution to 192×108. This stream is then interfaced with ROS through the Zed SDK, which allows the Zed2i to publish depth images, position, and orientation at 12 Hz.

Stereo depth computation and policy inference are run on a Jetson AGX Xavier, orchestrated using ROS. The NVIDIA Jetson is a series of embedded computing boards designed for AI edge computing, offering GPU-accelerated performance in a compact form factor. It is widely used in robotics, autonomous machines, and real-time AI applications, with models like the Jetson Xavier and Jetson Orin supporting deep learning, computer vision, and sensor fusion workloads. The Jetson is mounted on the M300 and communicates with it via an expansion module as seen in Figure 4.2. The full pipeline—from depth image capture to command transmission—runs at 12 Hz. While the policy generates new commands at 12 Hz, the Jetson publishes commands to the M300 at 80 Hz, reusing the last available command when necessary. This system is designed to be portable and can be deployed on other drones given appropriate mounting hardware and software integration.

Each sensor has its own coordinate frame, so careful conversion must be done to ensure they are all consistent. The policy outputs velocities in the drone's relative FLU frame, where x, y, and z refer to forward, left, and up respectively. The Zed2i is also in the FLU frame. Meanwhile, the DJI OSDK accepts commands in the global NED frame, where x, y, and z refer to north, east, and down respectively. To convert from the policy/Zed2i frame to the M300 frame, we rotate the translational velocities (x, y) by the drone's heading and negate the yaw rate, since the policy's z-axis points upward while the drone's z-axis points downward. This is done with $v_{\text{North}} = \cos(\psi) \cdot v_F, v_{\text{East}} = -\sin(\psi) \cdot v_F$ where v_F is FLU forward velocity and ψ is heading relative to North which is obtained from the drone's internal compass. Additionally, it was found that the number of devices running aboard the drone and the use of multiple connected cables caused significant electromagnetic interference, which prevented takeoff. To mitigate this, all exposed wires were covered with copper foil to reduce signal interference.

4.3 Real-World Experiments

Environments are different obstacle layouts and scenarios are different starting positions for the drone in each environment. To validate that the policy successfully transfers to the

(a)  (b) 

Figure 4.3: Pictures of the actual testing environments at (a) Berkeley Marina and (b) Hearst Mining Circle on the UC Berkeley campus. Note that the obstacles themselves consist of a paper pillar which is mounted on top of a tripod. This was to minimize the damage that might result from a head on collision with the obstacles.

Table 4.2: VIO device positioning trials across environment and starting position. There we 2 trials for 3 environments and 3 positions leading to a total of 18 trials.

Environment	**Scenario**	**Trial 1**			**Trial 2**		
		s [m/s]	t [s]	d [m]	s [m/s]	t [s]	d [m]
Wall	Left	2.53	8.95	2.90	2.63	8.98	2.40
	Center	*2.49*	*9.25*	*3.30*	*2.51*	*9.56*	*2.30*
	Right	2.46	9.33	1.90	2.54	9.00	3.10
Triangle	Left	2.54	8.99	1.05	2.58	9.05	1.85
	Center	*2.61*	*9.26*	*1.90*	*2.59*	*9.23*	*2.10*
	Right	2.54	8.84	2.70	2.60	9.19	2.20
Inverted Triangle	Left	2.63	8.94	2.60	2.71	8.83	2.90
	Center	*2.62*	*9.1*	*1.60*	*2.53*	*9.08*	*2.10*
	Right	2.55	9.28	1.10	2.83	8.07	3.00

Table 4.3: Average measurements across the 2 trials in Table 4.2 for each environment and starting position. The average distance to target measured in the real-world was 2.27m.

Environment	**Scenario**	**Trial Avg.**			**Env Avg.**		**Overall**
		s [m/s]	t [s]	d [m]	Success [%]	d [m]	d [m]
Wall	Left	2.58	8.97	2.65	**100% (6/6)**	2.65	2.27
	Center	2.50	9.41	2.80			
	Right	2.50	9.17	2.50			
Triangle	Left	2.56	9.02	1.40	**100% (6/6)**	1.95	
	Center	2.60	9.25	2.00			
	Right	2.57	9.02	2.45			
Inverted Triangle	Left	2.67	8.89	2.75	**100% (6/6)**	2.21	
	Center	2.575	9.09	1.85			
	Right	2.69	8.68	2.05			

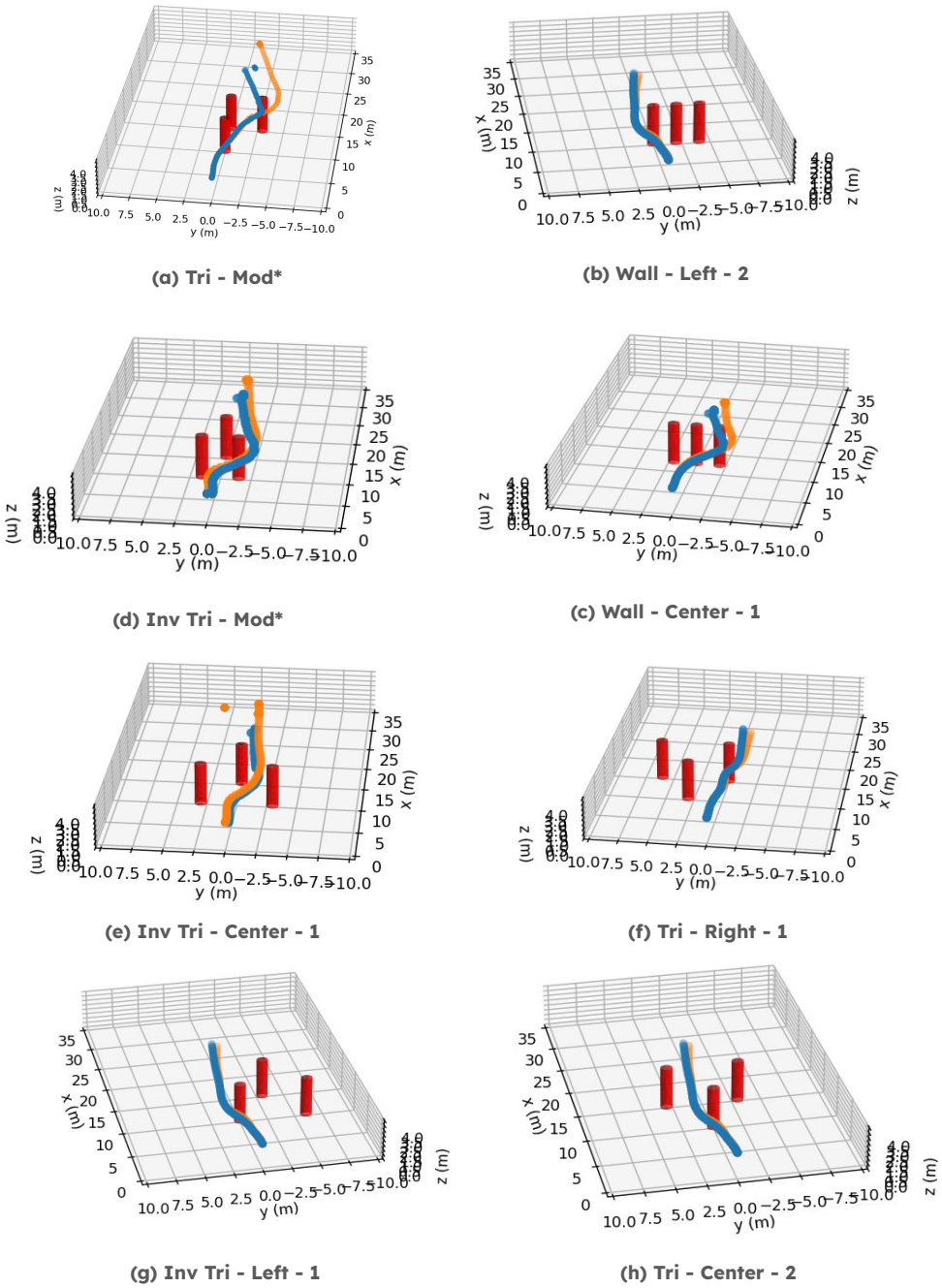

Figure 4.4: 3D visualization of 8 runs, labeled in the form of environment - scenario - trial e.g. (c) is Environment: Wall - Scenario: Center - Trial: 1. Depicted are also modified environments labeled as Tri - Mod and Inv Tri - Mod which are described in Section 4.4. The red pillars are the obstacles, the orange line is the GPS-measured trajectory, and the blue line is VIO device measured trajectory. Note that the blue line may seem like it is passing through obstacles, however that is due to VIO's inaccuracy.

Table 4.4: GNSS-positioning trials across environments. There were 2 trials across 3 environments for a total of 6 trials. Inv Tri stands for Inverted Triangle.

	Trial 1			**Trial 2**			**Trial Avg.**				Overall
	s [m/s]	t [s]	d [m]	s [m/s]	t [s]	d [m]	s [m/s]	t [s]	d [m]	Success [%]	$\bar{d}$ [m]
Wall	2.67	8.26	1.30	2.58	8.45	0.80	2.63	8.36	1.05	100% (2/2)	0.81
Triangle	2.46	8.60	0.52	2.55	8.80	0.78	2.51	8.70	0.65	100% (2/2)	
Inv Tri	2.56	8.34	0.60	2.60	8.46	0.84	2.58	8.40	0.72	100% (2/2)	

real-world, we construct 3 obstacle environments shown in Figure 4.5: wall, triangle, and inverted triangle. Each consists of foam pillars being positioned in different locations with the drone having to navigate through them to reach the goal point. For each environment, we test 3 different scenarios where the start and goal point are in different locations relative to the obstacle formation: left, center, and right. The environments and scenarios are shown in Figure 4.4 and the results are shown in Tables 4.2 and 4.3. The drone has two sources of attitude and positioning information: GNSS data from the M300's onboard receivers and VIO data from the Zed2i. We did two sets of experiments, each set using a different positioning method. For the first set of experiments with VIO device positioning shown in Tables 4.2 and 4.3, we performed 2 trials for each combination of scenario and environment for a total of 18 trials. During the VIO trials, we only use VIO for navigation, however we still collect GNSS data for post-flight analysis as a form of ground-truth positioning. We also performed a second set of experiments with GNSS positioning of 2 trials for each of the environments as shown in Figure 4.7 and Table 4.4. A run is declared to be successful if the drone stops within 1.0m of the goal as declared by the positioning device used by the policy. The device for the VIO trials is the Zed2i and the device for the GNSS trials is GPS. All runs operate at a speed $v_{x,max}$ of 3.0 m/s.

The experiments occurred in two locations shown in Figure 4.3: a field at Berkeley Marina and the Hearst Mining courtyard on the UC Berkeley campus. The field was open with no obstacles in view whereas the courtyard had uneven terrain, a sculpture, a pond, and surrounding buildings/trees. The GNSS experiments in Table 4.4 were conducted in Hearst Mining courtyard in addition to the 6 center scenario trials for the VIO results of Table 4.2, whose measurements are italicized for reference. The remaining 12 VIO trials of Table 4.2 were carried out in Berkeley Marina.

To run the trials, we first positioned the drone hovering just above the ground at the starting location. We then connected to the onboard Jetson by remotely accessing its virtual machine using Rustdesk over a Wi-Fi connection [25]. Next, we initialized the start sequence and input the target goal. For VIO-based positioning, the goal is specified as forward and lateral translations in meters relative to the drone's initial orientation. For example, to move 20 meters forward, we would input y = 20m and x = 0m into the start command. In contrast, GPS-based positioning requires specifying the goal using latitude and longitude coordinates. It is important to note that while the GPS provided excellent relative accuracy, its absolute positioning was less precise. To mitigate this, we hovered the drone over the desired goal location to record the latitude and longitude, then returned it to the starting position—all

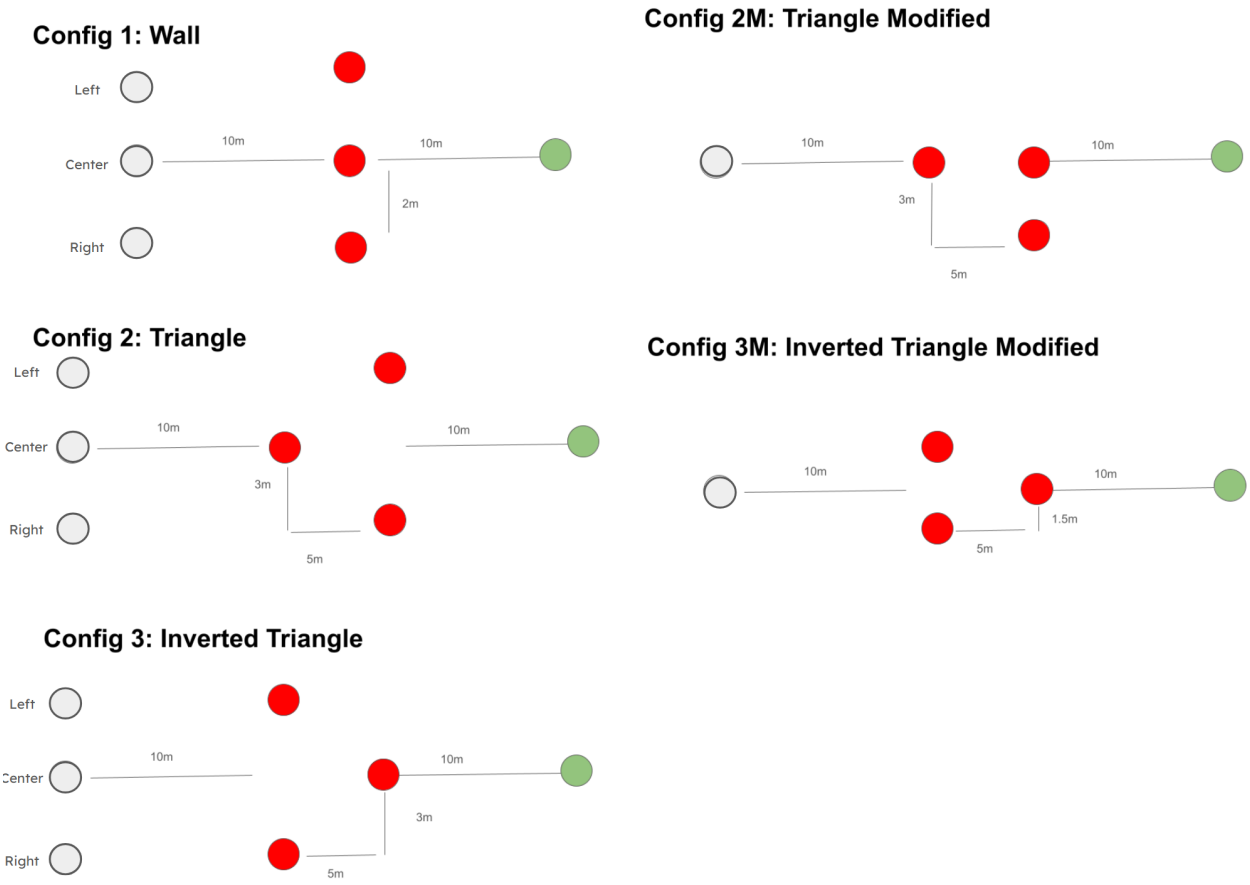

Figure 4.5: Here are the layouts of the three environments: wall, triangle, and inverted triangle. The gray circle is the start, the green circle is the goal, and the red circles are the obstacles. We used a tape measure to accurately position the obstacles and drone in these locations during real-world testing.

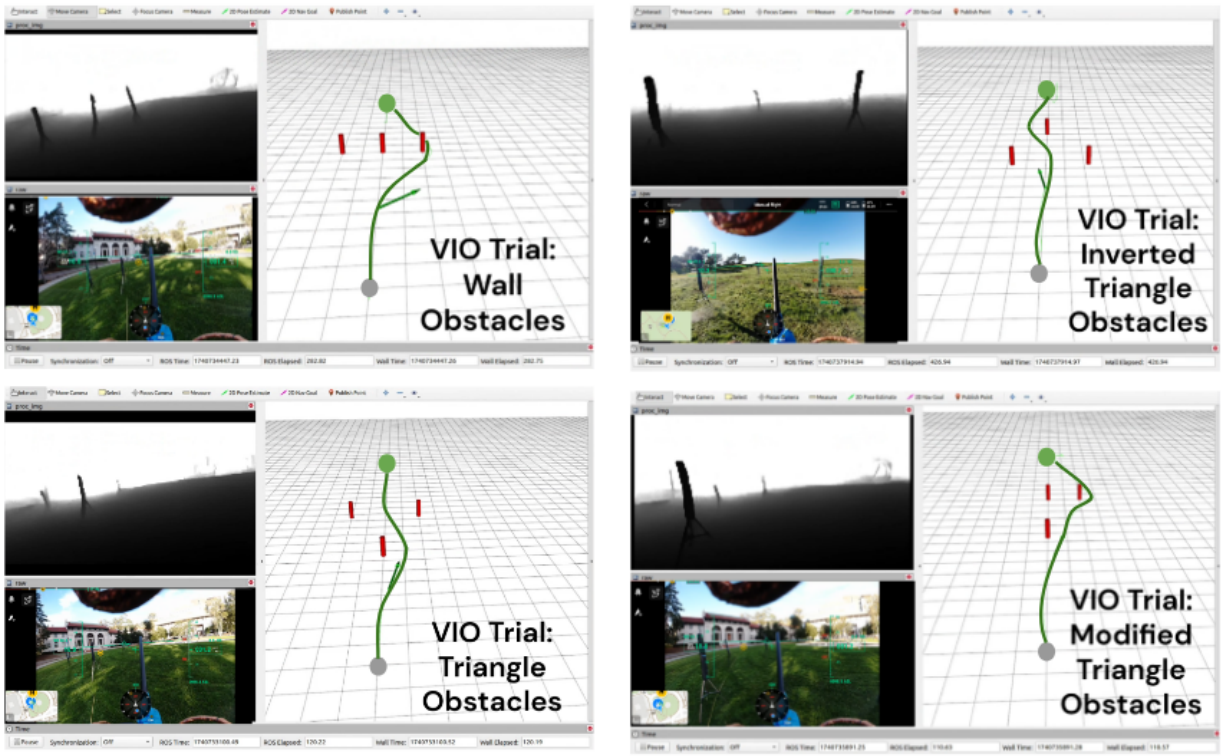

Figure 4.6: Examples of output from 4 real-world VIO test runs. The top left view is the depth image and the bottom left view is video from the drone's onboard FPV camera. The right shows the trajectory and obstacle positions for the run where the gray circle is the start position, the green circle is the goal position, and the green line is the trajectory.

within a single flight. If a different GPS device was used or the drone was landed between recording and execution, the latitude/longitude values could shift slightly, leading to position mismatch during the run.

Beyond the new obstacle configurations themselves, the system had to contend with multiple domain shifts to successfully cross the Sim2Real gap. The most obvious one is that the policy had never trained with the dynamics of a DJI M300 and as such had to compensate for the different flight characteristics. The environment scenery itself was also substantially different from training with the sloped terrain and clutter such as trees, a sculpture, light poles, and buildings. There were out-of-distribution environmental factors such as crosswinds reaching up to 14 kph and non-ideal lighting conditions leading to additional sensor noise. Lastly, the VIO positioning from the Zed2i was oftentimes substantially inaccurate drifting up to 3m as evidenced by GNSS measurements. Examples of actual in-flight depth and RGB images alongside trajectories during four of the runs can be found in Figure 4.6.

In spite of the aforementioned Sim2Real domain gaps, the system achieved a high success rates across all 3 environments. The results can be seen in Figure 4.4 and Table 4.2 where s[m/s] is average speed across the run as measured by GNSS, t[s] is time from start to goal, and d[m] is actual measured distance to goal. As seen in Table 4.2, the success rate (defined as reaching less than 1m to the goal point as measured by the positioning system used by the policy) across all environments and scenarios is 100% while the average distance to goal is 2.27m averaged over all three environments. This relatively large average distance to goal, also shown in the discrepancy between the orange (VIO) and blue (GPS) trajectories in Figure 4.4, are despite the fact that the VIO device believes it is within 1.0 meter of the goal 100% of the time. This reflects the positioning inaccuracy of the chosen VIO device which fundamentally limits the drone's ability to accurately reach the end goal and is a potential area of future research.

The drone had an average speed of 2.58m/s across the trials which is close to the maximum speed limit of 3.0m/s, especially when considering the acceleration phase at the start and deceleration phase near the goal. This indicates that the policy was aiming to achieve the highest possible speed during the trials. To decouple the inaccuracy of the VIO device from the performance of our policy, we conducted trials using GNSS for positioning. The results shown in Figure 4.7 and Table 4.4 indicate a 100% success rate and an average distance to goal of 0.81 which is about 2.8 times smaller than that of VIO device experiments.

4.4 Discussion and Conclusion

Overall, the system is able to perform in largely out-of-distribution conditions. The policy is able to both track a goal point and avoid obstacles in spite of degraded state estimation. It is also able to generalize from the simulation dynamics of a 0.7kg drone to a real-world drone 10 times the weight at a total of 7kg, thereby lending credence to the idea that robustly trained policies can be combined with low-level drone controllers without specific tuning. This is in contrast to [18] which tunes its policy specifically for the dynamics of a custom-built drone. These results can likely be attributed to the autoencoder perception being trained on diverse

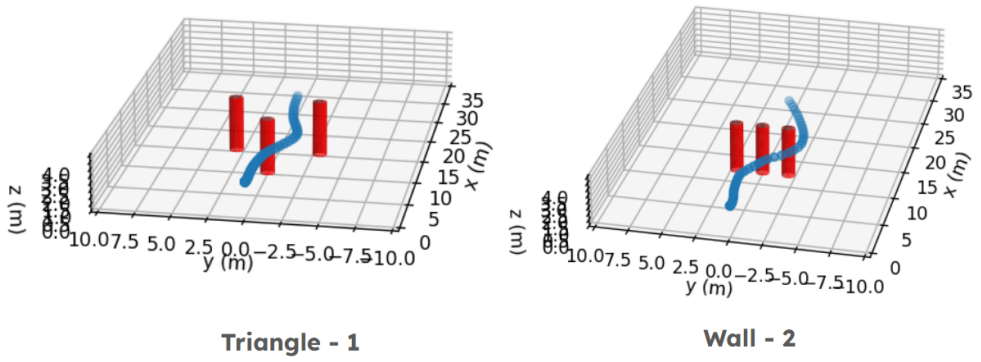

Figure 4.7: 3D visualization of GPS runs. The blue line is GPS-measured trajectory and the red pillars are obstacles.

environments and the use of regularization/randomization to train the policy to be robust to noise across the state-action space.

In addition to the standard trials, we built modified versions of the environments to test behavioral changes in response to certain factors. For instance, as shown in Figure 4.4(a), we created a variation of the Triangle environment where an additional obstacle was concealed behind the first, positioned along the drone's expected trajectory. Upon encountering the hidden obstacle within its field of view, the drone adapted its normal path to avoid it. Similarly, we tested a modified version of the Inverted Triangle environment shown with "Inv Tri-Mod" in Figure 4.4, reducing the gap from 6m to 3m to assess whether the drone would avoid passages with insufficient clearance relative to its size. As expected, the drone avoided the gap.

In terms of transferability to different hardware systems, one of the benefits of a learning-based approach is the extremely low computational cost of the policy itself. Our system (stereo camera input to action command output) has a latency of about 50ms. However, the policy including all surrounding ROS processes is able to complete a pass in only 1.5ms or about >600Hz on the Jetson GPU. The primary bottleneck of our system is the Zed2i's depth image computation process which compares the disparity between the left and right RGB images via a neural network to estimate a depth image. This occurs off-device on the Jetson and takes >90% of the latency i.e. 45ms. However, a depth sensor with on-device depth computation such as the Intel Realsense D435 or Luxonis OAK-D can result in a very fast and compute efficient system. Given the lightweight nature of the on-drone processes and the robustness of the policy, it is likely that the policy can be deployed on smaller drones such as the Modal AI Starling 2 Max or the Holybro X650 with minimal modifications. The most significant changes would involve adapting the on-drone software to interface correctly with the different API structures and control systems specific to each platform.

In conclusion, we successfully deployed a policy that can navigate an off-the-shelf quadrotor through an outdoor obstacle environment to a goal using only onboard compute and vision-based sensors. We employed both reinforcement and privileged learning paradigms where the actor-critic network is first trained using optimal trajectories from a privileged expert as a supervisory backbone before being fine-tuned in a curriculum environment. In real-world

experiments, the policy demonstrates successful zero-shot transfer to novel environments and a drone (DJI M300) that were not seen during training.

Chapter 5

FUTURE WORK

There are several avenues for improving the approach presented in this thesis. One class of improvements involves maintaining the core methodology while making targeted modifications to address performance bottlenecks. The most significant of these is the policy execution frequency, which currently runs at 12Hz. Increasing this frequency would reduce policy response time, enabling higher flight speeds and potentially improving the accuracy of VIO, thereby mitigating the position estimation errors discussed in Section 4.3. Although the system operates at 12Hz, its end-to-end latency is approximately 50ms, suggesting that a policy running at 20Hz could be supported without major performance optimizations. Furthermore, a substantial portion of the current latency arises from the neural depth computation for the ZED2i camera, which accounts for roughly 45ms. To address this, one could train the perception network to be more robust to noise, allowing the use of faster but less accurate depth estimation modes on the Zed2i such as neural light. This would be even further improved by using a camera such as the Intel Realsense D435 or Luxonis OAK-D which can run the depth image computation on the camera itself, eliminating this overhead from the Jetson. These methods would reduce depth image processing time and overall system latency, potentially enabling even higher policy frequencies. Faster policy execution would also support higher flight speeds—up to 4m/s or even 5m/s—as response times improve. However, training policies at higher frequencies poses challenges in simulation: the trajectories become longer due to the increased number of steps per simulated second. As noted in Section 3.6, this leads to greater variability in batch sizes, since entire trajectories must be sampled, which in turn reduces VRAM efficiency. Therefore, methods to stabilize training under these conditions would be essential for training policies that would operate above 20Hz.

The second class of improvements involves enhancing the simulation environment and training regime to produce a more generalizable policy. As discussed in Section 3.3, the current policy is trained in a flat plane with pillar obstacles arranged in various configurations. A natural next step is to develop more complex training environments, ranging from abstract geometric shapes like floating polygons to realistic scenarios inspired by forests, urban landscapes, and other real-world settings. These high-fidelity environments, with their increased geometric complexity, would enable the policy to generalize more effectively beyond

the current setup. A promising approach is to implement curriculum training, where the drone initially trains in a simple pillar-based environment and progressively transitions to more challenging scenarios, such as dense forests. In addition to structural complexity, the simulation could incorporate variable conditions—such as differing weather and lighting—to further enhance robustness and generalization. This expansion in environmental complexity would also necessitate improvements to the simulator itself. As noted in Section 3.3, the current system uses Flightmare, a Unity-based quadrotor simulation platform that becomes CPU-bound and suffers performance degradation when simulating more than 200 agents in parallel. To support scalable, high-throughput training in complex environments, a switch to a GPU-accelerated platform like AerialGym [14], which is built on NVIDIA's Isaac Gym, would likely be required. With these upgrades, it would be possible to train policies capable of navigating diverse obstacle configurations. Consequently, instead of validating in simplified environments with paper pillars, real-world evaluations could more onto using more complex natural obstacles such as trees and other environmental features.

The final category of improvements focuses on advancing the core architecture and capabilities of the neural network policy itself. One promising direction is to transition from depth-based perception to RGB-based perception. A straightforward approach would involve applying a monocular depth estimation method, such as that proposed by [35], to infer relative depth images from RGB inputs, which would then be processed by an image encoder. A more ambitious alternative is to train a perception module end-to-end on RGB inputs directly in simulation. This would require a higher-fidelity simulation environment to ensure that the simulated RGB inputs closely resemble real-world scenes. If successful, this would enable the development of an end-to-end neural policy capable of mapping RGB images to action commands. This would stand in contrast to most other works on autonomous drone flight with a monocular camera which usually use some form of state machine to give the actual action commands. Another avenue is to adopt more sophisticated policy architectures than the current LSTM-based model. For instance, transformer-based sequence models such as GTrXL or ViTs could be used to generate control commands. As discussed in Section 3.2, although we previously opted against these architectures since we determined that they weren't necessary for our objective, their expressivity may become advantageous as task complexity increases. Adjacent to this idea would be to adopt techniques from vision-language navigation and adapt foundation models for use in motion planning such as [2] which uses a large-language model to give action commands as next-token prediction. By fine-tuning a task-specific head off of a pretrained vision-language model backbone such as CLIP, it may be possible to easily get an extremely generalizable policy leveraging the world knowledge of the policy. The primary challenge with this approach lies in the significant computational overhead of such large models, which could substantially increase inference latency and reduce feasible flight speeds.

Bibliography

[1] Anish Bhattacharya, Nishanth Rao, Dhruv Parikh, Pratik Kunapuli, Yuwei Wu, Yuezhan Tao, Nikolai Matni, and Vijay Kumar. Vision transformers for end-to-end vision-based quadrotor obstacle avoidance. *arXiv preprint arXiv:2405.10391*, 2024.

[2] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

[3] Nathan Bucki, Junseok Lee, and Mark W. Mueller. Rectangular pyramid partitioning using integrated depth sensors (rappids): A fast planner for multicopter navigation. *IEEE Robotics and Automation Letters*, 5(3):4626–4633, 2020.

[4] Aditya M. Deshpande, Ali A. Minai, and Manish Kumar. Robust deep reinforcement learning for quadcopter control. *arXiv preprint arXiv:2111.03915*, 2021.

[5] Alessandro Devo, Jeffrey Mao, Gabriele Costante, and Giuseppe Loianno. Autonomous single-image drone exploration with deep reinforcement learning and mixed reality. *IEEE Robotics and Automation Letters*, 7(2):5031–5038, 2022.

[6] Robin Ferede, Christophe De Wagter, Dario Izzo, and Guido C. H. E. de Croon. End-to-end reinforcement learning for time-optimal quadcopter flight. *arXiv preprint arXiv:2311.16948*, 2023.

[7] Jiawei Fu, Yunlong Song, Yan Wu, Fisher Yu, and Davide Scaramuzza. Learning deep sensorimotor policies for vision-based autonomous drone racing. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5243–5250. IEEE, 2023.

[8] Sven Gronauer, Daniel Stümke, and Klaus Diepold. Comparing quadrotor control policies for zero-shot reinforcement learning under uncertainty and partial observability. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7508–7514. IEEE, 2023.

[9] Drew Hanover, Antonio Loquercio, Leonard Bauersfeld, Angel Romero, Robert Penicka, and et al. Autonomous drone racing: A survey. *IEEE Transactions on Robotics*, 2024.

[10] Jialin Ji, Zhepei Wang, Yingjian Wang, Chao Xu, and Fei Gao. Mapless-Planner: A robust and fast planning framework for aggressive autonomous flight without map fusion. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11720–11726, 2021.

[11] Katie Kang, Suneel Belkhale, Gregory Kahn, Pieter Abbeel, and Sergey Levine. Generalization through simulation: Integrating simulated and real data into deep reinforcement learning for vision-based autonomous flight. In *2019 international conference on robotics and automation (ICRA)*, pages 6008–6014. IEEE, 2019.

[12] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987, 2023.

[13] Mihir Kulkarni and Kostas Alexis. Reinforcement learning for collision-free flight exploiting deep collision encoding. *arXiv preprint arXiv:2402.03947*, 2024.

[14] Mihir Kulkarni, Theodor J. L. Forgaard, and Kostas Alexis. Aerial gym – isaac gym simulator for aerial robots, 2023.

[15] Jonathan Lee, Abhishek Rathod, Kshitij Goel, John Stecklein, and Wennie Tabib. Rapid quadrotor navigation in diverse environments using an onboard depth camera. In *2024 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, November 2024.

[16] Sikang Liu, Kartik Mohta, Nikolay Atanasov, and Vijay Kumar. Search-based motion planning for aggressive flight in se (3). *IEEE Robotics and Automation Letters*, 3(3):2439–2446, 2018.

[17] Xu Liu, Guilherme V. Nardari, Fernando Cladera Ojeda, Yuezhan Tao, Alex Zhou, Thomas Donnelly, Chao Qu, Steven W. Chen, Roseli A. F. Romero, Camillo J. Taylor, and Vijay Kumar. Large-scale autonomous flight with real-time semantic slam under dense forest canopy. *IEEE Robotics and Automation Letters*, 7(2):5512–5519, 2022.

[18] Antonio Loquercio, Elia Kaufmann, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Learning high-speed flight in the wild. *Science Robotics*, 6(59):eabg5810, 2021.

[19] Antonio Loquercio, Ana I. Maqueda, Carlos R. Del Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3:1088–1095, 2018.

[20] Kimberly McGuire, Guido de Croon, Christophe De Wagter, Karl Tuyls, and Hilbert Kappen. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters*, 2(2):1070–1076, 2017.

[21] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation*, pages 2520–2525. IEEE, 2011.

[22] Siddharth Mysore, Bassel Mabsout, Renato Mancuso, and Kate Saenko. Regularizing action policies for smooth control with reinforcement learning. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1810–1816. IEEE, 2021.

[23] NVIDIA Corporation. *TensorRT Developer Guide*. NVIDIA, 2023.

[24] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International conference on machine learning*, pages 7487–7498. PMLR, 2020.

[25] RustDesk Contributors. Rustdesk: Open-source remote desktop. `https://rustdesk.com`, 2025. Accessed: 2025-05-20.

[26] Varun Saran and Avideh Zakhor. Vision-based deep reinforcement learning for autonomous drone flight. Master's thesis, EECS Department, University of California, Berkeley, Dec 2023.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] Abhik Singla, Sindhu Padakandla, and Shalabh Bhatnagar. Memory-based deep reinforcement learning for obstacle avoidance in uav with limited environment knowledge. *IEEE transactions on intelligent transportation systems*, 22(1):107–118, 2019.

[29] Yunlong Song, Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, and Davide. Scaramuzza. Icra 2022 dodgedrone challenge: Vision-based agile drone flight. 2022.

[30] Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, and Davide Scaramuzza. Flightmare: A flexible quadrotor simulator. In *Conference on Robot Learning*, pages 1147–1157. PMLR, 2021.

[31] Yunlong Song, Kexin Shi, Robert Penicka, and Davide Scaramuzza. Learning perception-aware agile flight in cluttered environments. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1989–1995. IEEE, 2023.

[32] Vaibhav Viswanathan, Eric Dexheimer, Guanrui Li, Giuseppe Loianno, Michael Kaess, and Sebastian Scherer. Efficient trajectory library filtering for quadrotor flight in unknown environments. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2510–2517, 2020.

[33] Chao Wang, Jian Wang, Yuan Shen, and Xudong Zhang. Autonomous navigation of uavs in large-scale complex environments: A deep reinforcement learning approach. *IEEE Transactions on Vehicular Technology*, 68(3):2124–2136, 2019.

[34] Wei Xiao, Zhaohan Feng, Ziyu Zhou, Jian Sun, Gang Wang, and Jie Chen. Time-optimal flight in cluttered environments via safe reinforcement learning. *arXiv preprint arXiv:2406.19646*, 2024.

[35] Xin Yang, Jingyu Chen, Yuanjie Dang, Hongcheng Luo, Yuesheng Tang, Chunyuan Liao, Peng Chen, and Kwang-Ting Cheng. Fast depth prediction and obstacle avoidance on a monocular drone using probabilistic convolutional neural network. *IEEE Transactions on Intelligent Transportation Systems*, 22(1):156–167, 2019.

[36] Dingqi Zhang, Antonio Loquercio, Jerry Tang, Ting-Hao Wang, Jitendra Malik, and Mark W. Mueller. A learning-based quadcopter controller with extreme adaptation, 2024.

[37] Haokun Zheng, Sidhant Rajadnya, and Avideh Zakhor. Monocular depth estimation for drone obstacle avoidance in indoor environments. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10027–10034. IEEE, 2024.