

Vision-Based Deep Reinforcement Learning for Autonomous Drone Flight

*Varun Saran
Avideh Zakhor
Antonio Loquercio, Ed.*



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-280

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-280.html>

December 15, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to express deep gratitude, respect, and appreciation for Andrew Sue, who worked with me on this project from the beginning. This thesis would not have been possible without his hard work and commitment.

I would also like to thank my advisor, Professor Avidah Zakhor, for her guidance and support during this research journey.

Last but not least, I am forever grateful to my parents, Niraj and Ruchi Saran, and my brother, Vedant Saran, for their constant unwavering support toward me.

Vision-Based Deep Reinforcement Learning for Autonomous Drone Flight

by

Varun Saran

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Avideh Zakhor, Chair
Antonio Loquercio

Fall 2023

Vision-Based Deep Reinforcement Learning For Autonomous Drone Flight

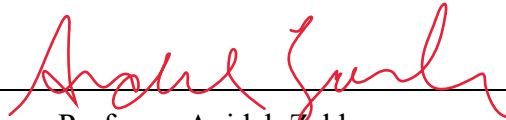
by Varun Saran

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.**

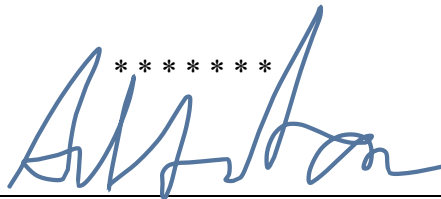
Approval for the Report and Comprehensive Examination:

Committee:



Professor Avidesh Zakhor
Research Advisor

12/13/23
(Date)



Antonio Loquercio, PhD
Second Reader

12/14/23
(Date)

Vision-Based Deep Reinforcement Learning for Autonomous Drone Flight

Copyright 2023
by
Varun Saran

Abstract

Vision-Based Deep Reinforcement Learning for Autonomous Drone Flight

by

Varun Saran

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Avideh Zakhor, Chair

Autonomous drone flight has emerged as a revolutionary technology with diverse applications across industries such as search and rescue, infrastructure inspection, deliveries, defense, and precision agriculture. Drones are packed with various sensor suites and are tasked to perceive their surroundings, navigate to goal locations, and detect points of interest, all while dealing with complex, unknown environments. Classical approaches separate the perception, planning, and control steps. Other works output trajectories for a Model Predictive Controller to follow.

In this work, we present a deep Reinforcement Learning (RL) approach for an off-the-shelf drone to fly through goal positions and avoid obstacles in unknown outdoor environments. The agent is given position, orientation, yaw rate, and depth image information. Given this, it outputs linear velocities and a yaw rate. Privileged learning is used during training, where full environment information is used beforehand to pre-compute optimal trajectories. Optimal trajectories provide a supervisory signal to the RL agent, which is penalized for deviating from the given trajectory. Our work combines the benefits of pre-computed optimal trajectories with the advantages of exploration with an RL agent, allowing for flight in previously unseen situations.

This policy transfers well to new hardware platforms with different dynamics, as many off-the-shelf platforms come with lower level velocity controllers. Real world experiments show positive results on a DJI Matrice 300 – a different hardware platform from the one used in simulation – in a simple outdoor environment, where the policy is applied zero-shot and is able to avoid an obstacle and navigate to desired goals at 1m/s.

Contents

Contents	i
List of Figures	iii
List of Tables	vii
1 Introduction	1
1.1 Motivation	1
1.2 Related Work	3
2 Methods	5
2.1 General Overview	5
2.2 Flightmare - Quadrotor Simulator and Rendering Engine	7
2.3 Trajectory Generation	8
2.4 Control Scheme	10
2.5 Vision-Based RL	11
2.6 Sim 2 Real	22
2.7 Physical Platform	25
3 Evaluation and Experiments	30
3.1 Simulation Results	30
3.2 Real-World Tests	32
3.3 Further Analysis	41
4 Conclusion	48
4.1 Discussion	48
4.2 Future Work	49
Bibliography	52

5 Appendix A: Additional Figures

List of Figures

2.1	The full training and deployment architecture defined in this work. Training happens in simulation, with privileged information of the environment. Given a pointcloud, optimal reference trajectories are created and used in the reward function calculation. During deployment, a DJI Matrice 300 quadrotor is used, with a ZED2i camera for sensing and visual odometry. The output velocities are tracked by Flightmare in simulation, and by DJI’s velocity controller when running on hardware.	6
2.2	The drone flying in a forest environment, running in Unity. On the left are the RGB and depth images taken by the drone in the current frame. This shows the hyper-realistic nature of the rendering engine. Fig. 2.3 is the simpler environment used in this work.	8
2.3	The simple environment of obstacles used during training. Left, cylinders were randomly generated at various positions with slightly different height and radius, and are meant to emulate tree trunks in a forest. Right, the drone flying through the environment, with an obstacle in front of it. . .	9
2.4	Reference trajectories through the pointcloud of the environment. The environment is mostly flat, and is filled with numerous cylindrical obstacles shown in blue and topped in purple. The black lines represent unique trajectories, where each one has its own 3D start and goal point marked in red, and their paths safely go around the cylindrical obstacles.	10
2.5	The outputs of the policy. We use a velocity controller, so policy outputs are a desired forward velocity, an upward velocity, and a yaw rate. The forward velocity is in the body frame, so in the same direction the camera is pointing.	11

2.6	The Flightmare dynamics modeling engine communicating with the Unity rendering engine. The dynamics modeling calculates the new state of the quadrotor, given its current state, velocity command, and Δt . This new state is passed to Unity, which updates the drone's position in the running game engine, and then captures a depth image from this new state. The state and depth image are passed to the RL network, which outputs a velocity for Flightmare to enact, repeating the cycle.	12
2.7	The inputs and outputs of the RL policy. The most recent depth image from the drone goes through a custom feature extractor where it is featurized and then pooled into a 1D vector. State and goal information is normalized and mixed via linear layers, and then concatenated with the image representation, to be inputted to a 2-layer MLP. The policy outputs 3 numbers, representing x, z body frame linear velocity in m/s , and a yaw-rate in rad/s	14
2.8	The architecture of the custom feature extractor. The full 1D state is separated into the 7 state values and 192×108 image pixel values. Both are independently processed into 1D feature vectors of size 64 and 10, respectively. They are then concatenated and passed through fully connected layers for more state mixing. The final vector of size 256 is passed as input to the RL actor and critic. This light-weight architecture runs at 10Hz on the physical drone platform.	20
2.9	Comparing the depth image with different amounts of processing. (a) is the raw noisy unfiltered image, (b) is the same image with ZED2i's built-in neural and fill modes. (c) is the mask used to get rid of noise from the propellers. (d) is the masked downsampled image that is passed to the feature extractor. The scale of pixel values in the final image is very large, so the ground loses its humanly visible gradient, even though the actual values are correctly varied.	24
2.10	The physical platform, including a DJI Matrice 300 drone, ZED2i camera, and Jetson Xavier AGX.	25
2.11	Propellers block the camera's field of view, resulting in noisy depth images. Left, an RGB image from the camera showing propellers jutting out from both sides of the image. Right, the depth image showing noise values where the propellers are rapidly spinning. This section is masked out before being passed to the policy.	27
2.12	The ZED2i camera is mounted slightly below and in front of the drone's center.	28

2.13	The global NED and local FLU coordinate systems. The NED frame's axes point North, East, and Down, while FLU points locally Forward, Left, and Up. The heading angle θ determines the rotation between the two coordinate systems, and is measured by the drone's compass.	29
3.1	The reward per drone per iteration, averaged over a 25 drone training environment. It converges in roughly 300 iterations.	31
3.2	Midway through the training process, the policy learned to stay close to the goal without actually reaching the terminal state, resulting in it circling the goal. Reward tuning is important to encourage the drone to go directly to the goal as soon as possible.	31
3.3	Top down view of example test trajectories. Starting at the initial point (yellow), the drone reaches within 0.2m of the goal point (green), and avoids obstacles (red). All units are in meters.	32
3.4	The Cesar Chavez park in Berkeley, CA, where physical flight tests were conducted. The black cylindrical object is the "obstacle" used in our tests, constructed using pool noodles.	33
3.5	Policy outputs visualized. Top, the drone is directly in front of the red obstacle, so the policy directs the drone to fly left and around the obstacle. Bottom, the drone has passed the obstacle, so the policy outputs a velocity to head back toward the green goal. The trajectory is formed by the ZED2i's real-time position estimation.	36
3.6	Comparing real-world footage to the real-time visualization. On the left is the RViz visualization with the latest image and drone position according to the ZED2i. On the right is the real world video, showing the drone to the left of the obstacle. The full video can be found here , at 2x speed. . .	37
3.7	The full trajectory of a real-world flight test based on the ZED2i's position estimation, visualized in RViz.	38
3.8	Some of the successful trajectories where the drone flies around an obstacle and reaches the desired goal.	38
3.9	A physical test where the goal was set behind the drone. The drone did a u-turn and then went toward the goal, highlighting the policy's ability to adapt to trajectories that were not seen in the training set.	39
3.10	The failed trajectories where the drone flew too close to an obstacle. Reasons for failure include high gusts of wind limiting the drone's desired movement, and a lack of temporal and spatial memory in the policy. . .	40

3.11	Comparing policy outputs versus the trajectory flown in simulation. Left, the theoretical trajectory assuming all policy output velocities are perfectly and immediately tracked. Right, the path flown from start (yellow) to goal (green), while avoiding the obstacle (red). Both trajectories are extremely closely aligned, as is expected from an ideal simulation environment.	42
3.12	Comparing policy outputs versus the GPS trajectory flown in the real world. Left, the theoretical trajectory assuming all output velocities are perfectly and immediately tracked. Middle, the path flown according to GPS data. The first right turn comes later in the GPS trajectory than the policy outputs, showing a delay between when outputs are sent and when they are applied. Right, an attempt to align the two trajectories to show the controller achieves reasonable performance at matching corresponding turns. Errors come from the controller's inability to perfectly track desired velocities. Additionally, since velocities are naively integrated to form the theoretical trajectory, errors propagate. These reasons lead to the two paths not being as well aligned as in simulation.	43
3.13	Comparing GPS data (left) against the ZED2i's visual odometry based state estimation (right) to evaluate the stereo camera's performance. Overall, trajectories are similar, but visual odometry suffers from some noise. Notably, a jump to the right is seen initially when the drone starts flying forward. This jump is quickly corrected for with an opposite leftward jump briefly after the drone flies forward.	44
3.14	Comparing heading angles over time between the ZED2i (green) and the drone's IMU (blue). Heading from the ZED2i aligns well with the drone's IMU.	45
5.1	The ports on the DJI Expansion Module. Use a USB-TTL cable to connect the Jetson to the expansion board.	55

List of Tables

2.1	The state space of the agent. It contains 7 robot states – the positional error defined as the difference between the goal position and the current position in x , y , and z , the absolute roll angle of the drone, the absolute pitch of the drone, the difference between the theoretical heading angle creating a straight line to the goal and the actual current heading, and the current yaw rate of the drone – and 192×108 pixel values from the depth image. For the states, positional error and heading error are used instead of separately defining the current position/heading and goal position/heading, to help generalize over various situations.	16
2.2	The reward terms of a given state, as a function of the initial distance to the goal d_0 , the current distance from the goal d , the heading error θ_e , the distance to the closest point on the reference trajectory d_{traj} , the closest object in the center of the image $d_{obstacle}$, the velocity v_g along the axis pointing directly to the goal, the heading rate ω , and error in z position e_z . Reward shaping was used to guide the RL agent toward effective behavior, and substantial trial and error over numerous trained models led to the final weights.	18
2.3	Terminal state rewards given for reaching positive or negative terminal states. These have much higher magnitude than per time step rewards because they represent the performance of the policy much better than any random state can. Like before, d is the distance to the goal.	19
3.1	Results from real world flight tests. The model was trained at 1m/s, with 0 or 1 obstacles in each trajectory. These situations performed very well. Increasing the number of obstacles to 3, or speed to 1.5m/s resulted in bad performance where a human pilot had to take over control. Starting at the wrong initial heading also worked, showing the policy’s learned knowledge of path planning in cases where the goal may be in any direction.	35

3.2	Training models flying at 1m/s and 3m/s, and evaluating them at both 1m/s and 3m/s to compare performances, all in simulation. Results show good performance is achieved when evaluating at the same or lower speed as training, and poor results stem from training at high speeds but evaluating at lower speeds. A sample size of 5,000 was used when evaluating all models.	46
3.3	Results at different flight speeds and policy frequencies in simulation. A model was optimized to fly at 1m/s at 10Hz in simulation, reaching the goal every time. A new policy was then trained at 20Hz, which learned to avoid crashes but not reach the goal. Doubling the flight speed to 2m/s resulted in reasonable performance. This pattern shows that simply increasing the frequency may not improve results. We hypothesize this to be because at low speeds and very small timesteps, all actions have a similar effect – only after multiple timesteps does it become clear that an agent is turning left rather than going straight. Increasing the speed to 2m/s countered this increase in frequency, so each action again has more distinct effects. Again, maximum speeds were increased by scaling policy outputs by the desired maximum speed. Note that reward function and hyperparameter tuning was only done for 1m/s and 10Hz, so more tuning should be done with the other speeds and frequencies.	47

Acknowledgments

I would like to express deep gratitude, respect, and appreciation for Andrew Sue, who worked together with me on this project from the beginning. This thesis would not have been possible without his hard work and commitment.

I am also thankful for Kehan Li, who helped analyze sensor and policy performance, and conduct real real-world tests.

I would like to thank my advisor, Professor Avidah Zakhor for her continued guidance and support during this research journey. I am also grateful to Antonio Loquercio, one of the main contributors to `agile_flight`, the core flight simulator used in this work. He provided valuable insights and timely advice to overcome numerous obstacles throughout the project. He is also the second reader for this thesis.

Last but not least, I am forever grateful to my parents, Niraj and Ruchi Saran, and my brother, Vedant Saran, for their constant unwavering support toward me.

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been significant development in autonomous capabilities for drones. The ability of drones to operate autonomously has the potential to revolutionize various industries, such as search and rescue, infrastructure inspection, and precision agriculture. However, achieving true autonomy with drones presents numerous challenges, including perception, planning, and control in dynamic and unknown environments. In this work, we propose an approach to address these challenges and develop a fully autonomous drone system capable of traversing a simple unknown real-world environments, with one obstacle.

There are many high-level path planners that can plan global paths across long distances using waypoints. In [1], they optimally traverse a given area while ensuring all points of interest are visited. [2] and [3] ran a modified A-star search to find a path to a goal. These could be used for infrastructure inspection, agriculture, and even aerial deliveries. However, in both cases, the path planners output intermediary positions as goals, without defining how to actually get the robot to those states, and without guaranteeing collision-free paths. For this, a lower-level planner is needed to determine how to follow the given path while avoiding potential obstacles along the way.

Ground robots may use optimization based planners from [4] and [5] or Rapidly Exploring Random Trees (RRT) like in [6], but both types of planners require full knowledge of the environment, which is not the case for many real-world applications. Additionally, optimization solvers for 3D flight must handle an exponentially larger search space due to the higher dimensional state and input spaces, so are infeasible to run in real-time. Empirically, they can take on the order of hours to run, after

various simplifications are made to reduce the search space.

Classical approaches for safe aerial flight include a step-by-step process that separates mapping and planning. For example, [7] and [8] first perform Simultaneous Localization and Mapping (SLAM) to self localize and build a map of the world, after which [7] uses RRT to path-plan around the environment.

This makes it easier from an engineering perspective, as each problem, i.e. SLAM and RRT, is well defined and can be worked on in parallel. However, it results in high latency due to its serial nature, and error propagation as future steps cannot account for errors from past steps. With the desire for robust and responsive flight, such approaches become unreliable. More recent approaches such as [9] and [10] include neural models that take in images and state information and directly output control commands, without separating the perception and planning aspects. However, they output thrusts and body rates which do not transfer zero-shot to new platforms.

There is a rise in off-the-shelf drones with various payload capabilities. With such a wide range of drone use-cases, we wish to freely switch between drone platforms based on the payload needed, without re-engineering the entire controller. For example, if we add a LiDAR sensor to a platform and only have a controller that outputs thrust and body-rates, the controller cannot be used zero-shot for this new platform with different dynamics. It can only be used on a platform with the same dynamics used during training, so it does not transfer to platforms of different size and mass. [11] outputs desired trajectories for the drone to follow, but a Model Predictive Controller (MPC) is necessary to track those trajectories, and MPC must be tuned separately for different hardware platforms. In this work, we aim for a single controller, that can work as-is without additional engineering, on various drone platforms of different size and mass, such as an AR Parrot, any drone platform running the ArduPilot software suite, a DJI Matrice 300, and even a DJI Matrice 600, a hexacopter. This approach allows for easy deployment on different platforms.

While some platforms have different dynamics and cannot track aggressive trajectories as well as others, for most applications we do not need a high speed, highly agile platform capable of performing aggressive maneuvers. Speed is important in racing, but for search and rescue missions, environment mapping, drone delivery, etc., we mainly need a controller to get the drone where it needs to go, without crashing. This makes a velocity controller a good choice because most off-the-shelf drones, such as the ones listed above, have built-in APIs to track desired velocities.

1.2 Related Work

In this work, we use the Flightmare [12] quadrotor simulator from ETH Zurich to simulate quadrotor flight. Many prior works have used Flightmare, as it provides a flexible platform with accurate dynamics modeling, communication with a photorealistic rendering engine, depth image rasterization, and capabilities to add custom environments. Additionally, it provides a base framework for an Actor-Critic method for state-based reinforcement learning.

Inspiration for this work was taken from [11], which uses the idea of privileged learning to learn from ideal reference trajectories in simulation. During their training process, the full environment is known, and optimal reference trajectories can be computed. Given a depth image, drone velocity and attitude, and a desired heading direction, a network is then trained to output the reference trajectories. This work achieved state-of-the-art results and was able to fly at high speeds in dense real-world environments such as a forest, showing the power of privileged learning and reference trajectories. However, since this work outputs trajectories as future positions over time, it requires a finely tuned MPC controller to follow the desired trajectory. So while it works well on their highly customized drone platform, it does not transfer well to off-the-shelf drone platforms from DJI or Parrot which do not have publicly released MPC controllers for their platforms.

In [13], they train a Deep RL policy to output motor thrusts to follow desired waypoints at extremely high speeds. Their application is racing, and each waypoint represents the center of a gate the drone is meant to fly through. Their policy inputs include the drone state and gate observations, where gate observations are mathematically calculated using pre-defined gate positions defining the race track. They successfully achieve high speed flight with minimal collisions, but they require a known environment with minimal obstacles of known and simple geometry. Collisions can be tested algebraically by comparing the drone’s position against the known position of the gate, the drone’s radius, and the gate’s shape. In this work, we build off of [13] in that we also train our agent using the Proximal Policy Optimization (PPO) algorithm, but we deal with unknown environments rather than known ones. To handle such situations, we include a depth image as an input to the policy, and must compute mesh collisions at every timestep for every agent, to deal with geometrically complex obstacles of unknown shape. In this way, collision information can be used as part of the reward function.

The very recent work of [9] also combined RL with imitation learning. They train a privileged state-based RL agent to output thrusts and body rates, and then distill the policy via imitation learning onto a vision-based model. We combine their serial imitation and reinforcement learning steps into one single process, and output

velocities rather than body rates.

In summary, we aim to train a deep RL policy that outputs velocity controls to avoid previously unknown obstacles and reach a desired goal position.

The outline of this thesis is as follows. In Chapter [2](#), we go in-depth into the methods followed in this approach. In Chapter [3](#), we summarise the results from experiments both in simulation and in the real-world. Finally, in Chapter [4.1](#), we discuss the implications of this work, its strengths and limitations, and discuss some future work to expand on this approach.

Chapter 2

Methods

In this Chapter, we describe in full detail the methods used in this approach. The outline is as follows. Section [2.1](#) is a brief overview of the entire approach – the architecture, training procedure, hardware platform, and how the approach can be used for short or long-distance flights. Section [2.2](#) describes the simulator and rendering engine, section [2.3](#) describes how optimal reference trajectories are generated, and section [2.4](#) describes the restricted velocity controller used. Section [2.5](#) goes in-depth into the goal of Proximal Policy Optimization (PPO), the full state and action spaces, the reward function, and feature extraction. Section [2.6](#) describes our attempt to bridge the gap between simulation and the real-world, and section [2.10](#) describes the drone system and mounted sensors on our real-world platform.

2.1 General Overview

The full pipeline of our proposed approach is summarized in Fig. [2.1](#), including both training in simulation with privileged information, as well as deployment on a physical DJI Matrice 300 drone. Training includes simulating a drone in the Flightmare simulator, calculating 10-20 meter reference trajectories offline from thousands of start to goal points in 3D space, and using this information to train an RL agent. The agent is given an input including a depth image and state information. State information includes the drone’s position, orientation, and yaw rate. The image is passed to a feature extractor and the states are normalized, before being passed to linear layers. The policy finally generates 3 outputs – a forward velocity, a vertical velocity, and a yaw rate. These outputs are passed to the drone controller, which accordingly updates the drone’s state after a very short interval of time. The updated state is used to calculate the reward for the action taken, using factors such as

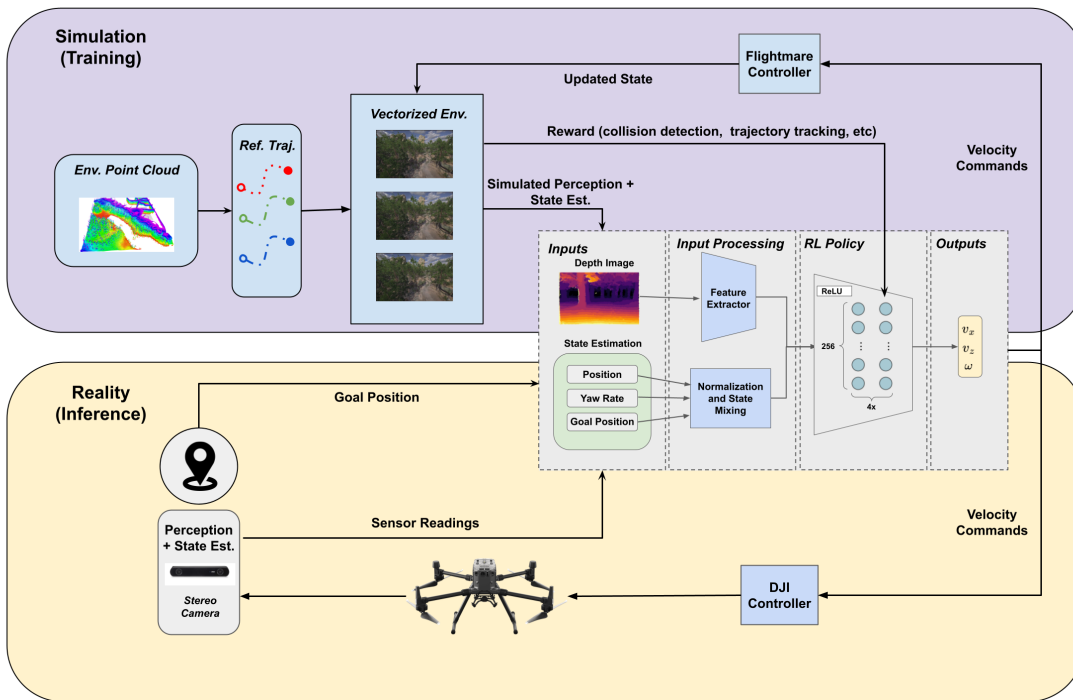


Figure 2.1: The full training and deployment architecture defined in this work. Training happens in simulation, with privileged information of the environment. Given a pointcloud, optimal reference trajectories are created and used in the reward function calculation. During deployment, a DJI Matrice 300 quadrotor is used, with a ZED2i camera for sensing and visual odometry. The output velocities are tracked by Flightmare in simulation, and by DJI’s velocity controller when running on hardware.

the distance to the desired goal, the error distance from the reference trajectory, and any potential collisions. The new state and corresponding depth image of the environment are then passed to get the next policy output. As described in Section 2.5, we use the Actor-Critic method to train two policies – an actor policy that predicts the best actions, and a critic policy that predicts the reward of a given state.

The trained actor policy is deployed on hardware – an off-the-shelf DJI Matrice 300 drone, with a mounted Jetson Xavier AGX, and ZED2i stereo camera. The DJI Matrice 300 has an onboard velocity controller that can track desired velocities. The ZED2i camera captures depth images. Additionally, it runs visual odometry to perform state estimation. In this way, the hardware platform has access to all

required policy inputs, and can track policy outputs. The onboard Jetson Xavier AGX is used for all computation and policy inference.

The policy can be used to control a drone from any initial position, to a goal position 10-20 meters away. The policy handles both path planning to find a feasible path to reach the goal and obstacle avoidance to deviate from any potential collisions. Though its trained to fly 10-20 meter trajectories, it can be extended for use in long-distance flights, or even for indefinitely long missions. For planners that output intermittent waypoints, sequential waypoints from the list can be used as the desired goal position, using the subsequent waypoint once the current one is reached. Goals further than 20m away can also be reached, by continually setting a temporary goal by choosing the point 20m away on the line joining the drone's current position and the goal position. In this moving-horizon manner, we can create intermediary goals at similar distances to those the policy is trained on, which eventually allows the drone to reach far-away goals that it was not trained on.

We now explain each step of the training and deployment process in further detail.

2.2 Flightmare - Quadrotor Simulator and Rendering Engine

The Flightmare [12] quadrotor simulator comes with a physics and dynamics engine, and an executable standalone, which performs environment rendering and camera simulation. The standalone executable communicates with the physics and dynamics engine using NetMQ, a lightweight networking library. In addition to the publicly available standalone, some modifications were needed for the purposes of this work. First, to train a meaningful RL policy, collisions must be detected. Previous RL based works using Flightmare have been purely state-based, with known positions of simple obstacles such as spheres or gates. With such obstacles, it is easy to mathematically check for collisions, such as checking if the distance to the center of a sphere is less than its radius. For this work, however, we wish to deal with more complex environments which can not be modeled using simple geometry. So to detect collisions, Unity must perform mesh collision detection between the entire environment and the given drone object. If a collision is detected, the drone's state is updated, and this information is passed the next time Unity sends a NetMQ message to Flightmare. During training, an environment with 25 drones was used to train in parallel, so minor modifications include ensuring collisions between drones are not flagged. We do not want to penalize collisions from obstacles that may not be in the drone's field of view, and handling dynamic obstacles is outside the scope of this

work.

Our depth images have resolution 192x108, giving it the same 16:9 aspect ratio that the physical ZED2i camera uses. Depth values are linearly scaled from a distance in meters to be in the range $[0,1]$, such that the value 1 represents 100 meters and 0.01 represents 1 meter. An example hyper-realistic forest environment and example depth image can be seen in Fig. 2.2, showing the high quality environment and images the simulator is capable of rendering, and detecting collisions within. To simplify the problem for this work, a simpler environment was used. The environment consisted of many cylinders randomly generated throughout a terrain, similar to tree trunks in a forest as shown in Fig. 2.3

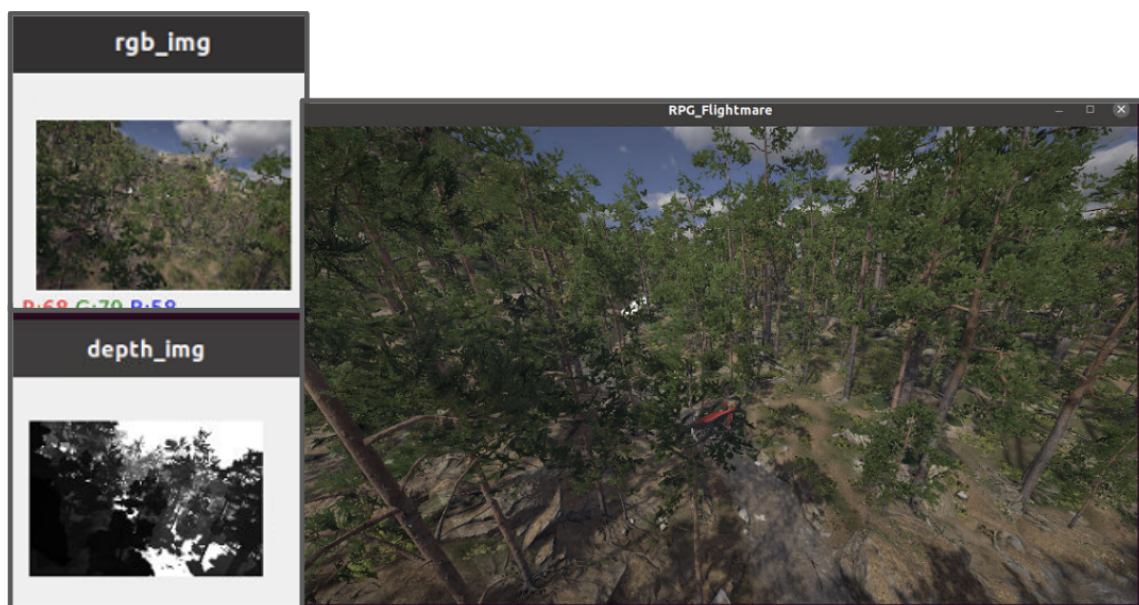


Figure 2.2: The drone flying in a forest environment, running in Unity. On the left are the RGB and depth images taken by the drone in the current frame. This shows the hyper-realistic nature of the rendering engine. Fig. 2.3 is the simpler environment used in this work.

2.3 Trajectory Generation

During training, pre-computed reference trajectories are passed as a supervisory signal to the RL policy. This helps with computing the reward for a given state,

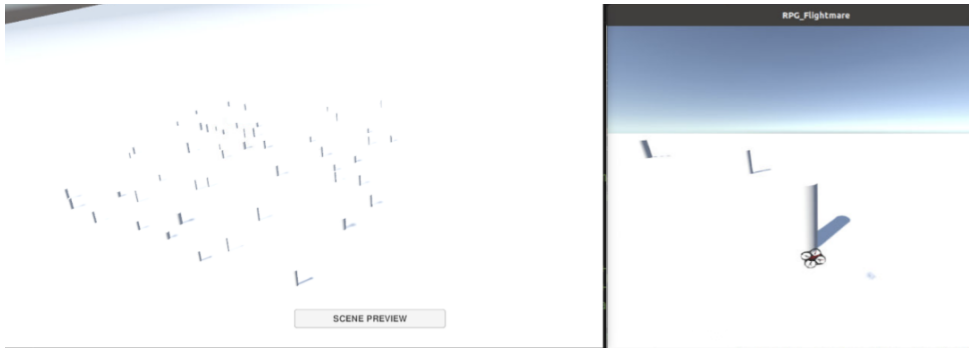


Figure 2.3: The simple environment of obstacles used during training. Left, cylinders were randomly generated at various positions with slightly different height and radius, and are meant to emulate tree trunks in a forest. Right, the drone flying through the environment, with an obstacle in front of it.

because we know the desired position of the drone at every timestep. Reference trajectories were calculated using [14]. The trajectory generation process consists of a few steps. First, we create a point cloud of an $80\text{m} \times 80\text{m}$ section of the environment. This number was chosen because it balances giving a large area to fly in, while not taking too much memory for future steps. Unity was used to do this conversion of the environment mesh into a pointcloud, with a resolution of 0.15m .

Once we have this pointcloud, we randomly sample the space for a point, defined as the start point. If this point is not within 2 times the drone’s radius of any point in the pointcloud, it is a safe starting point. We then randomly sampled a heading direction between 0 and 2π , and choose the goal as a point between 5 and 20 meters away from the start point, at this heading. If this point is at least 2 times the radius of the drone away from the closest point in the pointcloud, it is safe and we set it as the goal point. We now have a start and goal point. Finally, we use code from [14] to solve for the optimal trajectory between the start and end points. Fig. 2.4 shows trajectories going through the environment with cylindrical obstacles. Each trajectory is between 5-15 meters long. The paths are used as part of the reward function, whereby deviating from them results in lower reward. Note that these trajectories are merely to help the policy learn how to reach the goal, they are not meant to be perfectly replicated. Generally, the space of valid trajectories is multi-modal because many valid solutions exist. For example, you can go to the left or the right of an obstacle and still reach the goal. Therefore, the final reward of reaching the goal is much higher than the reward of perfectly following an entire

given trajectory.

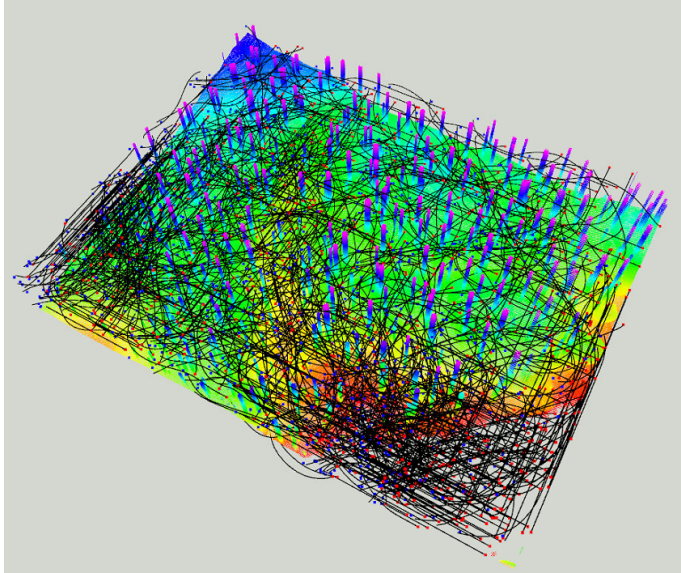


Figure 2.4: Reference trajectories through the pointcloud of the environment. The environment is mostly flat, and is filled with numerous cylindrical obstacles shown in blue and topped in purple. The black lines represent unique trajectories, where each one has its own 3D start and goal point marked in red, and their paths safely go around the cylindrical obstacles.

2.4 Control Scheme

Since the camera is the only way for the drone to sense the environment and detect obstacles, we simplified the drone’s dynamics such that it can only fly in the direction it’s camera is pointing. This means the policy only outputs a forward x velocity, a vertical z velocity, and a yaw rate, as shown in Fig. 2.5, with no way to control the lateral y velocity. In order to achieve motion in that direction, the drone first needs to yaw such that its x -axis now points in the desired direction of motion, and then a body frame x velocity will result in the desired motion. This takes away from the main benefit of drone flight, which is the ability to fly in any direction on the x,y plane regardless of heading, but is important to guarantee safe flight, especially since the camera has a limited field of view. A simplified but safe dynamics model is better than a more complex model that is bound to crash due to limited visibility. In Section 4.2, we describe a 360 degree depth estimation model that could give the

drone full 360 degree vision, allowing the drone to fly anywhere on the x,y plane with full vision of the environment.

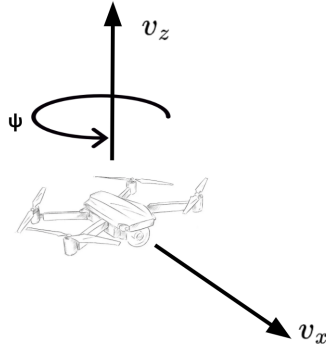


Figure 2.5: The outputs of the policy. We use a velocity controller, so policy outputs are a desired forward velocity, an upward velocity, and a yaw rate. The forward velocity is in the body frame, so in the same direction the camera is pointing.

2.5 Vision-Based RL

The Agile Flight Github repo from [15] contains sample code to train an RL agent using PPO to perform state-based RL. This means the drone has full information of its own state and the environment's, so it always knows how far it is from any and all obstacles. Therefore, no onboard cameras are necessary.

For this work, we use the state-based RL as a baseline, and implement vision-based RL on top of it. This includes changing the observation from just a state vector to also include depth image information. The image is first row-major flattened into a 1D vector, concatenated with the original states, and then passed as input to the RL policy shown in Fig. 2.7. The network un-flattens the vector back to 2D and processes the image and states separately. The inputs, processing, and outputs of the policy are shown in Fig. 2.7, and is described in more depth in the Feature Extraction subsection later in this Section.

In the purely state-based example, images were not used during training. This meant Flightmare could mathematically calculate all quadrotor dynamics, updating the drone's state every Δt based on the given velocity. Given these new states, it could algorithmically detect any collisions. This does not require running Unity, the rendering engine and collision detector. To add vision, we needed Unity to render the environment, simulate 25 cameras, and perform mesh collision for each drone. It

also had to stream $192 \times 108 = 20,736$ pixel values for each camera to the RL policy. The Flightmare + Unity communication is shown in Fig. 2.6. Due to this added computation, the vision-based pipeline is estimated to be over 10x slower than the default state-based pipeline.

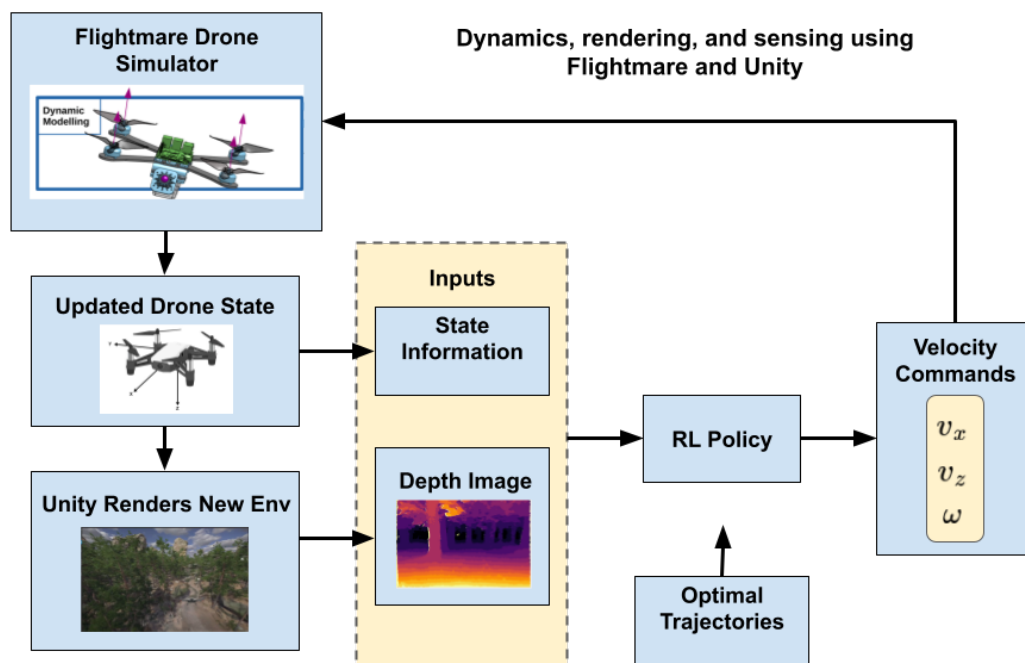


Figure 2.6: The Flightmare dynamics modeling engine communicating with the Unity rendering engine. The dynamics modeling calculates the new state of the quadrotor, given its current state, velocity command, and Δt . This new state is passed to Unity, which updates the drone’s position in the running game engine, and then captures a depth image from this new state. The state and depth image are passed to the RL network, which outputs a velocity for Flightmare to enact, repeating the cycle.

Proximal Policy Optimization

The Proximal Policy Optimization (PPO) algorithm [16] is used, by defining a wrapper around the stable baselines 3 implementation for PPO. In PPO, an actor and critic policy are trained. The actor agent takes in the processed agent state, which includes state and image information, and outputs 3 floats representing a forward velocity, a vertical velocity, and an angular yaw rate. This is shown in Fig. 2.7. Over time it learns to take actions that result in maximal reward. The critic is a policy that predicts the rewards of a given state. Given the same processed agent state as the actor, rather than predicting actions, it predicts a singular scalar value representing how good the given state is. For example, if the agent is bound to crash, it predicts a reward of -1, and if the drone is very close to the goal, it predicts a reward of +2. A reward value of -1 depicts a crashed state, and +2 depicts successfully reaching the goal without crashing.

From [16], the actor minimizes the following surrogate loss, where the first term encourages actions that lead to higher rewards, and the clipping term helps prevent excessively large policy updates and stabilizes the training process.:

$$L_{\text{actor}}(\theta) = \mathbb{E} [\min (r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)]$$

where:

- θ : Actor's parameters
- $r_t(\theta)$: Probability ratio defined below
- A_t : The advantage function, how much better or worse an action is compared to the average action in a given state

The probability ratio $r_t(\theta)$ is the probability of taking action a_t at state s_t in the current policy divided by the previous one. It is computed as:

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

where:

- a_t : Action taken at time t
- s_t : State at time t
- $\pi_{\theta}(a_t|s_t)$: Probability of taking action a_t in state s_t under the current policy
- $\pi_{\text{old}}(a_t|s_t)$: Probability of taking action a_t in state s_t under the old policy

The critic minimizes the mean squared advantage, the different between the predicted and true value function:

$$L_{\text{critic}}(\theta) = \mathbb{E} [(V_{\text{target}} - V_{\text{predicted}})^2]$$

where:

V_{target} : Target value, the discounted sum of future rewards from this state

$V_{\text{predicted}}$: Predicted value by the critic network for a given state

The final objective function in PPO combines the actor's policy gradient loss and the critic's loss, and adds an entropy bonus to reward exploration:

$$L(\theta) = L_{\text{actor}}(\theta) - c_1 \cdot L_{\text{critic}}(\theta) + c_2 \cdot \text{Entropy}(\pi_{\theta})$$

The Adam optimizer is used while training, with a learning rate of $1e - 4$.

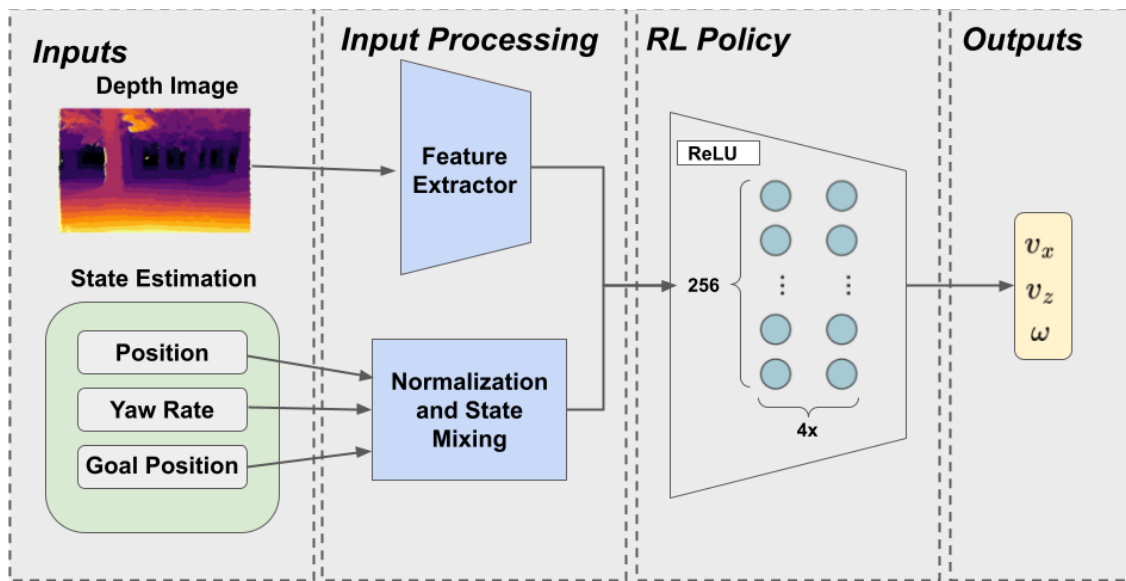


Figure 2.7: The inputs and outputs of the RL policy. The most recent depth image from the drone goes through a custom feature extractor where it is featurized and then pooled into a 1D vector. State and goal information is normalized and mixed via linear layers, and then concatenated with the image representation, to be inputted to a 2-layer MLP. The policy outputs 3 numbers, representing x, z body frame linear velocity in m/s , and a yaw-rate in rad/s .

General Architecture

Deep RL agents follow a state-action-reward loop, where the agent measures its state, follows an action based on that state, and then measures the performance of that action by checking the corresponding reward. But the raw state is not directly passed into the network. Fig. 2.7 summarises the processing the input goes through. The raw states are first passed through a feature extractor – where the robot states like position and orientation are normalized, and the image is featurized to a lower-dimensional space with more meaningful, task-specific information. The normalized state is stacked on top of the image’s feature representation, and then this new 1D vector is passed to the network. A full description of the feature extraction process can be found in Subsection 2.5, and in Fig. 2.8.

In the following sections, we discuss the agent’s state space, action space, feature extraction to better understand state information, and the reward function that determines how good a given state is.

State Space and Action Space

The agent’s state at any given time is defined as a 1D vector including various robot position, orientation, and depth image information. The exact state vector is defined in Table 2.1.

There are 7 state values, all defined in a fixed map frame. The first 3 states are positional errors, defined as the difference between the goal position and the current position, along the x,y, and z axes in meters. Then there is the absolute roll and pitch angles, in radians. Next is a heading error, defined as the difference between the ideal heading assuming no obstacles, and the actual current heading, in radians. The ”ideal heading” is assumed to be the heading that makes the drone face perfectly toward the goal. The final robot state value is the drone’s yaw rate, in radians per second. Rather than having two separate entries for the current and goal positions/headings, the difference between the current and goal positions/headings is used as one quantity. This was done to help the policy generalize to various situations. For example, in a state-space where current and goal positions are separately passed in, a current position of (0,0,0) and goal position of (10,0,0) is a whole different situation from a current position of (10,0,0) and a goal position of (20,0,0), even though the general idea – go 20m straight ahead – is the same. By using the error to combine the current and goal positions into one term, both situations above can be depicted with an error of (10,0,0), showing a relative goal of +10 along the x axis, and 0 in y and z. For the heading, this provides two benefits. The first is the same as the case with positions. Separating current and desired headings would mean a

State value	Unit
goal_pos_x - curr_pos_x	meters
goal_pos_y - curr_pos_y	meters
goal_pos_z - curr_pos_z	meters
roll	radians
pitch	radians
desired_heading - curr_heading	radians
yaw rate	radians/second
depth image pixel row0 col0	meters
depth image pixel row0 col1	meters
depth image pixel row0 col2	meters
...	...

Table 2.1: The state space of the agent. It contains 7 robot states – the positional error defined as the difference between the goal position and the current position in x, y, and z, the absolute roll angle of the drone, the absolute pitch of the drone, the difference between the theoretical heading angle creating a straight line to the goal and the actual current heading, and the current yaw rate of the drone – and 192×108 pixel values from the depth image. For the states, positional error and heading error are used instead of separately defining the current position/heading and goal position/heading, to help generalize over various situations.

current heading of 0 radians and desired heading of 0.5 radians is seen differently from a current heading of 0.5 radians and desired heading of 1 radian. By using the difference, in both cases the error is 0.5 radians, and since the policy outputs yaw rates, all it needs to know is which direction to turn. Additionally, using the difference simplifies the situation. Since headings are measured in a global coordinate system rather than a local coordinate system, multiple situations with the goal straight in front of the drone will have different ideal headings. For example, going straight West when already heading West is a different ideal heading from going straight North when already heading North because these angles are calculated in the global coordinate system. But since the policy outputs yaw rates, all it needs to know is a relative heading to turn left, right, or not at all. By giving a relative heading, the error is 0 radians in both cases of maintaining a North-bound or West-bound heading, which means the drone does not need to turn at all. So passing relative positions and headings help the policy generalize across various situations.

In simulation, all state values come from the Flightmare simulator. In the real world, the goal position is human defined – and eventually passed in by a higher-level

planner, as described in Section 4.1). The instantaneous position and body angles are measured by the ZED2i’s camera running visual odometry. This means GPS is not required. The yaw rate is given by the drone’s Inertial Measurement Unit (IMU).

In addition to robot state information, the agent’s state includes depth image information. For this, the two dimensional 192×108 image is flattened to a one dimensional $192 \times 108 = 20,736$ -vector of row-major pixel values, where each value represents a depth in meters. In simulation, depth information comes from Unity, and in the real world, the depth image is retrieved from the ZED2i stereo camera.

The RL agent takes in the latest agent state passed in as a 1D vector as described in Fig. 2.7. It passes this state through a feature extractor, and then through the policy network. The final output of the policy defines the action it believes the agent should take. Since we aim for a transferable controller that can be used across drone types with varying dynamics, the outputted actions are high-level velocities that the lower-level drone controllers then needs to follow, as seen in Fig. 2.5. The action space is defined by 3 values, a forward velocity in the direction the camera is pointing, an upward z velocity, and a yaw rate for the drone to turn. As described in Section 2.4, we only want the drone to fly in a direction where it has visibility. This means there is no lateral velocity followed, and if the drone wants to move left, it must first yaw toward that direction so the camera can see there, and then fly locally forwards.

Reward Function

The reward of a given state is based on many factors, with the total reward being the sum of all reward terms shown in Table 2.2. Final reward weights were set using reward shaping and substantial experimentation. Terms were added to influence specific positive behavior or detract from negative behavior, and trial and error led to important terms having higher weights and vice versa. The most important term is the distance to the goal, whereby lower distances result in higher reward as it shows progress toward reaching the goal. This expression is calculated as $(d_0id)/d_0$, where d_0 is the distance to the goal from the starting position and d is the distance to the goal from the current position. This expression has a reward of 0 at the starting position, and +1 at the goal position. Next, there is a reward for pointing in the direction of the goal, set inversely proportional to the heading error θ_e . This is meant for cases with no obstacles, where the ideal path is to simply fly straight toward the goal. The detailed expression can be found in Table. 2.2. Next, is a term that penalizes deviating from the privileged reference trajectory. Then there is a term that penalizes being too close to an obstacle, based on the center of the depth image – the area directly in front of the drone. If a depth in this section is very small, the drone may be bound to crash. The above terms epitomize the desired

Term	Expression	Weight
Distance from goal	$\frac{d_0-d}{d_0}$	$6e - 3$
Heading	$e^{-16\theta_e^2} + \begin{cases} 0.5 & \text{if } \theta_e < 0.26\text{rad}, \\ -4.0 & \text{if } \theta_e > 1.3\text{rad}. \end{cases}$	$5e - 4$
Trajectory following	$e^{-2d_{traj}} + \begin{cases} -0.4d_{traj} & \text{if } d_{traj} > 3m, \\ 0 & \text{otherwise.} \end{cases}$	$\frac{d_0-d}{d_0} \cdot 3e - 4$
Obstacle Penalty	$\begin{cases} d_{obstacle} - 1 & \text{if } d_{obstacle} < 1m, \\ 0 & \text{otherwise.} \end{cases}$	$-2.5e - 4$
Vel toward goal	v_g	$4e - 5$
Heading Rate	$abs(\omega)$	$3e - 4$
Vertical position error	$abs(e_z)^2$	$-8e - 5$

Table 2.2: The reward terms of a given state, as a function of the initial distance to the goal d_0 , the current distance from the goal d , the heading error θ_e , the distance to the closest point on the reference trajectory d_{traj} , the closest object in the center of the image $d_{obstacle}$, the velocity v_g along the axis pointing directly to the goal, the heading rate ω , and error in z position e_z . Reward shaping was used to guide the RL agent toward effective behavior, and substantial trial and error over numerous trained models led to the final weights.

high-level behavior of the RL agent. After experimentation, some lower magnitude terms were added to guide the learning process of the agent, using the reward shaping approach. These terms include a reward that encourages higher magnitude velocities pointing toward the goal, a term to penalize high yaw rates to avoid jittery behavior, and a term that penalizes errors in the z height of the drone relative to the goal height. These terms encourage positive behavior and penalize negative behavior in the beginning of the learning process when the agent is still experimenting with random actions and the main reward terms are likely to be zero or even negative. Over time, these terms influence positive behavior, which in turn results in higher rewards from the main terms. Expressions for all reward terms can be found in Table 2.2, where the total reward is the sum of all individual reward terms. These rewards are calculated at every time step, and a higher reward means the drone is in a better state, while a lower reward mean it is in a worse state.

In addition to time step based rewards, there are also terminal rewards. These rewards are given if the drone is in a terminal state, such as reaching the goal or crashing into an obstacle, resulting in rewards or penalties with much higher

Terminal State	Terminal Case	Reward Value
Reached Goal	$d < 0.75m$	+2.0
Crashed	Drone collision with any obstacle or terrain	-1.0
Timed Out	$time > 50seconds$	-0.08
Out of Bounds	$d > 30m$	-0.160

Table 2.3: Terminal state rewards given for reaching positive or negative terminal states. These have much higher magnitude than per time step rewards because they represent the performance of the policy much better than any random state can. Like before, d is the distance to the goal.

magnitude. These terminal rewards have much higher magnitude than the individual time step rewards, because a terminal state signifies the performance of the policy much better than any other time step possibly can. Terminal rewards can be found in Table 2.3, where final reward values are chosen after significant trial and error to limit negative behavior but still encourage exploration. For example, a penalty for crashing is evidently needed to promote safe flight, but an excessively high penalty may also lead to non-optimal behavior. For example, the agent may get stuck in a local maximum where it learns not to crash, but only by flying in small tight circles until it times out after 30 seconds. Such a local maximum prevents exploration by excessively penalizing crashing and heavily restricting the learning process.

Terminal rewards are magnitudes of order greater than per timestep rewards because they signify the overall performance of the agent. While a policy may not follow the reference trajectory well, if it still reaches the goal, that is still much better than a policy that mostly follows the reference trajectory, but crashes right before reaching the goal. Additionally, this balances the two forms of learning in this work. The supervisory reference trajectories show up in the reward, where following the reference well results in high reward. Additionally, since other potential solutions still exist, such as going around the left of an obstacle versus going around the right, both solutions are rewarded for reaching the goal despite deviating from the reference.

Therefore this work combines traditional Deep Reinforcement Learning, where an agent interacts with an environment to maximize rewards, with imitation learning, where an expert supervisor provides the learner with optimal data. In [11], pure imitation learning is used, as a neural network is trained to strictly match the optimal trajectories given by the supervisor. There was no idea of an agent in an environment, any learned dynamics of the world, or any experimentation with policy outputs to find optimal actions based on rewards. In this work, we keep an optimal supervisor, but as a part of training a reinforcement learning agent. The supervisory signal is

part of the reward function, but the agent still interacts with an environment and performs different actions over time, learning dynamics. Since following the expert supervisor’s trajectory results in higher reward, knowledge is distilled down from the expert supervisor to the learned policy.

Feature extractor and Input Normalization

The full agent’s state is overly complex and image data is hard to understand through raw pixel values. Therefore, we pass it through a feature extractor, as seen in Fig. 2.8. A feature extractor reduces the dimensionality of an input and converts it to a better representation for the given task. The state is processed in three steps – state mixing, image featurization, and then concatenation of the two – as described below.

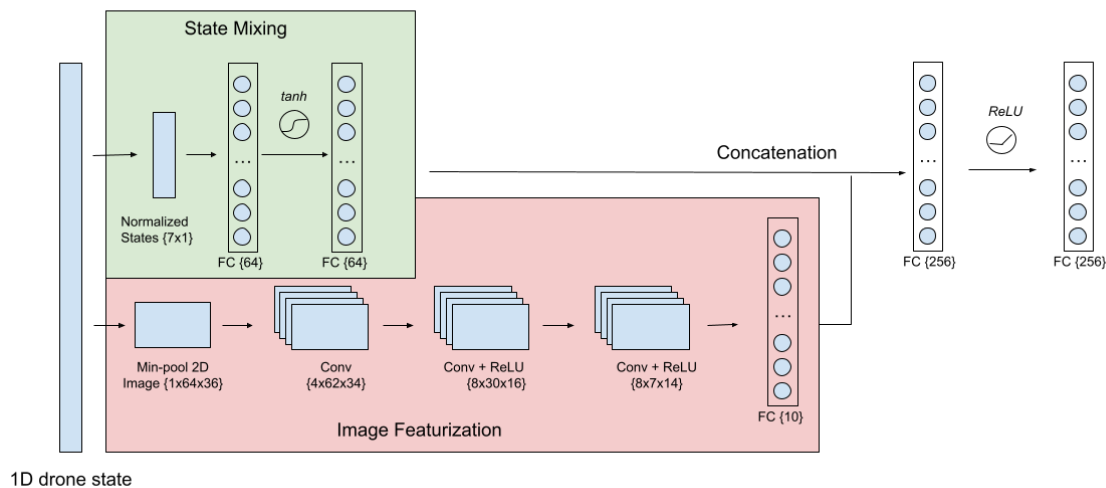


Figure 2.8: The architecture of the custom feature extractor. The full 1D state is separated into the 7 state values and 192×108 image pixel values. Both are independently processed into 1D feature vectors of size 64 and 10, respectively. They are then concatenated and passed through fully connected layers for more state mixing. The final vector of size 256 is passed as input to the RL actor and critic. This light-weight architecture runs at 10Hz on the physical drone platform.

State Mixing

The state mixing architecture is shown in the green box of Fig. 2.8. The 7 state values are first normalized to be in the range $[-1, 1]$. Distances in meters are normalized assuming a world size of $80 \times 80 \times 60$, and angles are reduced from $[-\pi, \pi]$ to $[-1, 1]$. Normalization is an important step to ensure some values do not have higher weights than others. For example if distances can be up to 80 meters, but angles can only be up to 2π radians, then distances will have a higher weight than angles. After normalizing the 7 states, they are passed through a fully connected linear layer of output size 64, then a *tanh* layer, and then another fully connected linear layer of output size 64. The linear layers are used to mix state values, to find potential relationships between state values.

Image Featurization

Image featurization is shown in the red box of Fig. 2.8. The image is passed in as a 1D vector of size 20,736, in row-major order. It is first reshaped back to a 192×108 2D image to regain spatial relationships and features such as edges, textures, and shapes. To reduce the number of weights, it is then min-pooled down to 64×36 . We choose the min-pool resize operation in order to keep the lowest values from each image window, thereby ensuring we always keep track of the closest obstacle from the depth image. With interpolation, pixels are averaged and therefore the resulting depth values do not accurately represent real world depths. The 2D image is then passed through a custom feature extractor. This architecture is adapted from [17], by modifying the input and output channels to keep our aspect ratio of 16:9 instead of their 4:3. The output of the feature extractor is a 1D vector of size 10, down from 20,736. We do not need all 20,736 raw pixel values, but rather a general understanding of the obstacles around us. The 10 numbers encode a non-human-readable representation of obstacles, such as distances, angles, and heights. The output length of 10 is matched from [17], and our empirical results show these 10 features provide enough environment understanding to avoid obstacles. The image featurization network is much smaller than popular image classification models because the problem it solves is also simpler. The network does not need any understanding on classes of objects in the environment, or even to identify textures or shapes. Rather the obstacle avoidance problem just needs to identify regions with low pixel values, as low pixel values represent nearby objects.

We initially used MobileNet for image featurization from [18], as is used in [11]. MobileNet is a pre-trained image classification model meant to be run on low compute mobile and embedded devices. Despite its lightweight architecture, we achieved a

maximum frequency of 2Hz on our hardware, which is much too slow for real-time autonomy. By changing to our custom feature extractor shown in Fig. 2.8, we were able to run our policy at 10Hz.

Concatenation

After normalizing and featurizing the states and image, we concatenate the 64-vector representation of state information with the 10-vector representing obstacle data. This is shown in Fig. 2.8. The resulting vector is then passed through a fully connected layer of output size 256, then through ReLU, and then another fully connected layer of output size 256. These linear layers add more mixing between states and obstacles. For example, the pitch of the drone impacts what region of the image is most important. When the drone is parallel to the ground, the center of the image represents the environment ahead of the robot. If the drone pitches down, however, the center of the image likely contains the ground, and the top of the image now represents the environment ahead of the robot.

The final vector of size 256 is passed as input to the RL agent, for it to output actions and predicted rewards.

2.6 Sim 2 Real

To ensure a policy trained in simulation transfers over smoothly to the real world, a few steps are taken.

The main difference in moving from simulation to real hardware is the quality of depth images captured. In simulation, Unity returns perfect images with exact depths. Such perfect data can never be realized in the real-world, so we must process the simulated images to make them more life-like. This includes adding random gaussian noise to every pixel depth value, with a mean of 0m and standard deviation of 0.2m. This adds inaccuracies to depth values, mimicing outputs of a camera in the real-world. These noisy pixels are maxed with 0 to ensure all depth values are greater or equal than 0. Additionally, 0.1% of randomly chosen pixels are set to NaN, representing unknown depth values. Training is done on these processed images, so the model learns to expect noisy data thereby transferring well to the real-world. Ideally, we should have added depth-dependent noise, but we found this gaussian noise with the same μ and σ to be reasonable.

Additionally, when deployed on hardware, images from the camera are processed to make them as clean as possible. This is done by the ZED2i camera's built-in neural depth mode to make depth images as accurate and smooth as possible. While

normal depth images may contain noisy values, neural mode on the ZED2i smoothens each object with high accuracy. Fig. 2.9 shows the results of applying neural mode, where noisy unrealistic values are filtered out.

In addition to using the neural depth setting, we also use fill mode, which does its best to fill in areas of low confidence. With a stereo camera, it is sometimes mathematically impossible to find depth, such as at the edges of objects where there is occlusion. Only one of the two cameras can see some sections of an edge so depth cannot be calculated, and normally these areas are set to NaN. With fill mode, these areas are filled in with the best estimate that can be made. Empirically, these values are quite accurate, and are essentially set to the background’s depth. Fig. 2.9 compares the depth image with various amounts of processing. (a) shows the original depth image without fill mode or neural mode. It is extremely noisy. (b) shows the same environment, using fill mode and neural mode, and is much smoother. However, it still has some obstruction from the drone’s propellers, so the mask from (c) is applied to filter out the remaining noise. (d) is the masked, downsampled image that is passed to the feature extractor. Overall, the resulting image is much cleaner than the initial raw image captured. In this way, we reduce the gap between simulation and the real-world and images at deployment similarly match those seen during training.

Regarding drone dynamics, we simply follow a zero-shot transfer approach. If we were operating at the level of body rates or thrusts, a policy trained on one platform would have to be re-trained to be used on a new platform. However, since we use a velocity controller, we can mostly ignore the differences in dynamics, and rely on each drone controller to convert desired velocities into motor thrusts. The drone used in simulation has different dynamics than the physical one used, but our results show the controls were not an issue, as discussed in Section 3.2. One downside of this approach, however, is that velocities are not tracked in a closed-loop manner. [11] outputs trajectories that an MPC controller tracks, meaning disturbances from external factors like wind and model inaccuracies can be accounted for in future time steps. The velocity control is applied for a single timestep, and if a desired velocity is not closely followed there is no closed loop control to correct for it. The subsequent policy output is responsible for a corrective output. Additionally, by controlling velocities instead of body rates or thrusts, the platform loses some agility, but for the purposes of this work we are not focusing on complex maneuvers.

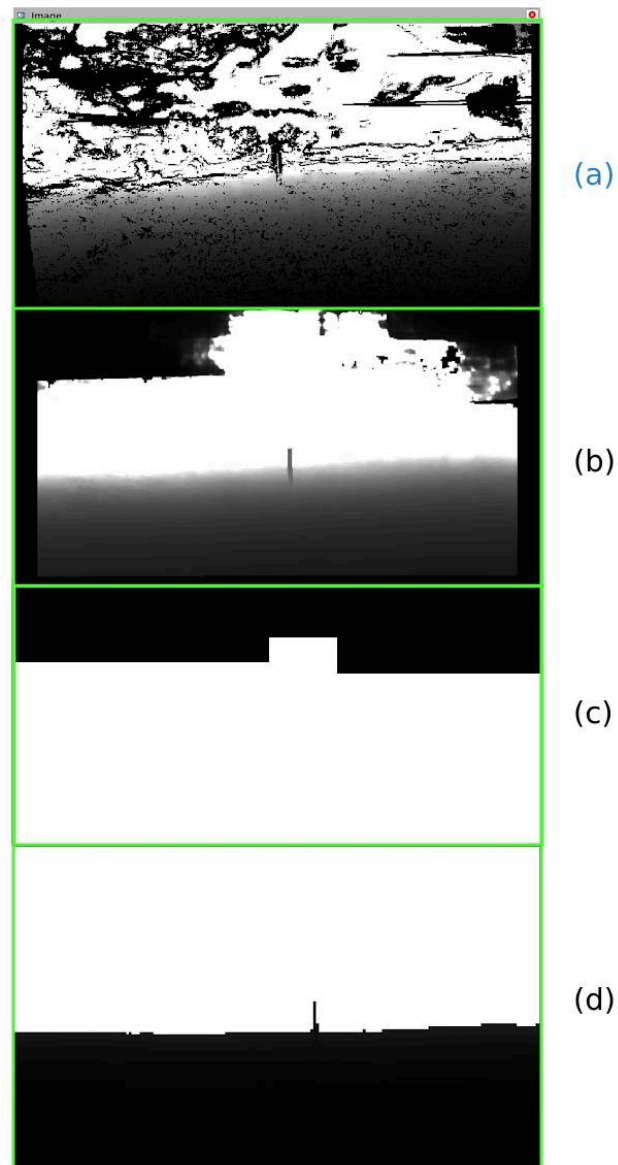


Figure 2.9: Comparing the depth image with different amounts of processing. (a) is the raw noisy unfiltered image, (b) is the same image with ZED2i’s built-in neural and fill modes. (c) is the mask used to get rid of noise from the propellers. (d) is the masked downsampled image that is passed to the feature extractor. The scale of pixel values in the final image is very large, so the ground loses its humanly visible gradient, even though the actual values are correctly varied.

2.7 Physical Platform

The entire platform with all sensors mounted can be seen in Fig. 2.10. It includes a DJI Matrice 300 drone, ZED2i stereo camera, and Jetson Xavier AGX. All components are described in further detail in the following sections.



Figure 2.10: The physical platform, including a DJI Matrice 300 drone, ZED2i camera, and Jetson Xavier AGX.

DJI Matrice 300

We use the off-the-shelf DJI Matrice 300 as our drone platform. DJI provides an On-board Software Development Kit (OSDK) with ROS integration, to send and receive important information. It streams high frequency values such as attitude, IMU, and

accelerometer data, and accepts control commands such as waypoint positions or desired velocities. In this way we use ROS to subscribe to important state information, and make ROS service calls with our policy outputs to control the drone. It also has onboard cameras and sensing, but these are explicitly turned off. The default sensing capabilities include detecting the presence of an obstacle and accordingly stopping before colliding with it. While this is a valuable safety feature for a human operated system, it does not solve the path planning problem for an autonomous system to figure out how to go around an obstacle and get to a goal location. Again, these features were disabled during all flight tests.

ZED2i Camera

The ZED2i camera is an off-the-shelf stereo camera. It provides high quality depth images and performs visual odometry for pose tracking. Both of these are important because the drone needs to know where it is within the environment and what the environment around it consists of. The ZED2i uses a stereoscopic camera to estimate depth and track its position and orientation as it moves.

We mount the ZED2i to the gimbal under the DJI Matrice 300. The gimbal has dampers acting as shock absorbers that drastically reduce vibrations and oscillations. This results in smoother and more stable camera motion, despite any erratic drone flight.

As described in Section [2.6](#), the camera is running in Neural depth mode to get accurate and smooth depth values, and in fill mode to reduce the number of *NaNs*. We query images at 20Hz with both processes running onboard the Jetson. Images are 720×1280 , with a 16:9 aspect ratio similar to those used in simulation. We downsample the image to be 192×108 using bilinear interpolation, so it has the same resolution as those used during training. The Zed2i also has the Ultra depth mode, which smoothens depth values. While it runs at a higher frame rate than the Neural depth mode, we choose to use Neural for its higher quality resulting image.

The ZED Python API is a wrapper around the ZED SDK, which is highly optimized C++ code. It allows us to query images, position, and orientation, which we publish via ROS at 20 Hz.

The field of view of the camera includes the spinning propellers, as shown in Fig. [2.11](#). This results in very noisy depth images, as the ZED2i's onboard depth calculation gets confused by the rapidly moving propellers. To solve this, we simply mask out part of the image to remove this noise. Fig. [2.9](#) compares the depth image at various processing levels.



Figure 2.11: Propellers block the camera’s field of view, resulting in noisy depth images. Left, an RGB image from the camera showing propellers jutting out from both sides of the image. Right, the depth image showing noise values where the propellers are rapidly spinning. This section is masked out before being passed to the policy.

Jetson Xavier AGX

We use the Jetson Xavier AGX as an onboard computer running all computation necessary for this work. It queries raw images from the ZED2i, runs the image smoothing neural networks for neural mode, obtains position information using the ZED Python API, and streams everything via ROS. It then takes in these inputs, featurizes them, and passes the resulting feature vector through our RL policy. Finally, it sends the resulting output command through a ROS Service call to the DJI ROS OSDK node, also running on the Jetson. The Jetson communicates with the DJI Matrice 300 via an expansion module that mounts to the drone. Two cables are needed for this. The first is a USB-A to USB-C cable, with the USB-A side plugging into the Jetson and USB-C plugging into the OSDK port on the drone. Additionally, a USB-TTL cable is needed, where the USB-A side again plugs into the Jetson, and the TTL serial side plugs into the expansion module’s ports, as shown in Appendix A.

Coordinate Frames

Every sensor has its own coordinate frame, so we must be careful to properly convert between each one and stay consistent in their notation. The policy outputs velocities in the drone’s FLU frame, where x,y,z refers to Forward, Left, and Up from the drone, as seen in Fig. 2.13. The ZED2i is also in the FLU frame, but at a slight offset. It is centered on the y -axis, but 0.2159m ahead in x , and 0.1778m below in z from the

drone's center, as seen in Fig. 2.12. This means when sending the drone's position to the policy, we must perform a slight translation upwards and backwards, so the policy gets the position of the center of the drone rather than the camera.

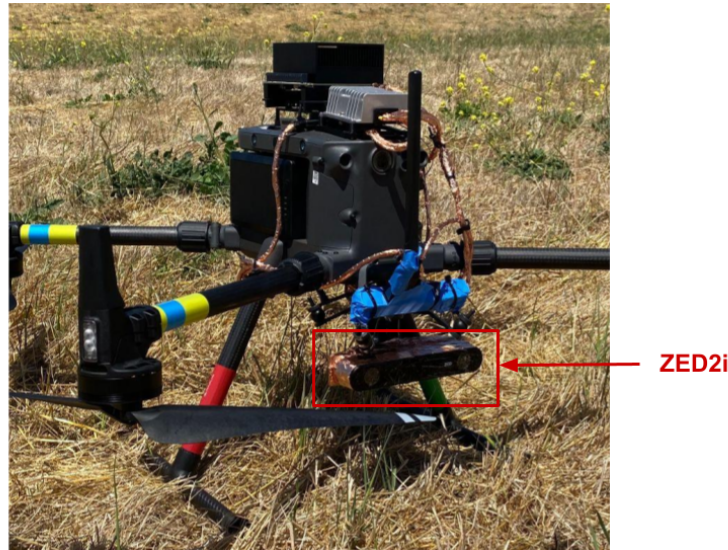


Figure 2.12: The ZED2i camera is mounted slightly below and in front of the drone's center.

The DJI OSDK uses a different coordinate frame as seen in Fig. 2.13. The velocity controller accepts velocities in the global NED frame, where x is North, y is East, and z is Down. Since the policy outputs body frame velocities, we must convert these from FLU to NED by rotating the forward velocity by the drone's heading and negating the desired yaw rate. We must negate the yaw rate because it is an angular rotation about the z -axis, and NED's z -axis points Down while FLU's z -axis points up. The global frame velocity is calculated with the following:

$$v_{\text{North}} = \cos(\psi) * v_F$$

$$v_{\text{East}} = -\sin(\psi) * v_F$$

where:

v_F : the forward velocity in the FLU frame

ψ : the heading angle relative to North

The drone's heading ψ comes from the drone's internal compass, and allows us to convert between the global NED and local FLU coordinate frames.

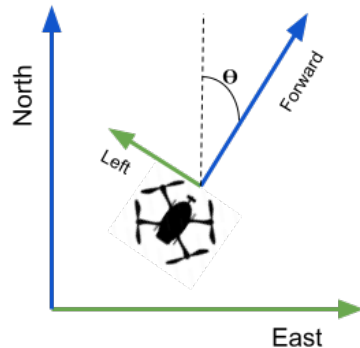


Figure 2.13: The global NED and local FLU coordinate systems. The NED frame's axes point North, East, and Down, while FLU points locally Forward, Left, and Up. The heading angle θ determines the rotation between the two coordinate systems, and is measured by the drone's compass.

Interference Issues

With all devices and sensors running onboard the drone, there is a great deal of data and therefore current, being passed through the various cables. This resulted in the drone detecting signal interference, preventing us from taking off. To get around this, we covered all wires with Electromagnetic Interference (EMI) shielding copper tape, which reduces the electromagnetic interference and noise created by current traveling through wires. This reduced the signal interference.

Chapter 3

Evaluation and Experiments

In this Chapter, we describe the results seen upon deploying our trained RL policy. In Section [3.1](#), we go over results from experiments in simulation, and in Section [3.2](#), we go over the physical test location, safety procedures, results, and sensor analysis to characterize the performance of our policy.

3.1 Simulation Results

In simulation, the drone performs well at reaching the goal. Fig [3.1](#) shows how the policy learns over time. Initially, it takes random actions, resulting in negative reward from crashing. In roughly 300 iterations it converges to a policy that averages a reward of roughly 2.0, which is the reward for reaching the goal. As the model learns, it first learns to fly close to the goal. Earlier iterations try to maximize the reward by staying close to the obstacle without getting the terminal state of reaching within 0.25m of the goal. This resulted in the drone simply circling around the goal as seen in Fig. [3.2](#), which collects many small rewards for being close to the goal. Eventually, it learns that the terminal reward of reaching the goal as soon as possible is better than multiple small rewards for being close to the goal. Reward tuning was important to ensure the terminal state is reached as fast as possible. Reward tuning is discussed in Chapter [2.5](#).

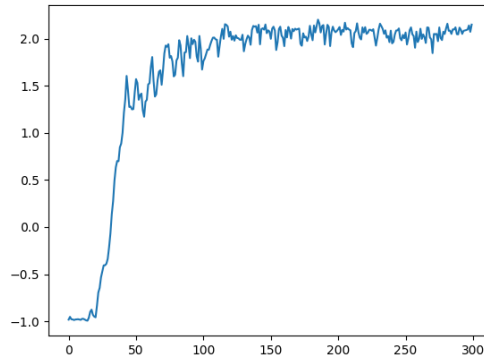


Figure 3.1: The reward per drone per iteration, averaged over a 25 drone training environment. It converges in roughly 300 iterations.



Figure 3.2: Midway through the training process, the policy learned to stay close to the goal without actually reaching the terminal state, resulting in it circling the goal. Reward tuning is important to encourage the drone to go directly to the goal as soon as possible.

It took roughly 3 hours and 10 minutes to train on an NVIDIA Quadro RTX 8000 graphics card with 48GB memory. This policy is tested in a new, unseen test environment, where every trajectory was forced to avoid an obstacle. The drone

safely reaches the goal in 500/500 cases. Training and testing are both done at 1m/s.

Fig. 3.3 shows a top down view of some of these drone trajectories, where the drone starts at the yellow point and reaches within 0.2m of the green goal point, while avoiding the red obstacles. The plots showcase the RL agent’s ability to fly around obstacles and reach desired goal locations.

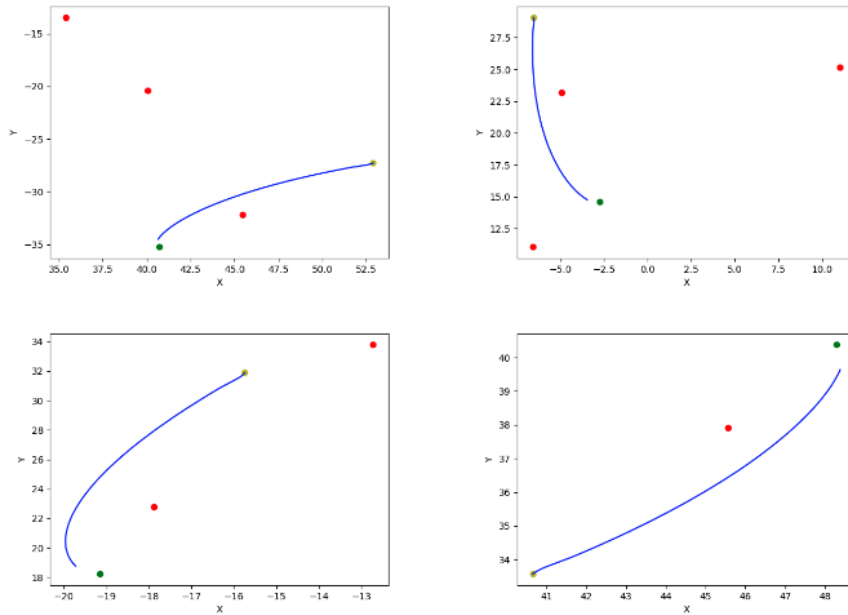


Figure 3.3: Top down view of example test trajectories. Starting at the initial point (yellow), the drone reaches within 0.2m of the goal point (green), and avoids obstacles (red). All units are in meters.

3.2 Real-World Tests

Test Location and Setup

All physical flights were done at the Cesar Chavez Park in Berkeley, CA, and were logged and approved by the Risk and Safety Solutions Platform for UC Berkeley. We flew in an open, mostly flat section of the park, and installed a foam man-made ”obstacle” to mimic a tree trunk.

The test field and obstacle setup can be seen in Fig. 3.4. The foam obstacle was carefully crafted out of soft pool noodles that would not risk damage to the drone, and affixed to the ground using wooden and rubber stakes. The drone platform and sensors cost over \$10,000 so a safe test field was of utmost importance. While we do not want the drone to crash, we must prepare for such an event. Additional safety precautions are described below.



Figure 3.4: The Cesar Chavez park in Berkeley, CA, where physical flight tests were conducted. The black cylindrical object is the "obstacle" used in our tests, constructed using pool noodles.

Validation and Safety

Various measures were taken to ensure we are in full control of the drone at all times. First, we made sure policy outputs were properly converted from the drone FLU frame to the world NED frame by creating a manual keyboard controller. Pressing

the 'w' key would take a locally forward velocity at 1m/s, convert it to NED, and send it to the drone for 0.1 seconds, making the drone move slightly forward. The 'q' and 'e' keys would similarly send positive and negative yaw rates to rotate the drone. A "locally forward" velocity means the drone always moves in the direction it and its camera are facing. After yawing in any direction, pressing 'w' again would result in motion in a different direction in the world frame. This manual validation ensured that when policy outputs are correctly converted from the FLU to NED frame, as discussed in Section 2.7.

Additionally, a few safety features were added to ensure safe testing even if the policy sent erroneous outputs. At all times, there was a human pilot watching the drone ready to take away SDK control and fly the drone manually if needed, by flicking a switch out of 'P' to the 'T' or 'S' mode on the remote controller. These modes represent Positioning mode, Sport mode, and Tripod mode. The drone can only be programmatically controlled in the 'P' mode, so switching modes takes away OSDK control and gives full control to the human pilot.

Moreover, if the drone flew over 30m away from the given goal, it would immediately stop running the policy and force the drone to hover in place, returning full control to a human pilot. This ensured the drone could not fly out of the safe bounds of the test site. Additionally, at all times, the user could press the 'r' key to release DJI OSDK authority, meaning the OSDK could no longer control the drone programmatically, forcing the drone to hover in place and only follow the human pilot's commands. In these ways, we ensured all tests were safe and under full human control in case something unexpected happened.

Real World Results

Various real-world tests were conducted to test the effectiveness and robustness of the system. All results are summarised in Table 3.1. All tests began by manually taking off, hovering in place, setting a goal 20m away, and then giving the RL policy full control to reach the goal. The policy was trained at a maximum speed of 1m/s. In the real world we tried three different maximum speeds – 0.5m/s, 1m/s, and 1.5m/s. These speeds were applied by scaling the policy's output by the desired maximum speed. In all cases, the policy was running at 8Hz.

First, we ensured the drone could reach the given goal in the absence of any obstacles. 2/2 tests succeeded in reaching the goal. Next, we added a single obstacle roughly halfway between the drone and the goal position, requiring the drone to maneuver around the obstacle to avoid a crash. At 0.5 m/s, it reached the goal 4/5 times, and at 1 m/s it reached the goal 9/12 times. In each case, the drone was set at different distances from the obstacle, and at different real-world angles relative to

# Obstacles	Speed	Initial Heading	Success Rate
0	0.5 m/s	0°	2/2
1	0.5 m/s	0°	4/5
1	1 m/s	0°	9/12
1	1.5 m/s	0°	0/1
1	0.5 m/s	180°	1/1
3	0.5 m/s	0°	0/1

Table 3.1: Results from real world flight tests. The model was trained at 1m/s, with 0 or 1 obstacles in each trajectory. These situations performed very well. Increasing the number of obstacles to 3, or speed to 1.5m/s resulted in bad performance where a human pilot had to take over control. Starting at the wrong initial heading also worked, showing the policy’s learned knowledge of path planning in cases where the goal may be in any direction.

the obstacle. This was done to ensure the policy can generalize to avoid obstacles at different distances and with different backgrounds. Backgrounds included flat ground, small nearby hills, the distant Berkeley hills, and the San Francisco Bay.

While the drone is flying, we can visualize policy inputs and outputs in real-time. Fig 3.5 shows two example policy outputs, visualized in RViz. In each case, there are two grayscale images to the left. The top image of the two is the raw depth image from the ZEDi, and under it is the lower resolution masked depth image that is passed in to the policy. The green path shows the drone’s trajectory based on the ZED2i’s position estimate, and at the end of the trajectory is an arrow, showing the policy’s desired output velocity. The small green sphere is the goal. In the top image of Fig. 3.5, the drone is in front of an obstacle and outputs a velocity to go left to avoid the obstacle. Once it is passed the obstacle as seen in the bottom image, it outputs an opposite velocity to head back toward the goal. The cylindrical obstacle in this image was added as a post-processing step for visualization purposes, as RViz has no positional or structural information of the real obstacle.

Fig 3.6 is a screenshot of a video comparing real-world footage of the drone flight to the RViz visualization. Similar to before, RViz data includes the starting position, current position estimated by ZED2i using visual odometry, processed depth image, real-time policy outputs, and the goal location. The full video comparing the entire flight with the real-time visualization can be found [here](#), where the video is at 2x speed. Fig 3.7 shows a successful test at 0.5m/s, showing the drone goes around the obstacle and then reaches the desired goal. Fig. 3.8 shows a few more trajectories where the drone successfully goes around an obstacle and reaches the goal.

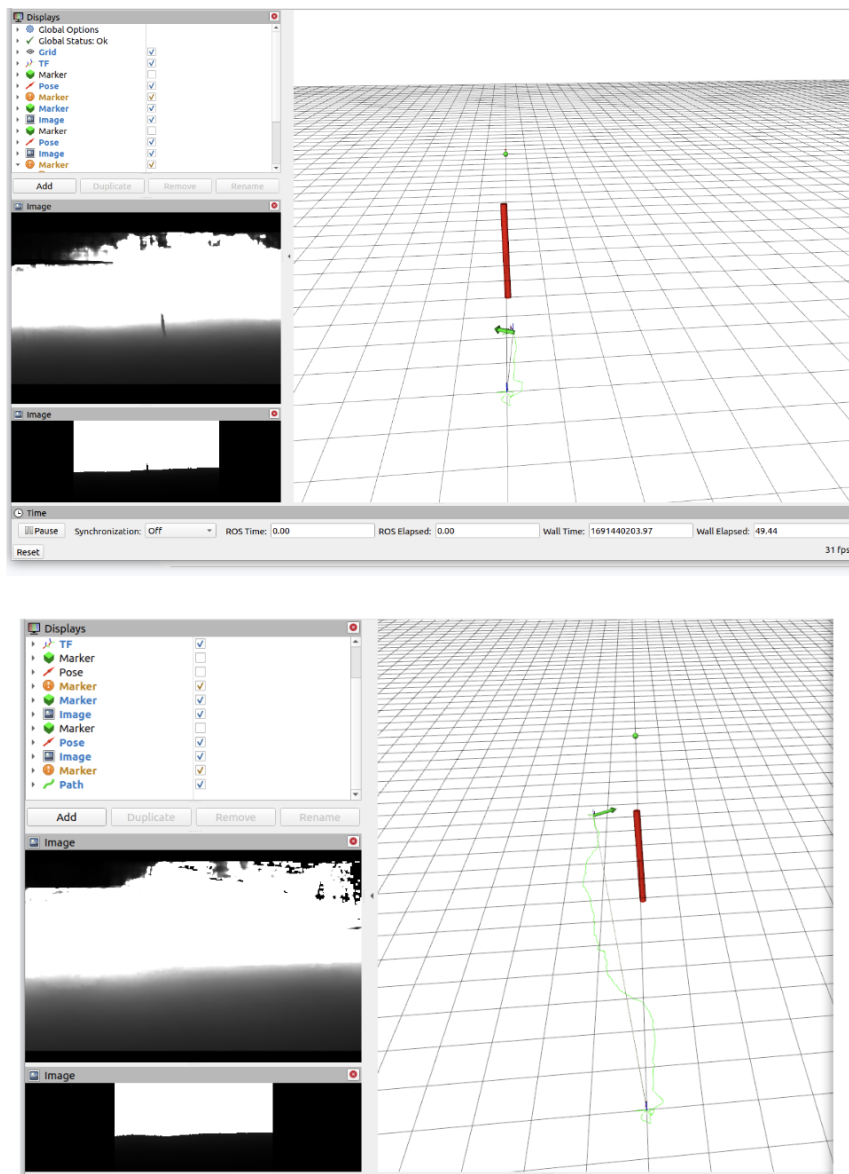


Figure 3.5: Policy outputs visualized. Top, the drone is directly in front of the red obstacle, so the policy directs the drone to fly left and around the obstacle. Bottom, the drone has passed the obstacle, so the policy outputs a velocity to head back toward the green goal. The trajectory is formed by the ZED2i’s real-time position estimation.

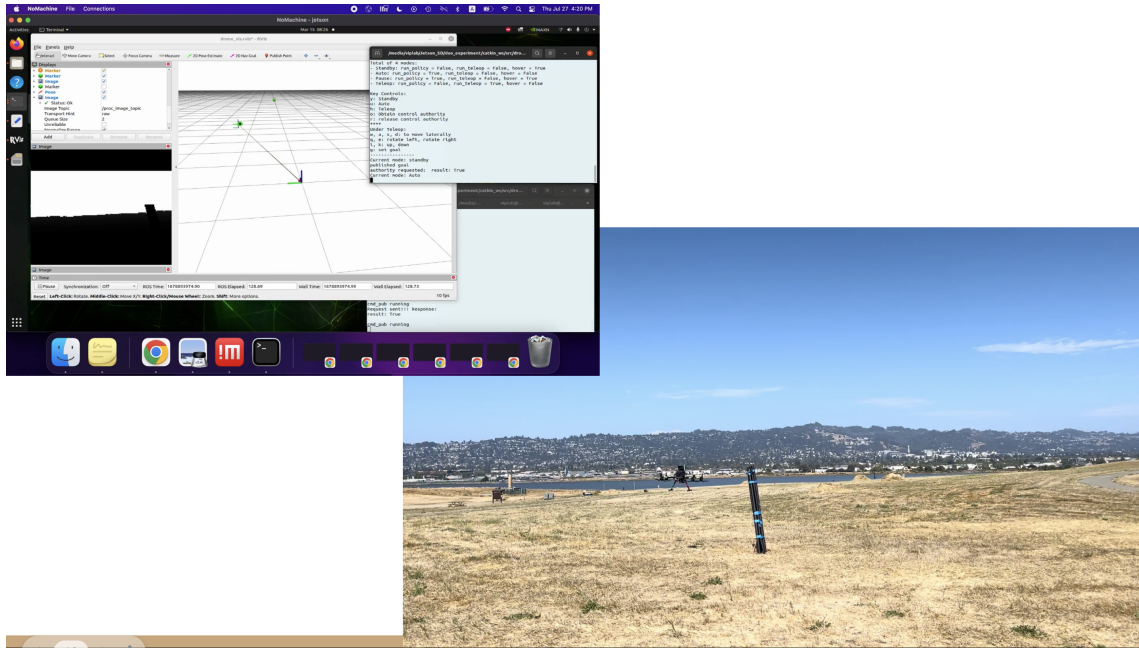


Figure 3.6: Comparing real-world footage to the real-time visualization. On the left is the RViz visualization with the latest image and drone position according to the ZED2i. On the right is the real world video, showing the drone to the left of the obstacle. The full video can be found [here](#), at 2x speed.

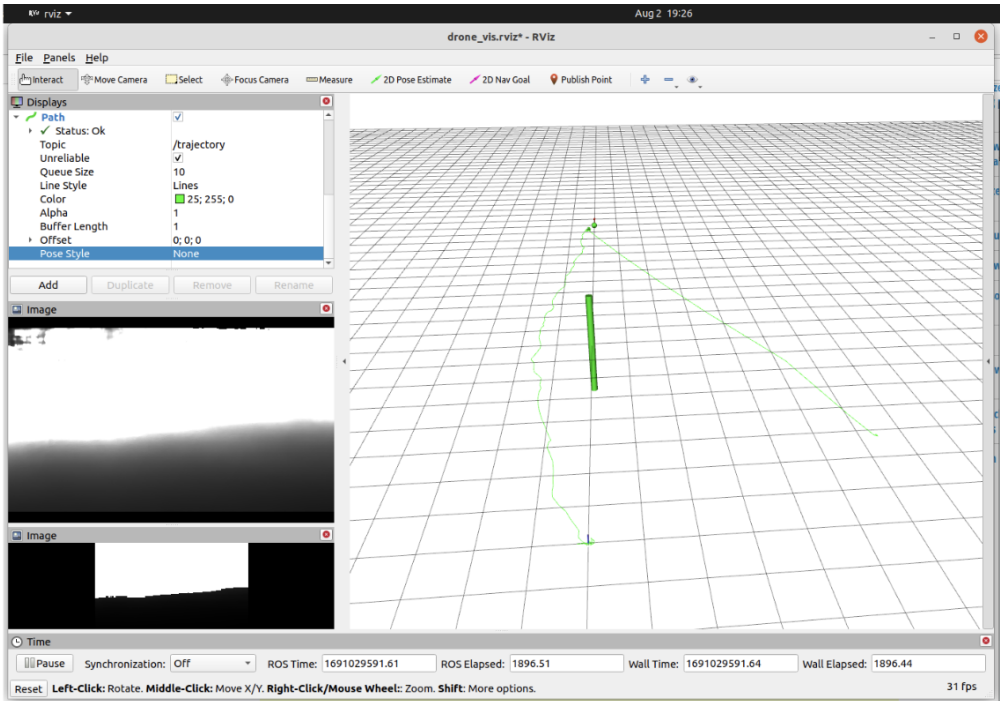


Figure 3.7: The full trajectory of a real-world flight test based on the ZED2i’s position estimation, visualized in RViz.

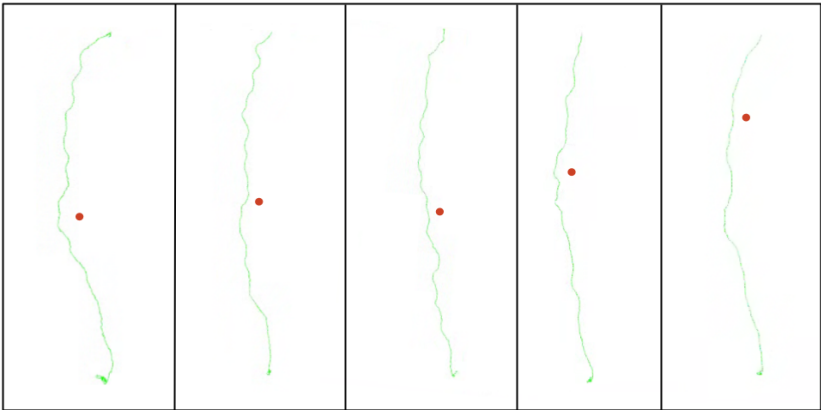


Figure 3.8: Some of the successful trajectories where the drone flies around an obstacles and reaches the desired goal.

We also wanted to test the policy’s ability to navigate, so instead of giving it a simple goal straight ahead, we gave it a goal that was directly behind the drone. Such a trajectory was not included in the training set, and showcases the benefits of using an RL agent, which really understands the dynamics of the world and can interpret errors in positions and orientations. The drone immediately did a u-turn as shown in Fig. 3.9, and then went to the goal, showcasing its ability to generalize to new types of trajectories.

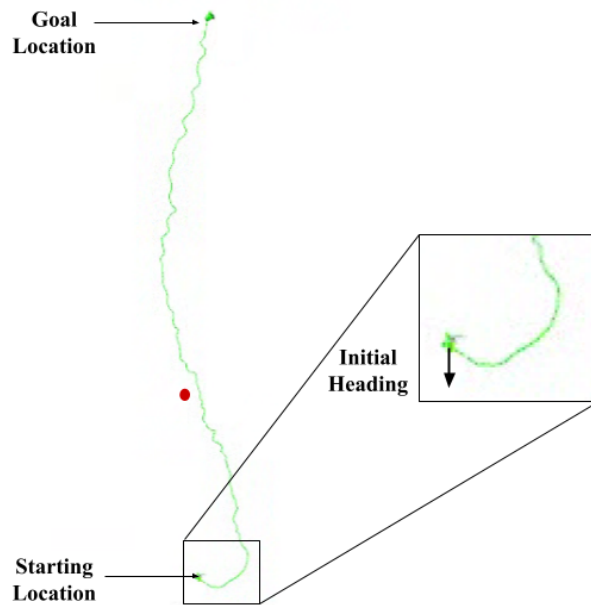


Figure 3.9: A physical test where the goal was set behind the drone. The drone did a u-turn and then went toward the goal, highlighting the policy’s ability to adapt to trajectories that were not seen in the training set.

The results are generally positive, but we must discuss the reasons for failure. Trajectories from failed tests can be seen in Fig. 3.10, where the drone gets too close to an obstacle. The first failed attempt came at 0.5m/s with one obstacle. In this test, we noticed the propellers blocking the camera’s field of view, resulting in the image seen in (b) from Fig. 2.9. The drone flew too close to the obstacle and a human operator took over control. After applying the mask seen in (c), the remaining tests at 0.5m/s worked successfully 4/4 times and the mask was used for all future tests.

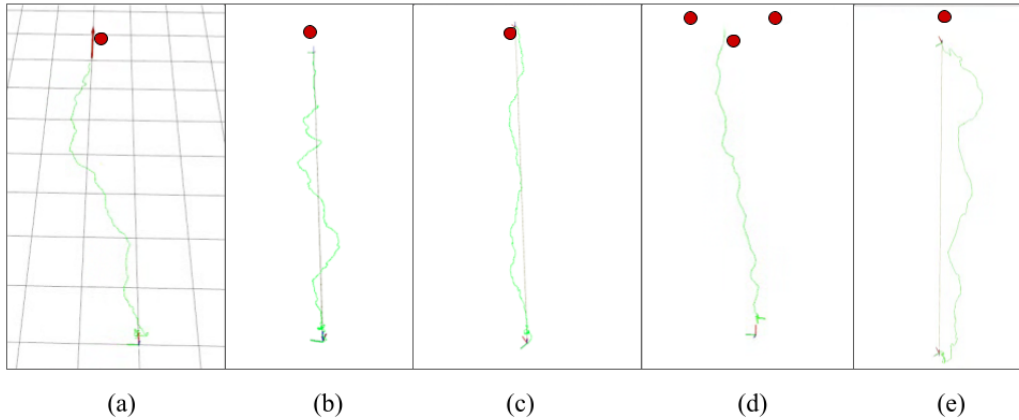


Figure 3.10: The failed trajectories where the drone flew too close to an obstacle. Reasons for failure include high gusts of wind limiting the drone’s desired movement, and a lack of temporal and spatial memory in the policy.

Failures at 1m/s with 1 obstacle happened for a couple reasons. In one case, the ZED2i’s state estimation mistracked its position, so we stopped the run prematurely. In the other two failed attempts the drone visibly rotated away from the obstacle, indicating an attempt to avoid the obstacle, but the drone barely moved forward in that direction. These trajectories are seen in (b) and (c) of Fig. 3.10, and in both cases minimal lateral movement is seen near the obstacle. We believe this happened due to strong wind, where the drone’s attempt to fly was thwarted by the wind gusting at around 15 mph. Since we constrain the drone to fly in the direction it is facing as described in Section 2.4, when the drone rotated away from the obstacle, it should have flown in that direction past the obstacle. However, the wind stopped it from going past the obstacle and it remained in front of the obstacle. Moreover, since the drone rotated away, it could no longer see the obstacle in its field of view. This meant subsequent policy outputs wanted it to turn back toward the goal, which took it into the obstacle. Discussion of how to fix this issue is in Section 4.2, including using an LSTM to remember recent images and obstacles that may not currently be in the field of view of the camera.

Finally, we tested the drone’s ability to navigate through three obstacles, where it was meant to veer out of the way multiple times to get to the goal. It had a

success rate of 0/1 due to its low speed and lack of temporal and spatial memory. The trajectory is in (d) in Fig. 3.10. The drone saw the first obstacle and turned left to go around it. After turning left, this obstacle was no longer in the camera’s field of view. Before the drone fully passed the first obstacle, it saw the second obstacle and turned right to avoid it. However, turning right made it veer back into the path of the first obstacle, which it could not see due to its limited field of view. After turning right, the first obstacle again became visible and it again turned left to avoid it. This process was repeated a few times as seen in (d) of Fig. 3.10 until one of the right turns took it too close to the first obstacle and a pilot had to manually take over control of the drone. Discussion of potential solutions for this issue are in Section 4.2, including using a 360 degree camera to know the whereabouts of all obstacles in all directions, and incorporating an LSTM with a sequence of past images, so the model remembers obstacles it saw in its recent history.

3.3 Further Analysis

A few metrics are calculated to analyze the performance of the policy in the Flightmare simulator and in the real world, to characterize sensor performance and compare the effectiveness of the velocity controller at following policy outputs.

Since the policy is trained to output velocities, which we leave to a lower level controller to follow to the best of its capabilities, we first compare the desired policy’s outputs to the actual trajectory flown. For policy outputs, we plot a theoretical trajectory that assumes all velocities are perfectly and immediately tracked. To calculate the trajectory flown, we use the simulated position of the drone in Flightmare, and GPS data in the real-world. Fig. 3.11 shows these results in simulation. On the left is the theoretical trajectory formed by the output velocities, where each velocity is applied for exactly 0.125s. On the right is the trajectory flown by the drone, where Flightmare’s simulated controller follows the given velocities from the policy. Since its in simulation, the two trajectories are nearly identical, and the drone reached within 0.25m of the green goal. Both trajectories are a top-down view.

Fig. 3.12 shows this same comparison for a real world flight at 1m/s, comparing the output velocities to the GPS trajectory.

The overall path flown between the two images is somewhat similar, but the two are not nearly as well aligned as in simulation. There are a few reasons for this. First, there is a slight delay between when the policy sends an output and when the controller actually enacts it. This is noticeable when the first turn to the right happens slightly later in the real path than on the theoretical path. Unfortunately this delay cannot be quantified because we do not have access to the internal DJI velocity

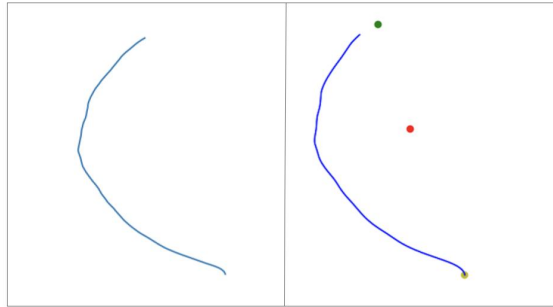


Figure 3.11: Comparing policy outputs versus the trajectory flown in simulation. Left, the theoretical trajectory assuming all policy output velocities are perfectly and immediately tracked. Right, the path flown from start (yellow) to goal (green), while avoiding the obstacle (red). Both trajectories are extremely closely aligned, as is expected from an ideal simulation environment.

controller. Next, the controller cannot perfectly track desired velocities. This happens for a few reasons: consecutive velocities are not continuous and the controller may struggle to counteract external forces such as wind gusts. Discontinuous velocities cannot feasibly be tracked because of acceleration and jerk constraints, requiring some amount of time to transition between two desired velocities. Additionally, external forces are challenging to counteract and dependent on drone dynamics and the internal velocity controller. We noticed gusts of 15mph during our tests, and sudden high speed gusts would prevent the drone from flying in its desired direction. The main conclusion from this comparison, however, is that the policy outputs tended to be noisy – rather than a smooth path around the obstacle as seen in simulation, the policy kept switching between turning left and right. This likely happened because when the drone would turn to the left to avoid an obstacle, it could then no longer see the obstacle, and therefore would turn to the right to go back toward the goal. Discussion of how to fix this issue is seen in Section [4.1](#).

Next, we want to characterize the ZED2i’s state estimation accuracy. Our architecture relies on the stereo camera’s visual odometry to estimate position and orientation which are passed to the policy. GPS is not used, meaning our system can be used in more complex GPS denied environments. However, visual odometry is less accurate than GPS. Since positional data during training is perfect, we want to compare our physical state estimation against GPS data. Fig. [3.13](#) compares visual odometry’s computed trajectory against the GPS trajectory for one of the test flights at 1m/s with one obstacle. While GPS data is also noisy and imperfect, since

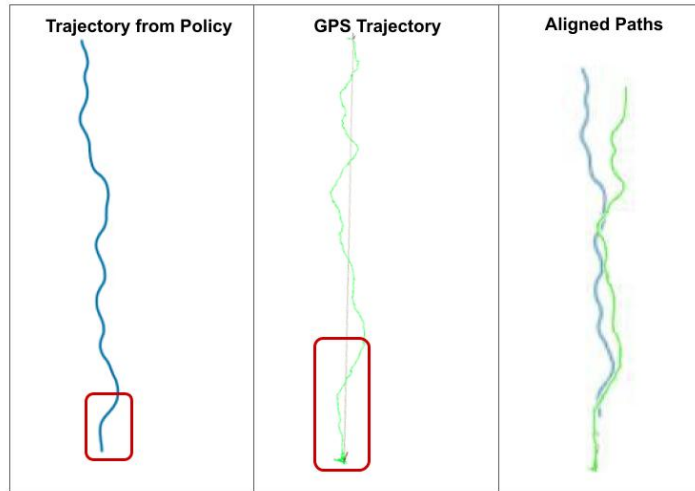


Figure 3.12: Comparing policy outputs versus the GPS trajectory flow in the real world. Left, the theoretical trajectory assuming all output velocities are perfectly and immediately tracked. Middle, the path flown according to GPS data. The first right turn comes later in the GPS trajectory than the policy outputs, showing a delay between when outputs are sent and when they are applied. Right, an attempt to align the two trajectories to show the controller achieves reasonable performance at matching corresponding turns. Errors come from the controller’s inability to perfectly track desired velocities. Additionally, since velocities are naively integrated to form the theoretical trajectory, errors propagate. These reasons lead to the two paths not being as well aligned as in simulation.

we cannot set up a proper motion tracking system outdoors, GPS is used as ground truth for this analysis. GPS data was measured at 50Hz, and state estimation was run at 20Hz. Results show that on average, visual odometry position data is similar to GPS position. However, it occasionally sees slight jumps in position, which are corrected for shortly. For example, in the right half of Fig. 3.13 which corresponds to the ZED2i’s estimated trajectory, there is an initial jump to the right just before the drone starts to move forward. This jump is not seen in GPS data, which simply goes straight forward. During this jump, the policy’s input position would have some error. Soon after, however, the ZED2i corrects for that positional error by jumping back to the left, and returning close to the GPS position. Subsequent input positions have lower error. This shows the ZED2i is somewhat capable of accounting for discrepancies and correcting errors from previous timesteps.

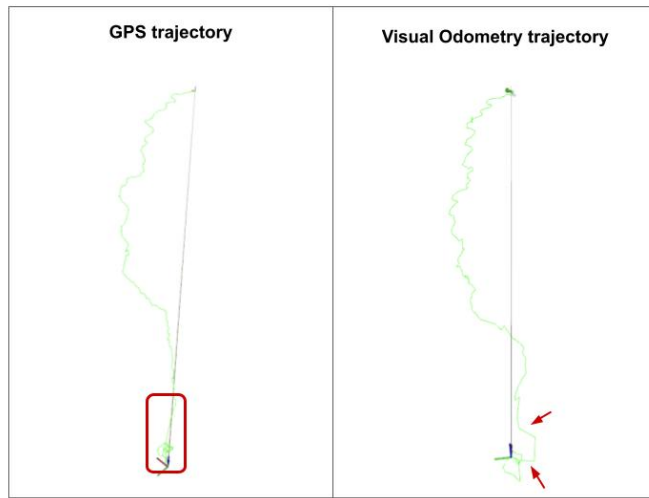


Figure 3.13: Comparing GPS data (left) against the ZED2i’s visual odometry based state estimation (right) to evaluate the stereo camera’s performance. Overall, trajectories are similar, but visual odometry suffers from some noise. Notably, a jump to the right is seen initially when the drone starts flying forward. This jump is quickly corrected for with an opposite leftward jump briefly after the drone flies forward.

According to GPS, the total displacement between the start and the goal positions in this flight was 19.24m. Visual odometry’s displacement shows 19.78m, resulting in an error of 2.8%. The total distance traveled during the trajectory is calculated by adding the differences in position between every timestep, aggregating many small steps taken over time. Since GPS data is noisy, its positions were smoothed with a window size of 5 to reduce high frequency noise. Using this method, GPS data showed a trajectory distance of 25.14m, while visual odometry showed 24.96m. An error of 0.7% shows that on average errors in visual odometry’s position estimates are low.

Finally, the ZED2i’s orientation estimate is compared against the drone’s IMU. Fig. 3.14 shows the two are extremely well aligned.

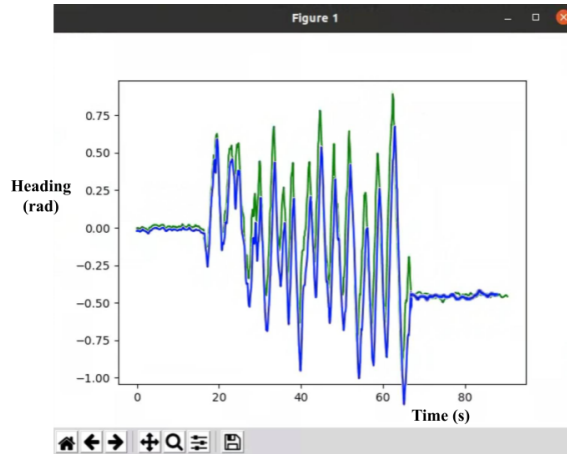


Figure 3.14: Comparing heading angles over time between the ZED2i (green) and the drone’s IMU (blue). Heading from the ZED2i aligns well with the drone’s IMU.

The policy we tested in the real world is trained at a max speed of 1m/s and tested at max speeds of 0.5m/s, 1m/s, and 1.5m/s. It works well at both 0.5 and 1m/s as seen in Table. 3.1, but at 1.5m/s, it is very jittery. This is likely because the policy sends discontinuous velocities at high frequency, resulting in high jerk on the platform. We did not save velocity, acceleration, or jerk data, but the positional trajectory can be seen in (e) of Fig. 3.10. After seeing this, we trained a new policy in simulation to see if higher velocities can be flown and we analyze the effects of testing a model at a different speed than during training. Table. 3.2 summarises the results for simulated flight at 1m/s and 3m/s, with a sample size of 5,000 used to evaluate each policy. Velocities were increased by scaling policy outputs by the desired maximum speed. Policy outputs are effectively percentages of the maximum speed. A model trained at 3m/s does well when being evaluated at both 3m/s and 1m/s, and a model trained at 1m/s does well when evaluated at 1m/s but poorly when flying at 3m/s.

We expect this to be because a model trained at a low speed learns to react slowly because it has more time to avoid an obstacle. Testing it at a higher speed means the drone does not realize it needs to react more quickly. Similarly, training at high speeds will still do well at low speeds, because its reaction time is faster and better than it is used to. By passing the current velocity to the network, the policy could learn to adapt on the go and optimize its actions based on its current velocity. However, for this work, velocity is not part of the state space.

Next, we increased the frequency of running the policy to see how well it can

	Trained @ 1m/s	Trained @ 3m/s
Tested @ 1m/s	Reached Goal: 100.0% Crashed: 0% Out of Bounds: 0%	Reached Goal: 99.2% Crashed: 0.08% Out of Bounds: 0.72%
Tested @ 3m/s	Reached Goal: 7.6% Crashed: 10.6% Out of Bounds: 81.8%	Reached Goal: 90.6% Crashed: 2.5% Out of Bounds: 6.9%

Table 3.2: Training models flying at 1m/s and 3m/s, and evaluating them at both 1m/s and 3m/s to compare performances, all in simulation. Results show good performance is achieved when evaluating at the same or lower speed as training, and poor results stem from training at high speeds but evaluating at lower speeds. A sample size of 5,000 was used when evaluating all models.

do in simulation if it has extremely fast reaction times. The physical platform was run at 8Hz and 1m/s, but running at higher frequency should allow flight at higher speeds. Table. [3.3](#) summarises these results. Increasing the frequency at a fixed velocity performs poorly. This is hypothesized to be because it is harder for an RL policy to converge when the differences in state between timesteps are small, and every action taken has a very small effect. Increasing both the speed and frequency performs better, and would potentially improve with some reward function tuning.

Speed	Frequency	Results
1 m/s	10Hz	Reached Goal: 100% Crashed: 0% Out of Bounds: 0%
1 m/s	20Hz	Reached Goal: 0% Crashed: 0.8% Out of Bounds: 99.2%
2 m/s	20Hz	Reached Goal: 83.4% Crashed: 16.6% Out of Bounds: 0%

Table 3.3: Results at different flight speeds and policy frequencies in simulation. A model was optimized to fly at 1m/s at 10Hz in simulation, reaching the goal every time. A new policy was then trained at 20Hz, which learned to avoid crashes but not reach the goal. Doubling the flight speed to 2m/s resulted in reasonable performance. This pattern shows that simply increasing the frequency may not improve results. We hypothesize this to be because at low speeds and very small timesteps, all actions have a similar effect – only after multiple timesteps does it become clear that an agent is turning left rather than going straight. Increasing the speed to 2m/s countered this increase in frequency, so each action again has more distinct effects. Again, maximum speeds were increased by scaling policy outputs by the desired maximum speed. Note that reward function and hyperparameter tuning was only done for 1m/s and 10Hz, so more tuning should be done with the other speeds and frequencies.

Chapter 4

Conclusion

In Section 4.1, we discuss the benefits and drawbacks of our approach, and in Section 4.2, we describe some future avenues to improve this work in various ways.

4.1 Discussion

In this work, we presented a deep Reinforcement Learning policy that takes in a depth image, drone state information, and a goal position, and outputs x and z linear velocities and a yaw rate to reach the given goal while avoiding obstacles. Simulation results prove very effective at safely reaching goal positions in a simple environment. In the real world, we see similarly positive results in an environment with a single obstacle. The drone is able to reach goal locations while avoiding the obstacle. Fail cases come during moments of 15mph wind gusts, where the drone visibly attempts to fly away from the obstacle, but cannot overcome the force of the wind.

This approach works due its 2 part approach combining imitation learning from a reference teacher and reinforcement learning to explore various actions given various inputs. It combines the best of two worlds. The reference teacher distills optimal trajectories by incorporating it into the reward function, and the agent's exploration lets it understand the dynamics of the world.

With pure imitation learning on optimal trajectories, the dataset is of utmost importance. The multi-modal nature of viable trajectories must be dealt with, or the model will be confused between multiple options that are equally viable – for example, going left or right around a tree. Moreover, there is a distribution of trajectories that are all viable that are hard to differentiate between. [11] predicts trajectory costs and then follows the minimum cost trajectory. In this work, the

”optimal” trajectory is merely used as reference – large deviations result in large penalties, but deviations are still allowed – and reaching the goal by any means still results in a large reward. This inherently creates a distribution of valid trajectories as each of them results in similarly high reward from reaching the goal without crashing, without forcing a specific trajectory as exceptionally better than others.

Additionally, with imitation learning the policy does not learn real-world dynamics. This means its training dataset must include a comprehensive set of trajectories, and it is important to note that different types of missions will have different navigation scenarios. For search and exploration applications, a goal ahead of the drone is usually passed in, assuming the drone has already explored the region directly behind it. However, for missions where previously explored regions must be revisited, there may be times when a u-turn is needed to get back to the right path. Such u-turns may not be a big part of a dataset meant for exploration or infinite-horizon missions, and pure imitation learning approaches may get confused if the goal is suddenly behind it. With an RL agent however, many types of states are explored, and world dynamics are learned. This means the RL agent can generalize to new navigation tasks not included in the training dataset, such as the u-turn discussed above.

Unlike [11], this work does not focus on highly agile flight or complex maneuvers. In this work, desired velocities are sent at 8Hz, wherein every 0.125 seconds the drone follows a fixed velocity. With trajectory tracking in [11], new trajectories are output at 24.7 Hz, and new desired thrust and body rates are sent at 100 Hz. Complex flight maneuvers are possible, and multi-timestep trajectories mean flightpaths are smooth rather than jittery. In the next section we discuss how this work can be improved to have similar smooth paths and plan for longer-term complex maneuvers rather than single-timestep discontinuous velocities.

4.2 Future Work

There are various avenues this work could improve on in the future. The most immediate path forward focuses on using the same hardware and software architecture, but re-training the policy in more realistic and complex environments to ensure the existing system can be run in dense forests, mountains, and near buildings at higher speeds. Once the policy works well in complex environments, there are many paths this work could follow, each with different focuses. To focus on running this work on more types of platforms with less payload and compute capabilities, a switch from a stereo camera to a cheap RGB camera could be made. To perform more complex flight maneuvers, a 360 degree camera could be added to give the drone full vision of its surroundings. To deal with more complex environments, an LSTM can be added

to give the policy memory of nearby obstacles, even if they are not in view. Each of these avenues is now discussed further.

The main next step would be to re-train the existing policy in multiple environments with more visual features. The current algorithm is trained in a simple environment with cylinders, and tested on a similarly simple environment with cylindrical obstacles made out of pool noodles. Since testing with multiple obstacles has mixed results due to flight speeds and wind, the natural next step would be to re-train in more complex and realistic environments such as forests or outdoor areas with buildings. Positive results in a real forest would prove the worth of this work, as it would showcase real-world flight in a complex environment requiring flight along all three dimensions, and weaving through dense forest would prove that this could compete with expert human pilots.

Additionally, another next step would be to replace the stereo ZED2i camera with a lightweight RGB camera, and use regular RGB images for state estimation. A simple learning based depth estimation model could be run on these RGB images to estimate relative depth, and those predicted depth images passed to the RL agent. RGB cameras are cheaper and lighter weight than a stereo camera, and require much less computation. By reducing compute and weight requirements, we can then run the algorithm on smaller drones with lower payload capacities.

Next, since this work constrains the drone to only fly in the direction the camera is pointing, by replacing the existing camera that has a limited field of view with a 360 degree camera, we can achieve full 360 degree non-planar flight. A 360 degree depth prediction model such as 360 MonoDepth [19] can be used to convert 360 degree RGB to depth, and this can be used to give the drone a full view of the environment around it. Now, we no longer need to constrain the quadrotor's dynamics to only fly forward, and can take full advantage of the drone's 3D flying capabilities. This would allow the system to do more complex maneuvers in complex environments, and fly at much higher speeds. Currently, the platform must yaw to turn, which greatly impacts the maximum speed it can fly, because it cannot see to the left, right, or behind the platform. However, with a 360 degree field of view, the drone can fly at high speeds in all directions, without worrying about obstacles unexpectedly showing up.

Other potential improvements include adding an LSTM. The current setup relies on using the latest depth image and state information, with no memory of any past information. The benefit of this is that there are no memory requirements, but a major drawback is that the agent does not remember the layout of obstacles nearby. As soon as an obstacle goes out of view, even if it is right next to the drone, the policy outputs act as if there is nothing there. Using a 360 degree camera could help with this, but another solution would be to incorporate an LSTM, where the agent

may have some local memory of nearby objects.

In general, this work proves the usefulness of combining an exploratory deep RL agent with a privileged supervisor, leading to optimistic results in a simple environments. There are countless opportunities of improvement that are not in the scope of this work, but that have exciting promise.

Bibliography

- [1] Karthika Balan and Chaomin Luo. “Optimal Trajectory Planning for Multiple Waypoint Path Planning using Tabu Search”. In: *2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 2018, pp. 497–501. DOI: [10.1109/UEMCON.2018.8796810](https://doi.org/10.1109/UEMCON.2018.8796810).
- [2] Ghulam Farid et al. “Modified A-Star (A*) Approach to Plan the Motion of a Quadrotor UAV in Three-Dimensional Obstacle-Cluttered Environment”. In: *Applied Sciences* 12.12 (2022). ISSN: 2076-3417. DOI: [10.3390/app12125791](https://doi.org/10.3390/app12125791). URL: <https://www.mdpi.com/2076-3417/12/12/5791>.
- [3] V. Jeauneau, L. Jouanneau, and A. Kotenkoff. “Path planner methods for UAVs in real environment”. In: *IFAC-PapersOnLine* 51.22 (2018). 12th IFAC Symposium on Robot Control SYROCO 2018, pp. 292–297. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2018.11.557>. URL: <https://www.sciencedirect.com/science/article/pii/S2405896318332634>.
- [4] John Schulman et al. “Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization”. In: June 2013. DOI: [10.15607/RSS.2013.IX.031](https://doi.org/10.15607/RSS.2013.IX.031).
- [5] Lingli Yu et al. “An Optimization-Based Motion Planner for Car-like Logistics Robots on Narrow Roads”. In: *Sensors* 22.22 (2022). ISSN: 1424-8220. DOI: [10.3390/s22228948](https://doi.org/10.3390/s22228948). URL: <https://www.mdpi.com/1424-8220/22/22/8948>.
- [6] Steven M. LaValle. “Rapidly-exploring random trees : a new tool for path planning”. In: *The annual research report* (1998). URL: <https://api.semanticscholar.org/CorpusID:14744621>.
- [7] Omid Esrafilian and Hamid D. Taghirad. “Autonomous Flight and obstacle avoidance of a quadrotor by Monocular Slam”. In: *2016 4th International Conference on Robotics and Mechatronics (ICROM)* (2016). DOI: [10.1109/icrom.2016.7886853](https://doi.org/10.1109/icrom.2016.7886853).

- [8] L. Nieto-Hernandez, Angel A. Gomez-Casasola, and H. Rodriguez-Cortes. “Monocular Slam position scale estimation for Quadrotor Autonomous Navigation”. In: *2019 International Conference on Unmanned Aircraft Systems (ICUAS)* (2019). DOI: [10.1109/icuas.2019.8797951](https://doi.org/10.1109/icuas.2019.8797951).
- [9] Yunlong Song et al. “Learning perception-aware agile flight in cluttered environments”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)* (2023). DOI: [10.1109/icra48891.2023.10160563](https://doi.org/10.1109/icra48891.2023.10160563).
- [10] Patrick Wenzel et al. “Vision-Based Mobile Robotics Obstacle Avoidance With Deep Reinforcement Learning”. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 14360–14366. DOI: [10.1109/ICRA48506.2021.9560787](https://doi.org/10.1109/ICRA48506.2021.9560787).
- [11] Antonio Loquercio et al. “Learning high-speed flight in the wild”. In: *Science Robotics* 6.59 (2021). DOI: [10.1126/scirobotics.abg5810](https://doi.org/10.1126/scirobotics.abg5810).
- [12] Yunlong Song et al. “Flightmare: A Flexible Quadrotor Simulator”. In: (2021). arXiv: [2009.00563 \[cs.R0\]](https://arxiv.org/abs/2009.00563).
- [13] Yunlong Song et al. “Autonomous Drone Racing with Deep Reinforcement Learning”. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2021). DOI: [10.1109/iros51168.2021.9636053](https://doi.org/10.1109/iros51168.2021.9636053).
- [14] Sikang Liu et al. “Search-based motion planning for aggressive flight in SE(3)”. In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 2439–2446. DOI: [10.1109/lra.2018.2795654](https://doi.org/10.1109/lra.2018.2795654).
- [15] Robotics and University of Zurich Perception Group. *DodgeDrone Vision-based Agile Drone Flight (ICRA 2022 Competition)*. 2022. URL: https://github.com/uzh-rpg/agile_flight.
- [16] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: [1707.06347](https://arxiv.org/abs/1707.06347). URL: <http://arxiv.org/abs/1707.06347>.
- [17] Wenhao Yu et al. “Visual-Locomotion: Learning to Walk on Complex Terrains with Vision”. In: *Proceedings of the 5th Conference on Robot Learning*. Ed. by Aleksandra Faust, David Hsu, and Gerhard Neumann. Vol. 164. Proceedings of Machine Learning Research. PMLR, 2022, pp. 1291–1302. URL: <https://proceedings.mlr.press/v164/yu22a.html>.
- [18] Andrew Howard et al. “Searching for MobileNetV3”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019). DOI: [10.1109/iccv.2019.00140](https://doi.org/10.1109/iccv.2019.00140).

- [19] Manuel Rey-Area, Mingze Yuan, and Christian Richardt. *360MonoDepth: High-Resolution 360deg Monocular Depth Estimation*. 2022. arXiv: [2111 . 15669](https://arxiv.org/abs/2111.15669) [[cs.CV](#)].

Chapter 5

Appendix A: Additional Figures

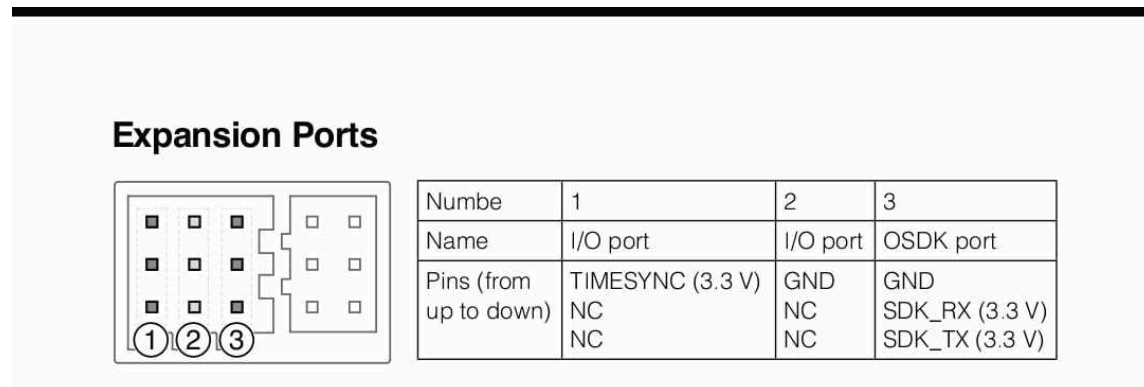


Figure 5.1: The ports on the DJI Expansion Module. Use a USB-TTL cable to connect the Jetson to the expansion board.