ML CONTROLLERS WITH MEMEORY FOR ROBUST QUADROTOR CONTROL AND RESEARCH

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Nicholas Enrique Navarrete

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Advisor: Dr. Nicola Elia

February, 2026

ACKNOWLEDGMENTS

I want to acknowledge and give my sincere gratitude to Professor Nicola Elia. The help and experience that Professor Nicola Elia has given me has been pivotal to my research. I hope that the work we did here will be useful for educating future researchers and make their lives easier. Thank you for your patience and guidance.

I would also like to acknowledge Professor Andrew Lamperski for his help and wisdom in regards to Machine Learning and control. This has been invaluable to me, allowing me to avoid many rabbit holes and continue my research. Thank you.

DEDICATIONS

This work is dedicated to Anna Baizley, who has supported me through all my studies. Without her I would not have been able to accomplish both research and work. She has kept me motivated during my highs and lows. I hope to be half the partner she has been to me.

Thank you.

Abstract

Over time, Non-Classical controllers are becoming more common and the interest of much research. Neural network controllers are some of the most common. These Neural Network controller offer advantages such as higher efficiencies, ability to learn nonlinear models, and adaptability. Designing such controllers typically relies on optimization and reinforcement learning methods, such as PPO or DDPG learning. These learned controllers require many action-time steps, and performing many of these actions directly while learning can result in undesirable effects on the real system. Undesirable effects may include hardware damage, behaving unsafely or the quadrotor crashing. In order to collect the required data over many action/time steps while also avoiding undesirable system effects, leaned controllers are typically learned on simulated systems. These simulations usually utilize different physics engines such as Bullet, NVIDA PhysX, or Jolt Physics. While these simulations are reasonably good, they are not perfect analogs of our real system; therefore, our controller can learn undesirable behaviors from these imperfect simulations.

In this thesis we will explore different control architectures and ML control methods for a quadrotor, and in doing so develop a framework for testing and evaluation of non-classical controllers for future SimToReal research. The system we will control a Crazyflie 2.X quadrotor drone, and its pose will be measured by the VICON motion capture system. We will be controlling the system with a neural network controller which is learned from a pyBullet simulation using the OpenAI Gym framework. Many different training methods and control algorithms are created, tested, and their robustness compared. These methods were all implemented into a real system and their effectiveness compared empirically. The most robust model was a RNN of multi-layer perceptron using Long Short Term Memory (LSTM) units. The resulting system enables users to define different control algorithms and control a quadcopter remotely. Allowing for continued research in neural and non-classical control methods for aerial robotics.

Contents

List of Figures

List of Tables

1. Literature Review

There has been extensive previous work on direct control of a quadrotor using reinforcement learning [1]. Steven L. Waslander and Gabriel M. Hoffmann show that using reinforcement learning, it is possible to find a (linear gain) policy to hover a quadrotor. Though they show that a integral sliding mode controller can more smoothly control the quadrotor. Kownacki and Romaniuk [2] show that multi layer perceptrons are capable of controlling a quadrotor, and that a 2 layer MLP has excellent training performance, beating both a 1 layer MLP and a DNN. In the work by Jack F. Shepherd III and Kegan Turner [3] an MLP-like network is used to control the attitude and position control of a quadrotor using cascaded controllers which is a form of direct control. Their work uses simulation to have the network learn to copy the response of a hand-tuned PID controller and simulate the controller. This process is shown to work well in simulation. The work by Dunfied and Tarbouchi [4] uses data gathered from a manual controller to learn a 2-layer MLP, and fly a real quadrotor using that learned network, the network commands rotations for an internal attitude control to realize.

Modern methods include using online reinforcement learning to fly quadrotors. The work by Hwangbo [5] uses online reinforcement learning to fly quadrotors through non-ideal flight paths such as starting from a throw. Their controller commanded thrust values which was fed into another non RL controller to realize. They were successful in stabilizing a thrown quadrotor but experienced higher than expected positional errors. There is a gap in literature for controlling the quadrotor's pose by controlling the motor PWM using a 2-layer MLP. Working on this problem will also allow for a framework for future research into direct control of a quadrotor. The work by Edhah and Mohamed [6] uses offline learning to learn a DNN for control, this method was shown to perform similarly to an LQR controller in simulation. Offline learning is especially beneficial for systems which can crash and cause dangerous situations, like a quadrotor. Work on a simulator of the Crazyflie 2.X for offline learning was done by Jacopo Panerati in [7]. The simulator uses OpenAI Gym API and the PyBullet physics engine to simulate the quadrotor. The simulator also uses previously identified system properties of the Crazyflie 2.X. This previous work can be leveraged to learn offline controllers to learn policies to control a real quadrotor.

2. Introduction

The motivation for this work is to explore different control architectures and create empirically more robust MLP based controllers. This would allow future students to perform their own research on quadrotor control using real systems and simulations and hopefully shorten the SimToReal gap. As it stands now there is no published open-source framework which links simulation with motion capture and a real quadrotor. SimToReal is an incredibly hard task, with many pitfalls and likely requiring extensive time and research, both of which are outside the timeframe of this thesis, so instead methods for designing more robust ML controllers will be explored to make closing the SimToReal gap easier.

Most literature focuses on cascaded control for the quadrotor. There is usually one ‘fast’ controller controlling the attitude of the quadrotor using the motor rpm, and a ‘slow’ higher level controlling the position of the quadrotor using the attitude. This setup has been implemented before on our real system using PID control. Instead, it may be possible to learn a ‘direct’ controller which can directly control position using the motor rpm. This would help greatly with simplifying the control architecture for future researchers and allows for the learning of only 1 controller instead of having to learn 2 controllers.

To create a more accurate simulation physical aspects of the system were modeled, such as the coefficients of thrust and the observation noise. The observation noise was quantified over different positions and camera frequencies, the observation noise was then modeled as gaussian random variables. To model the coefficients of thrust was a challenge, the thrusts vary greatly with battery voltage and are variable, so instead an ideal coefficient of thrust was chosen and a variance was added to represent the uncertainty. Along with this many improvements to the simulation were made such as adding new observation and action spaces, updating the reward algorithms, and simulation logic.

The first part of the developed framework for research is the simulation of the system, and training of controllers for the system. Figure 1 shows the flow chart for the simulation and training.

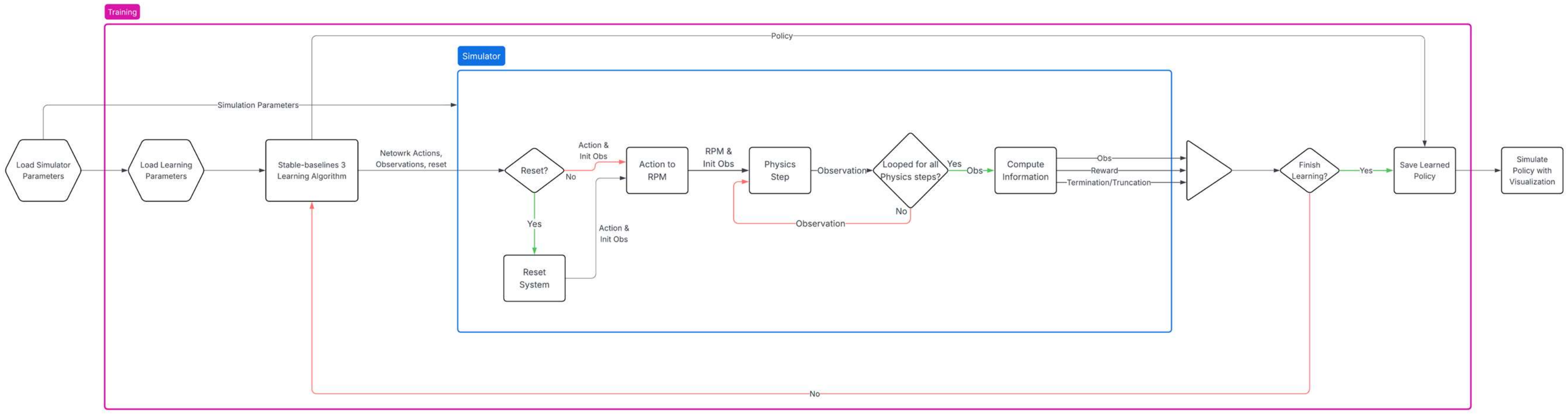

Figure 1. Simplified flow chart for the learning and simulation process used to produce a controller

This is a simplified flow chart, and the whole process can be viewed in the project GitHub repository. The simulation is done in Python using the OpenAI Gym API, which allows the simulation to be used with a variety of different ML libraries. Using the simulation (sometimes referred to as environment), Stable-baselines3 can be used to create a network for controlling the quadrotor given valid learning parameters using Proximal Policy Optimization (PPO). Using this process, controllers were learned which were able to control the simulated quadrotor smoothly and reliably.

The flow of the real quadrotor control is shown in Figure 2. The system is comprised of 3 independent systems running on different computers: The VICON motion-capture system, the controller (sometimes referred to as base-station), and the quadrotor. Communication between the motion-capture system and the controller is over the internet, and communication between the controller and the quadrotor is over radio. This communication / control scheme has previously been tested and has been shown to be capable of controlling the quadrotor position using cascaded PID control. Notably, the Crazyflie 2.x quadrotor does not come with the ability for remote direct control, so the firmware needs to be modified as shown in section 3.

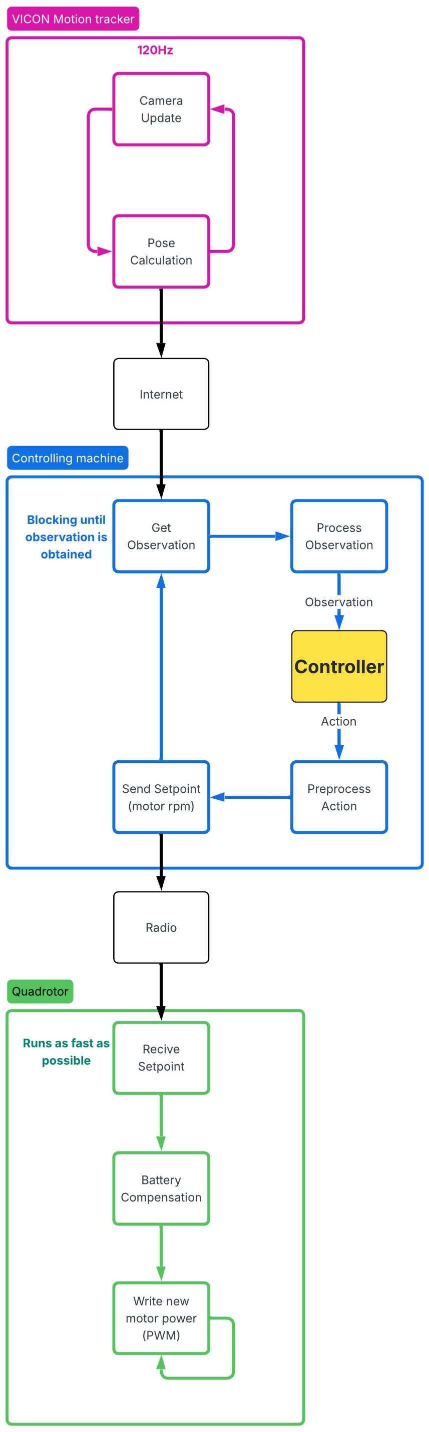

Figure 2. Flow chart for the real control of the quadrotor with each computer in a unique color

To develop the framework, we will first deal with the physical system. In the physical system section, we will first modify the quadrotor to accept setpoint commands as motor power [m1 m2 m3 m4] instead of attitude setpoints [roll pitch yawrate thrust]. The decision to directly control motor power was made for three reasons, (1) to make research and study easier with direct control, (2) allows for cascaded control to be implemented in the control machine instead of the quadrotor, and (3) to simplify the simulation. The second of the physical system section models the connection between the motion capture system and the controlling machine. The system has both the pose error and the latency measured. They were both fit into random variables, but only the pose error was included in simulation for reasons discussed in the section.

The next part of the framework is the simulation. The simulation is the most important part, especially as it enables the most research and serves as a good foundation for learning. The simulation developed by Jacopo Panerati[7] was used as a base simulation that was improved upon. The simulation uses the OpenAI Gym API for easy application of ML libraries. The observation and action spaces were both updated to improve learning, provide more options and development opportunities, and to better reflect the real system that will be controlled. The reset and some simulation logic was updated. Lastly the reward structure of the simulation was updated to create stable learning. This thesis also talks about the training settings and results in the training section, so future researchers can use these settings to learn their own controllers.

Controllers using different architectures and observation spaces were created and tested. The primary testing metric was robustness, which was tested by measuring the success rate vs action noise. It was shown that using a network with MLP + LSTM was the most robust network, and is also the only network capable of controlling the quadrotor without using an estimated hover power.

3. Physical System

3.1. Crazyflie Firmware

The Crazyflie Firmware is open source and can be found on GitHub. It has been developed to receive commands from an external radio and process them accordingly. Previous work has used this framework to pass roll, pitch, yaw-rate, and trust values into the firmware as ‘setpoints’ for an internal PID controller. This cascaded controller was robust and fast but does not allow for the control the individual motors.

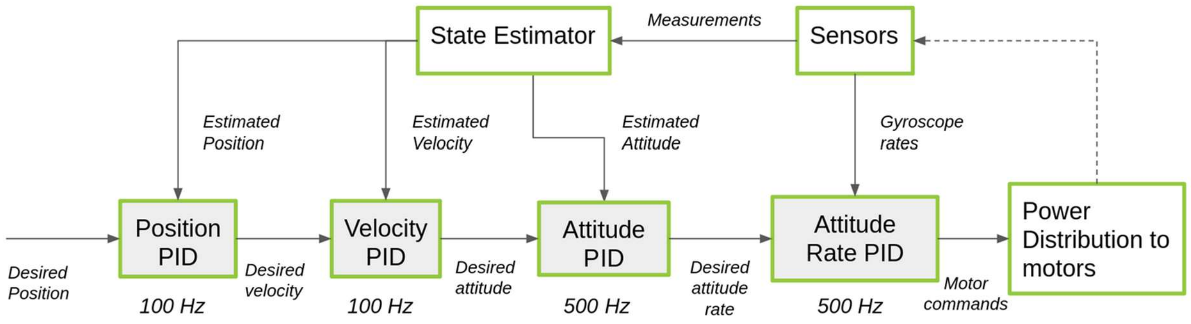

Figure 3. Crazyflie controller flow diagram showing their intended cascaded controller design (from Crazyflie website)

The external commands are sent as ‘setpoints’ these setpoints were a radio packet which contains four floating point numbers. These setpoints are received by the Crazyflie quadcopter and loaded into a FreeRTOS queue. The firmware then processes this queue before writing a value to the motors. The motors are controlled by PWM, and the firmware has 2 bytes of resolution.

The published API does not have the ability to control the motors individually using the same control frequency as the setpoints. This will cause controllability issues if we want to control the motors individually, so we need to modify the firmware to quickly write to the individual motors. From the previous work done to show that the quadcopter was controllable using the previous setpoint architecture, the easiest method to control the quadcopter would be to hijack the setpoint commands.

3.1.1. Old Crazyflie Firmware

The old Crazyflie Firmware received the data packet as a 16 bit float for roll, pitch, yawrate, and an unsigned 16 bit integer for the thrust. These values are first passed into a cascaded PID controller. The first controller controls the attitude rates of the quadcopter and the second controller controls the attitude of the roll and pitch. Then to control the motors the following equations are used to determine the motor speeds.

M1 = thrust - roll + pitch + yaw

M2 = thrust - roll - pitch - yaw

M3 = thrust + roll - pitch - yaw

M4 = thrust + roll + pitch - yaw

The motor values were then sent through a battery compensation algorithm which uses a cubic fit to compensate the thrust for battery voltage. Lastly, the motor thrusts are capped to 65535 as they are unsigned integers.

3.1.2. Updated Crazyflie Firmware

In order to control the motors individually for direct control, we had to modify the firmware of the quadcopter. Firstly, all of the inputs roll, pitch, yawrate, and thrust are changed to be 16 bit unsigned integers. The cascaded PID controllers are then set to pass the inputs as outputs. The battery compensation and the motor capping are left unchanged, as the battery compensation is beneficial, and the motor capping does not do anything in our case. After modification, the system is ready to receive the setpoint as uint_16 m1 m2 m3 and m4, instead of the original roll pitch yawrate and thrust.

3.1. Camera system

In order to observe our system we use the VICON motion tracking system. Using 10 VICON Vero cameras to measure a 6.2m by 3.7m area. In order to receive images from our system our machine running the controller needs to be connected to the same network as the machine running the VICON tracker. The controller machine sends a request to the tracker machine to receive the next frame of our system. Once the frame is received it then calculates the global position of the quadrotor and the Euler rotations. These positions and Euler rotations will be used as observations for our controller.

3.2. Pose Variance Modeling

The system is very dynamic, and needs to be accurately modeled if we wish to achieve zero-shot learning of a control algorithm. The system is not ideal and have both inaccuracies in the pose of the drone, and the frequency of measurements. In order to model the pose inaccuracies, we take ‘n’ samples of the drones pose at different locations. We can then fit a gaussian random variable to our samples. This was done for a measurement frequency of 120Hz and 240Hz in order to determine the best control frequency.

The following figures show these pose measurements and their gaussian fits for all x y z positions and roll pitch yaw rotations for varying positions in the motion capture system’s range. The green bars show the distribution of ‘n’ samples of measurements. The black line represents the gaussian distribution fitted to the green measurements, the gaussians variance and mean are shown in the legend. Figures 4 to 9 show the distributions for the 240Hz system, figures 10 to 15 show the distributions for the 120Hz systems.

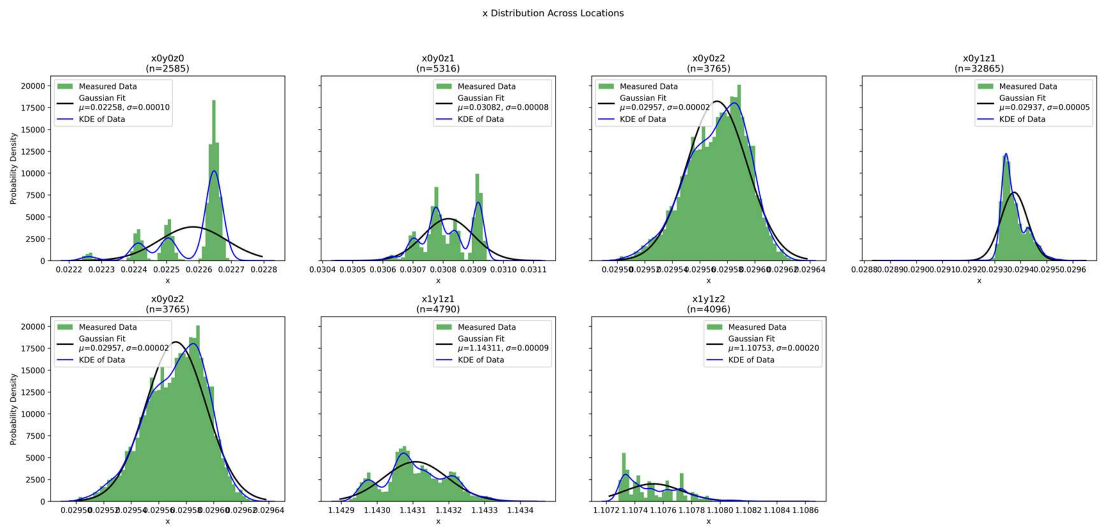

Figure 4. X position distributions across locations for 240 Hz system

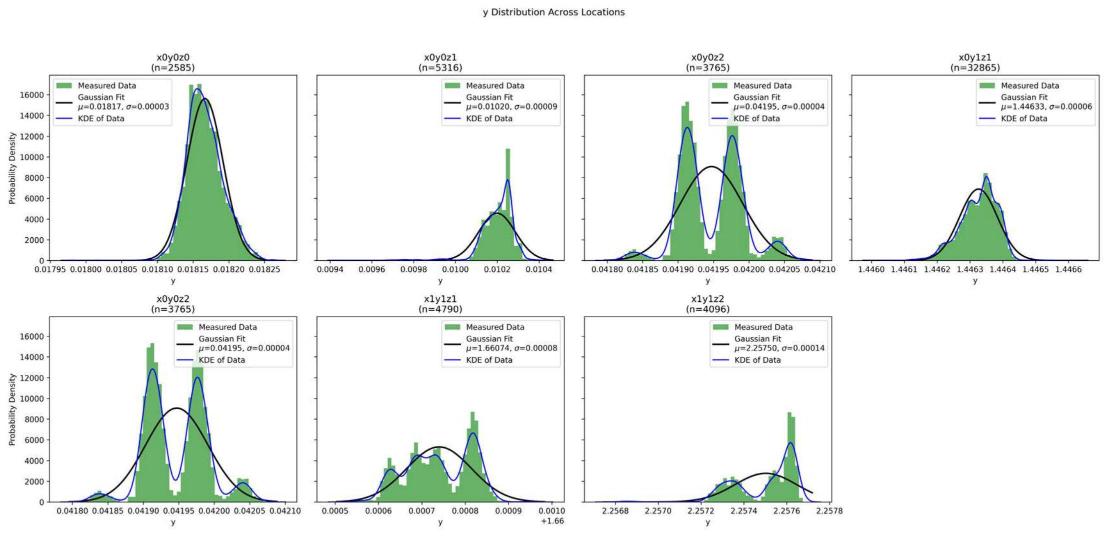

Figure 5. Y position distributions across locations for 240 Hz system

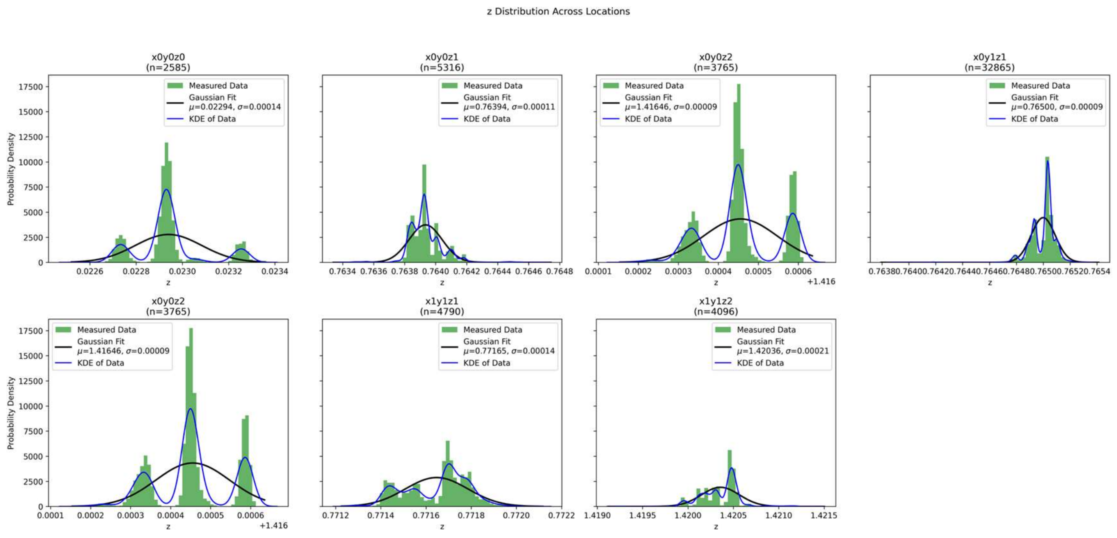

Figure 6. Z distributions across locations for 240 Hz system

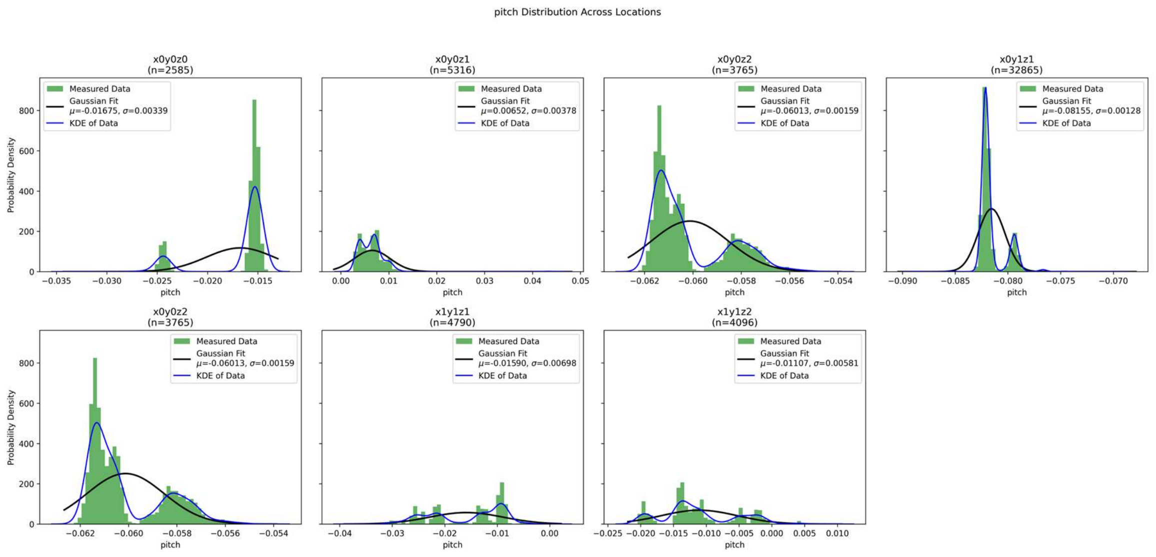

Figure 7. Pitch distributions across locations for 240 Hz system

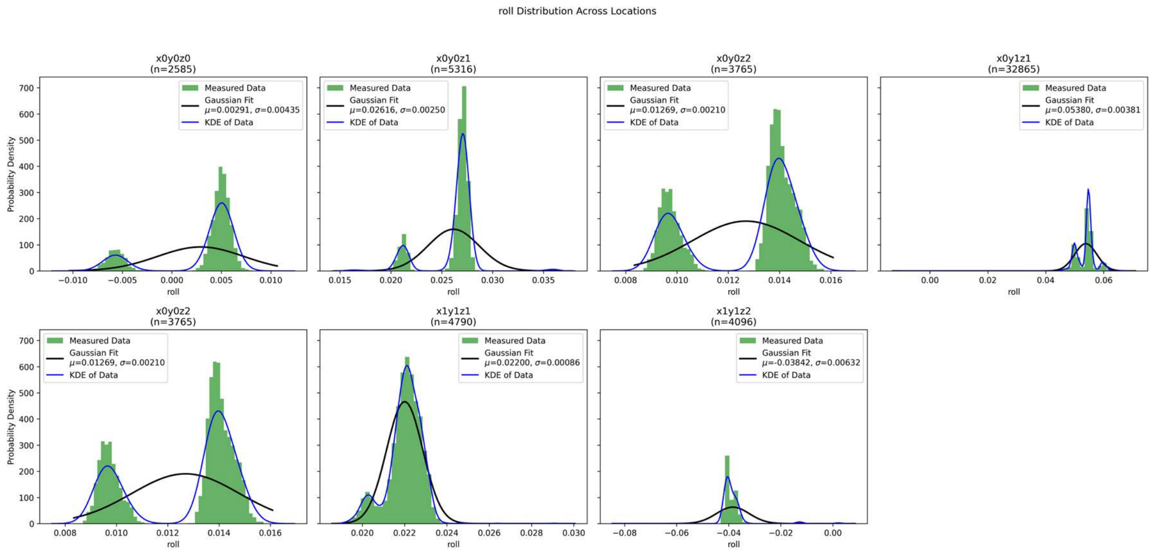

Figure 8. Roll distributions across locations for 240 Hz system

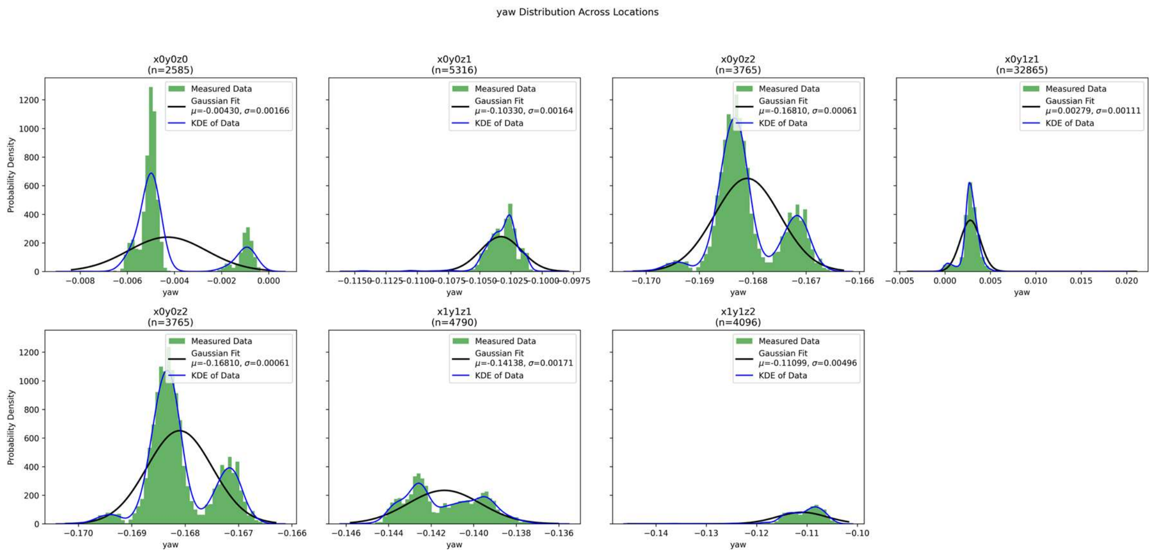
Figure 9. Yaw distributions across locations for 240 Hz system

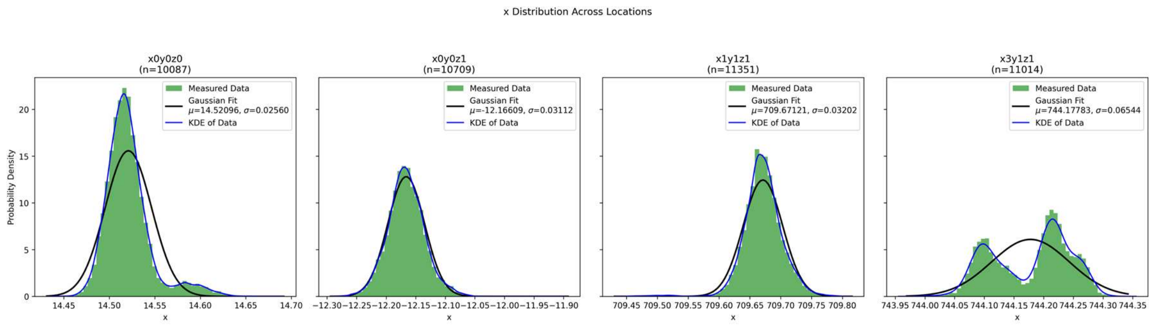
Figure 10. X position distributions across locations for 120 Hz system

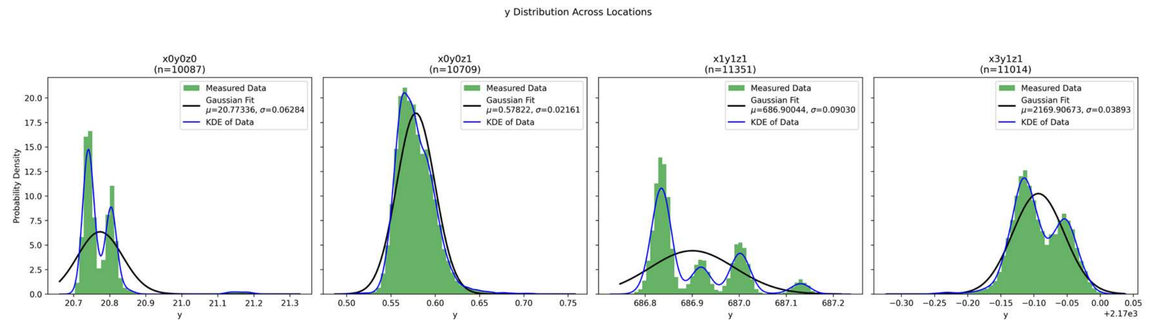
Figure 11. Y position distributions across locations for 120 Hz system

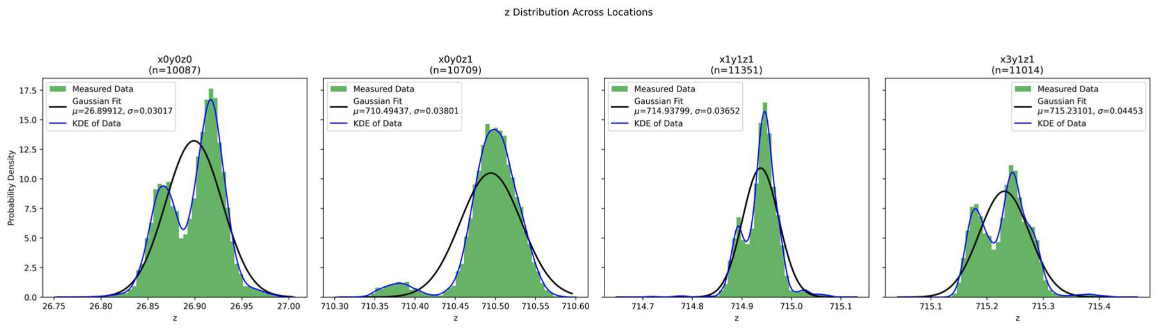

Figure 12. Z position distributions across locations for 120 Hz system

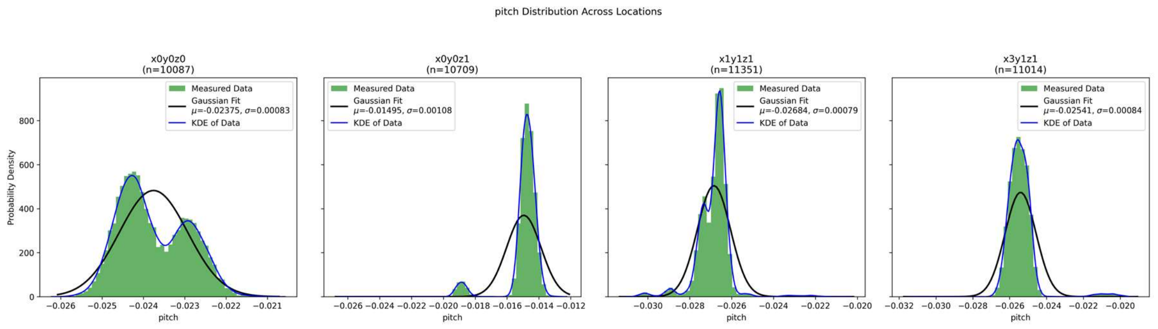

Figure 13. Pitch distributions across locations for 120 Hz system

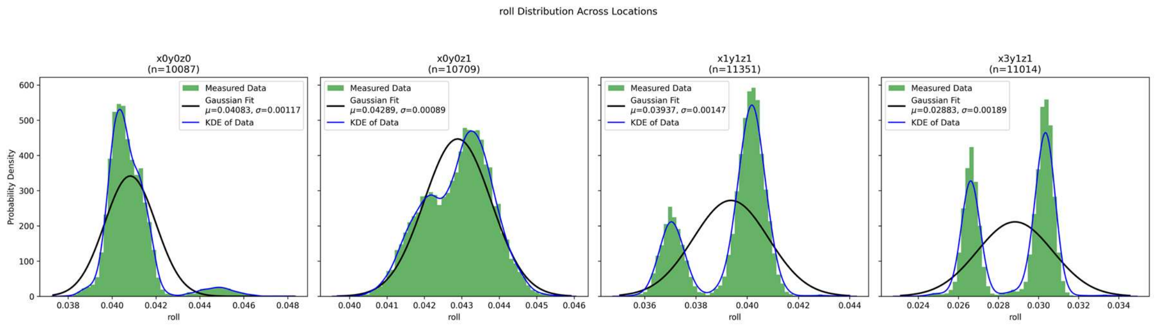

Figure 14. Roll distributions across locations for 120 Hz system

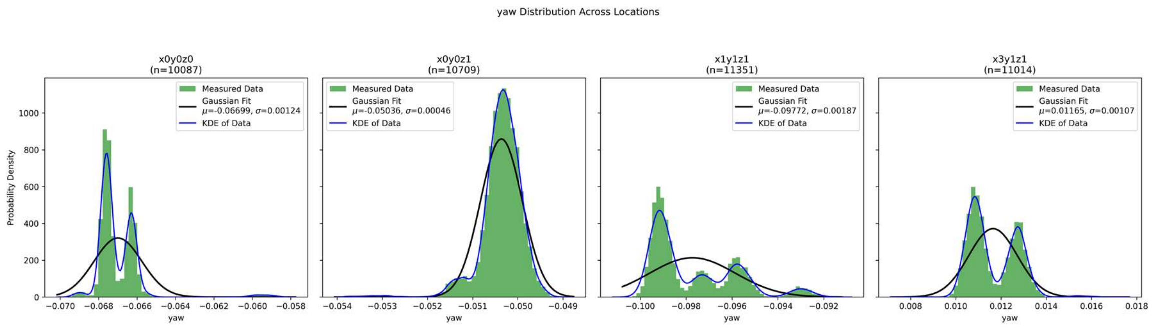

Figure 15. Yaw distributions across locations for 120 Hz system

These results show that the 120Hz system has roughly an order of magnitude less variance than the 240Hz system, and empirically it has been seen that the 240Hz system can suddenly have large errors in its measurements. Sometimes the drone is recorded as being upside down for a single frame, which destroys the control scheme and causes the drone to crash.

Continuing with the 120Hz system, in order to model the cameras in simulation, we will take the worst case variance across all positions for every position and rotation. Later with every observation of the simulated system, we will add the variance to every observation we make. The every observation has a zero-mean gaussian random variable added to it with the variance measured in this section. The simulation section talks more about how precisely the variance is implemented.

	Worst-case variance
X position (m^2)	0.06544
Y position (m^2)	0.09
Z position (m^2)	0.04453
Roll ($Radians^2$)	0.00189
Pitch ($Radians^2$)	0.00108
Yaw ($Radians^2$)	0.00124

Table 1. Worst case variance for all positions and rotations which will be used in the simulation

3.3. Motion capture network modeling

The Motion capture system is computationally demanding. If the controller algorithm was running on the same machine, it would encounter significant slowdown which is undesirable. Instead, the controller algorithm runs on a remote machine, and using the VICON API we can stream data to a remote machine. To measure the network delay a version of the control loop was implemented. The control loop was: get the video observations, process the observations to actions, send actions as setpoint, and repeat. The control loop is visualized in figure 2. The frequency of the control loop was measured and recorded. Figures 16 and 17 show two separate recordings with a gaussian random variable fit onto the data.

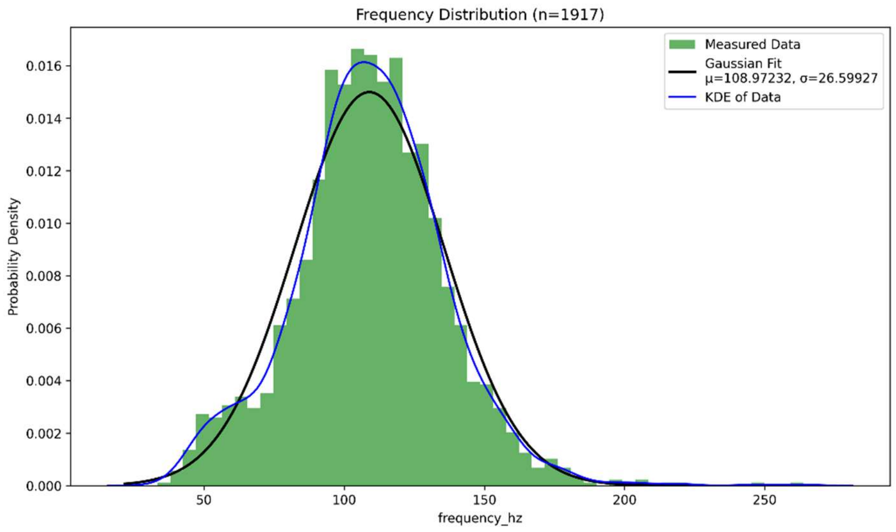

Figure 16. First set of control loop frequency measurements

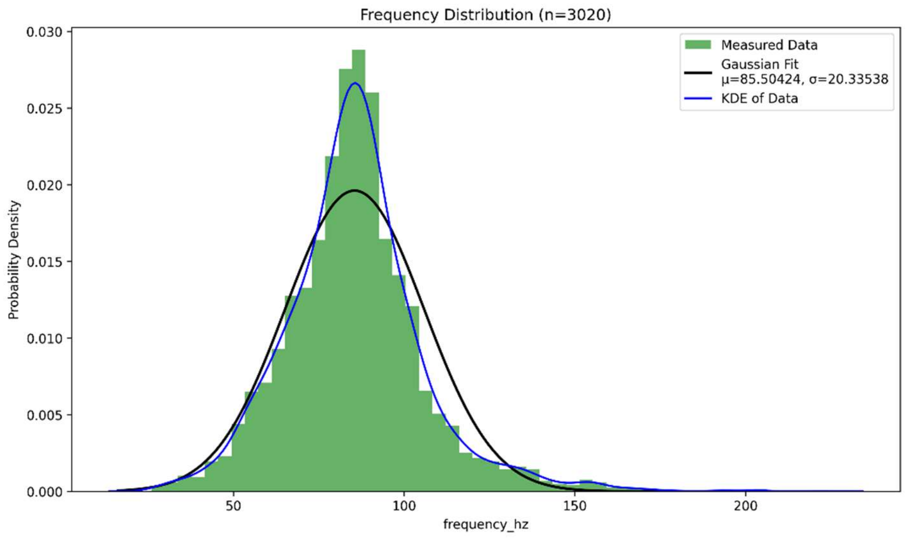

Figure 17. Second set of control loop frequency measurements

The stream mode was set to 'ClientPull' which means that the remote machine asks for the server to return the frame data. Each client pull is blocking, which adds some transmission delay to the control loop. The motion capture system was set to have a frequency of 120 Hz, with the transmission delay it is seen that the average control frequency can drop significantly in some cases, though in practice the average does hang around 120Hz. Importantly the variance is greater than 20Hz, in some cases the variance has been measured to be 28Hz.

4. Simulation

4.1. Overview

In order to train the controllers, we need to use an simulation which is compatible with the OpenAI gymnasium interface. Some work has already been done to simulate the Crazyflie 2.X drone for use in RL with the Gym interface by Jacopo Panerati[7]. This previous work was good at simulating the drone and provided a good starting point for RL. That being said, the system was extremely ideal, the observations and actions were perfect, the episodes were extremely short, the drone would not truncate when it flipped over, it could not learn to fly to various locations, and it was unable to learn if all the motors speeds were equal. A lot of work had to be done in order to use the simulation to help cover the Sim to Real gap.

The first thing that was done was updating the system to more closely resemble the real system. The most efficient way to do this is to model a zero mean gaussian noise onto the observations and to model the motor inefficiencies. The camera system can only observe the pose of the quadcopter and not the velocity or acceleration, therefore the zero-mean gaussian noise only has to be applied to the positions and the noise will propagate to any higher order observations such as velocity. This noise does not affect the kinematics of the quadcopter, so it should only be applied in the Open AI Gym *_computeObs()* method. The variance used is from the measurements in section 2, and they are applied as:

$$obs = obs_{real} + \mathrm{N}(\mu = 0, \sigma).$$

The motor thrust variance was extremely hard to model empirically as the rpm was impossible to precisely measure with laser tachometers and varied greatly as the battery voltage changed. In practice the quadrotor should be fully charged, and it can be expected to fail when the battery voltage gets too low. In testing, empirically the motors had a maximum thrust of about 9.6g and between the motors it varied about 0.5g. This would be a variance of about 5%.

In order to simulate the nonideal motors every time the environment reset the coefficient of thrust ‘KF’ had the zero mean gaussian noise added on. The ideal simulated KF was approximately 3e-10 so the variance was set to 1e-11, and it was applied as:

$$KF = KF_{ideal} + \mathrm{N}(\mu = 0, \sigma).$$

Varying the KF is a good stand in for the actual motor variance seeing as the equation for ideal propeller thrust is:

$$Thrust = KF * rpm^2$$

This makes SimToReal work easier, because the thrust/action noise is linear because KF is varied instead of rpm.

Importantly, the variable control frequency cannot be modeled using the current simulation. The current simulation requires that the simulation frequency is a multiple of the control frequency (control loop frequency). If the simulation frequency is varied then the simulation frequency should be set to a multiple of all the possible frequencies. Even if the control frequencies are limited to a small set the simulation frequency would greatly increase if variable control frequency is simulated. The machines used for training in this thesis are unable to simulate with such a high fidelity and therefore the variable control frequency is not simulated.

To setup the system for better RL learning the observation, reset, and truncation had to be updated. Previously the simulation was only able to learn how to start at the origin (x=0,y=0,z=0) and hover at (x=0,y=0,z=1) with all motor thrusts locked to being equal. The simulation had to be updated to hover at any given *setpoint*. For all observations the translation of the quadrotor was changed to errors, in this case error was defined as:

$$Error = setpoint - position.$$

This method was chosen over just adding the goal positions to the observation to reduce the dimensionality of the observations and to speed up the learning. Whenever the environment was reset a new random goal position / setpoint was chosen in a spherical point cloud of variance 1 meter, and the height was bounded to be z >= 0 to ensure the simulated quadcopter did not start underground (the floor is a physical plane with collision which cannot be crossed). Randomizing the setpoint every time the environment resets ensures that the controller learns to specifically minimize the error and not to fly to a trained setpoint. Truncation was updated to include bounds on the quadcopter pose. The translation of the quadcopter was bounded in order to limit the reward of a quadcopter that flies out of bounds. The simulation is at the end of the day a variation of the Euler method and is therefore susceptible to errors if the system is too dynamic, so we limit the rotations of the quadrotors to discourage the controller from rotating the quadrotor too far.

The reward functions are one of the most important considerations for any RL objective. Previously the reward function was:

$$Reward = e^{-2*(z-1)^2}$$

Which only rewarded hovering at z = 1, with a maximum reward of 1. To encourage better learning, rewards were added to minimize the translation errors, minimize the yaw, minimize the velocity, minimize the rotational velocity, and to maximize the progress towards the setpoint. The exact rewards which yielded the best learning were as follows:

$$R_{pos} = e^{-N \qquad {}_2(Target\ Position-Position)^2}$$

$$R_{yaw} = e^{-0.2*Norm_2(Yaw)^2}$$

$$R_{Vel} = e^{-2*Norm_2(Velocities)^2}$$

$$R_{Rotational\ Vel} = e^{-0.1*Norm_2(Rotational\ Velocities)^2}$$

$$\varphi_{Current} = -Norm_2(Target\ Position - Position)$$

$$\varphi_{Previous} = -Norm_2(Target\ Position - Last\ Position)$$

$$R_{Progress} = 0.99 * \varphi_{Current} - \varphi_{Previous}$$

$$R_{Total} = 0.8 * R_{pos} + 0.2 * R_{Yaw} + 0.15 * R_{Vel} + 0 * R_{Rotational\,Vel} + 1 * R_{Progress}$$

The parameters for the rewards were found through empirical methods and not through theory, that being said they performed excellently.

With this reward setup one local minimum is when the drone successfully lands but does not fly to its target position, and it results in a reward of approximately 1300. A quadcopter which successfully flies to the target archives a reward slightly greater than 3000. For this reward structure to produce the best learning, simulation starts the quadcopter at (x=0, y=0, z=2) and randomizes the target position. This is done so that the quadcopter can collect 'bad' rewards as it falls away from target position.

4.2. Observation Spaces

There were multiple feasible observation spaces for this system to learn. The simplest was to observe the errors, rotations, velocities and angular velocity. These observations include the camera noise for the pose, and the velocities are calculated from previously measured poses. These simple observations worked when the simulation was ideal but tends to fall apart when the motor variance gets too high.

In order to compensate for motor variance another observation space was designed. This one is similar to the previous one with errors, rotations, velocities, and angular velocities, but it also includes the average actions over the last 5 actions, and the difference between the last action and the average actions for all 4 motors. This was able to better compensate for the motor noise and is still small enough to train.

4.3. Action Spaces

All the actions result in motor RPMs, but the action space of the controllers are different and may be preprocessed in order to output motor RPMs. The first action space that was used was a value from -1 to 1 this value is then used to change the rpm around the hover rpm. Where the hover rpm is the rpm required for a motor to produce thrust ¼ of the quadrotor mass. The actual equation was:

$$rpm = rpm_{hover}(1 + \beta * Action)$$

Where β is the percentage that the controller can vary the rpm around the hover rpm. This action space allows the controller to learn a much smaller range of rpms, and it provides a stable point close to an action of 0. This action space comes with an inherent issue; there needs to be a good estimate of the hover rpm for each motor. This action space will be called real1 in further discussion.

Another, more obvious action space was to have the controller output a value from 0 to 1 where 0 was a motor rpm of 0 and 1 was the maximum rpm. This action space has a much larger range of rpms for the controller, and it has to learn a suitable rpm for hovering. Multi-Layer Perceptrons were unable to learn how to control the quadcopter using this action space. The MLPs were likely unable to learn how to control the quadcopter because the varying KFs would mean it would not be able to learn a suitable hover rpm. MLPs with a Long Short Term Memory component were able to learn to control this action space, because they are able to use their previous experience to learn to compensate for KF variance. This action space will be called real2 in further discussion.

5. Training

5.1. Training Setup

Firstly, Multi-Layer Perceptrons were trained to control the system using the real1 action space (Discussed in section 4.3). MLPs were chosen because they have been shown to be capable of controlling a large variety of systems such as inverted pendulum, cart pole, or even Atari games. The MLPs that were trained were ‘MlpPolicies’ trained from stable baselines, and as such the MLP was 2 layers of 64 neurons with a ReLU activation function. Proximal Policy Optimization (PPO) was chosen as the learning algorithm for the controllers. PPO was chosen because it provides smooth learning due to clipping, it is easy to implement, and is great for learning without too much tuning when compared to DDPG.

The controllers were trained to hover the drone at a setpoint by minimizing the rewards discussed in section 4. The initial positions and rotations are randomized at the start of each simulation allowing for the learning of control strategies instead of finding the best string of actions to take. The observation structure includes the position errors discussed in the simulation section, so we are able to fly the quadrotor along a trajectory. Each simulated trajectory is simulated with a KF motor variance of 10^{-11}.

While the rewards for all controllers were the same and they were all trained using PPO, the learning setting were different for the MLPs vs the MLP+LSTMs. The base MLPs had one normalized vectored environment. The batch size was 512, 3 epochs, 4096 steps, entropy coefficient of 0.01, a clipping range of 0.2, and a learning rate of 5e-5+(5e-5*sqrt(frac)) where frac is the percentage of training left. The MLPs with LSTMs had 10 normalized vectored environments. The batch size was 512, 5 epochs, 4096 steps, entropy coefficient of 0.02, a clipping range of 0.2, and a learning rate of 5e-4+(5e-4*sqrt(frac)). The learning for both types of controllers was done on an intel core i7-9750H CPU running at 2.6 GHz, and had to learn for approximately 10 hours for 5e6 timesteps (simulation steps).

For testing the robustness of the trained control algorithms, one metric to use could be success rate. Success rate can be defined as the rate at which the quadcopter does not crash or fly out of bounds in our simulated environment. The variability if the coefficient of trust KF directly maps to variance of the motors, the more variant the motors the less ideal the system. By varying the motors and measuring the success rate, we can get good estimates of robustness of our system.

5.2. Multi-Layer Perceptrons without memory

The first multi-layer perceptron (Named *SimToReal_61*) was trained without motor variance and with the simple pose and velocity observations of observation real 1 and was able to learn how to control the simulated quadcopter but quickly began to fail when the motor variance began to rise too much. This makes sense as the MLP optimized for the ideal case and would lack robustness when the motors began to vary, though it would be highly efficient at controlling the ideal case. The next MLP (Named *SimToReal_60*) was then trained with the same observation and action space, but this time included motor variance. This MLP provided slightly more robustness than the ones trained without motor variance, but not enough to be beneficial.

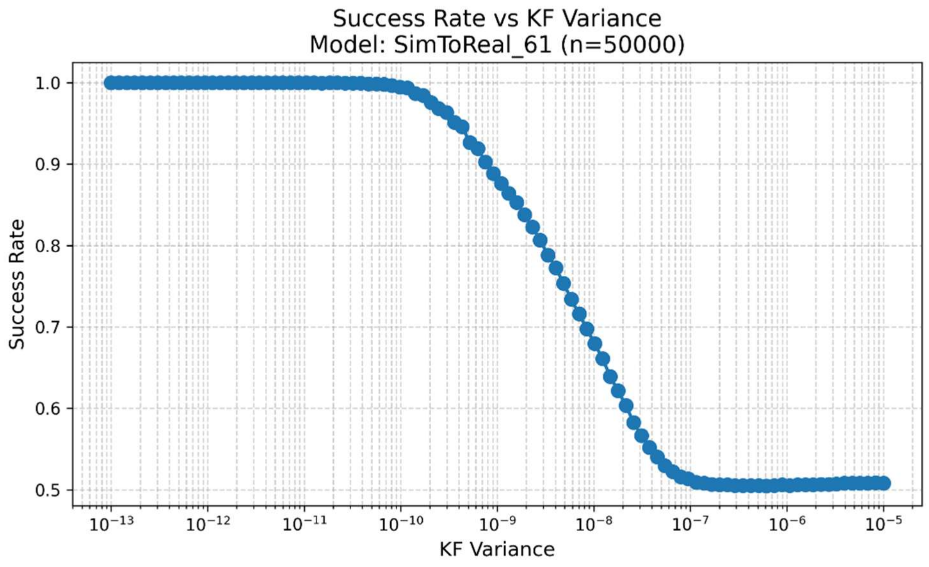

Figure 18. Success rate vs Thrust Variance for model SimToReal_61

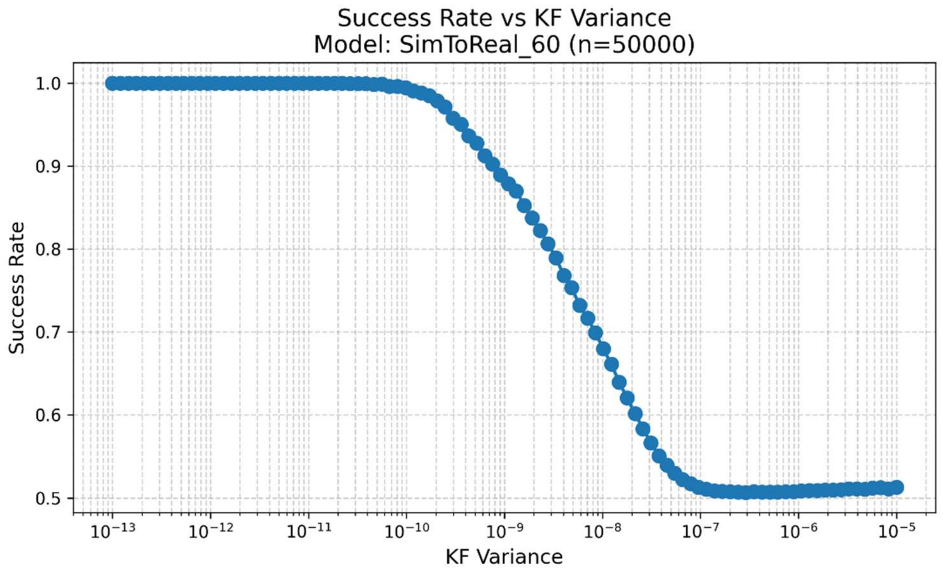

Figure 19. Success rate vs Thrust Variance for model SimToReal_60

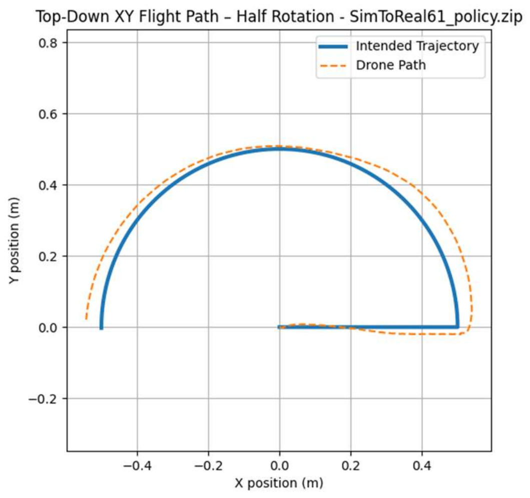

Figure 20. A sample trajectory of the quadrotor controlled by SimToReal61 starting at (0,0) and ending at (-0.5,0)

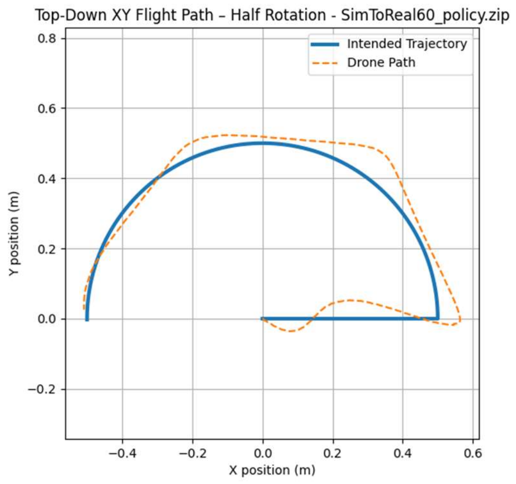

Figure 21. A sample trajectory of the quadrotor controlled by SimToReal60 starting at (0,0) and ending at (-0.5,0)

Figure 18 shows the success rate of the MLP trained without motor variance and figure 19 shows the success rate of the MLP with motor variance, as you can see they are nearly identical. Figures 20 and 21 show both controllers following the same semicircular trajectory, and graph their X and Y positions. The controllers were both able to follow the trajectory, which is to be expected. The *SimToReal_61* controller provided a much smoother control than *SimToReal_60*, likely due to *SimToReal_60* being trained with noisy actions.

5.3. Multi-Layer Perceptron with Memory

The previous MLPs may not be as robust as possible because they had no memory of their previous actions and their effects on the system. To rectify this another MLP was trained with the observations which included the average previous actions and the delta of the last action along with the pose and the velocities. Ideally this model would be able to learn to correlate the last actions and velocities to compensate for the variable motors. *SimToReal_63* is a model trained with such an observation.

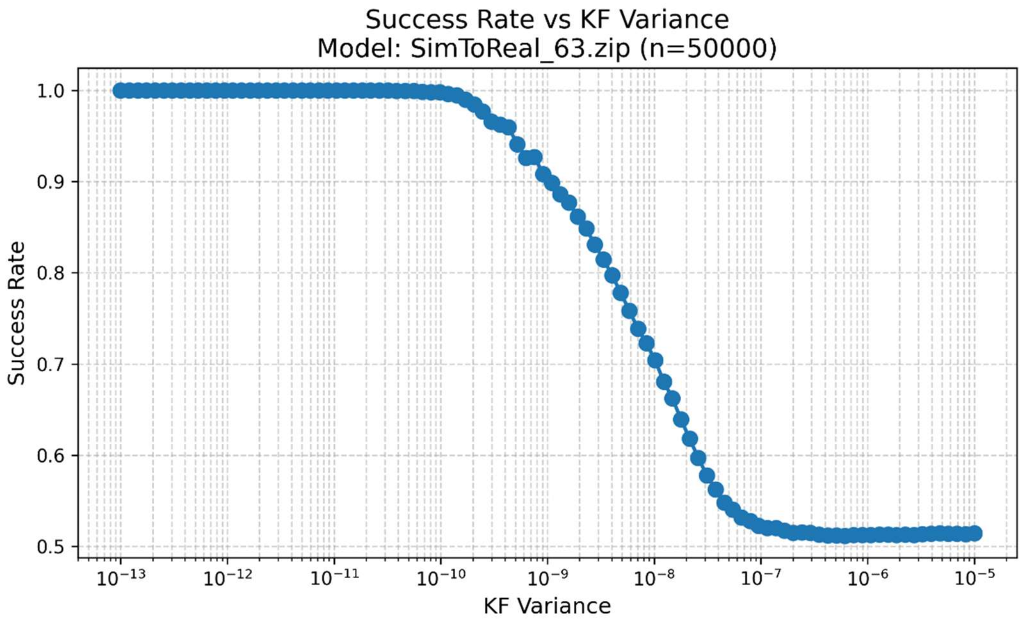

Figure 22. Success rate vs Thrust Variance for model SimToReal_63

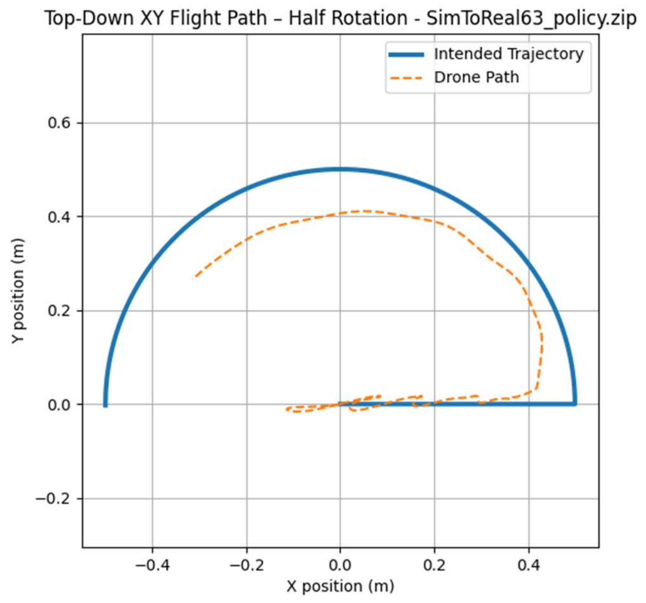

Figure 23. A sample trajectory of the quadrotor controlled by SimToReal_63 starting at (0,0) and ending at (-0.5,0)

Figure 22 shows the results of the new model *SimToReal_63* in simulation. This model provides a visible improvement on robustness when compared to the previous two models. This model flies extremely smoothly in simulation and can handle much more variable motors. Figure 23 shows the trajectory of the quadrotor controlled by the *SimToReal_63* controller in a semicircle. *SimToReal_63* was clearly able to control and stay stable along the trajectory, but it was unable to fly as fast as the previous controllers.

The improvement in success rate suggests that the controller is utilizing the improved observation space (which includes the previous actions). The number of inputs being utilized by the MLP can be measured by checking the rank of the weighting matrices, and the number of non-zero biasing layers. In a Multi-Layer Perceptron, the biasing layers are the individual 'neurons' of the network and the weighting matrices are the connections between layers of neurons, this is shown in figure 24 except the X H and O neurons are all 1 dimensional. The rank for the weighing matrices of SimToReal_63 was 20 (full rank/the size of an observation) and the biasing layers were non-zero, which indicates that SimToReal_63 is truly using memory.

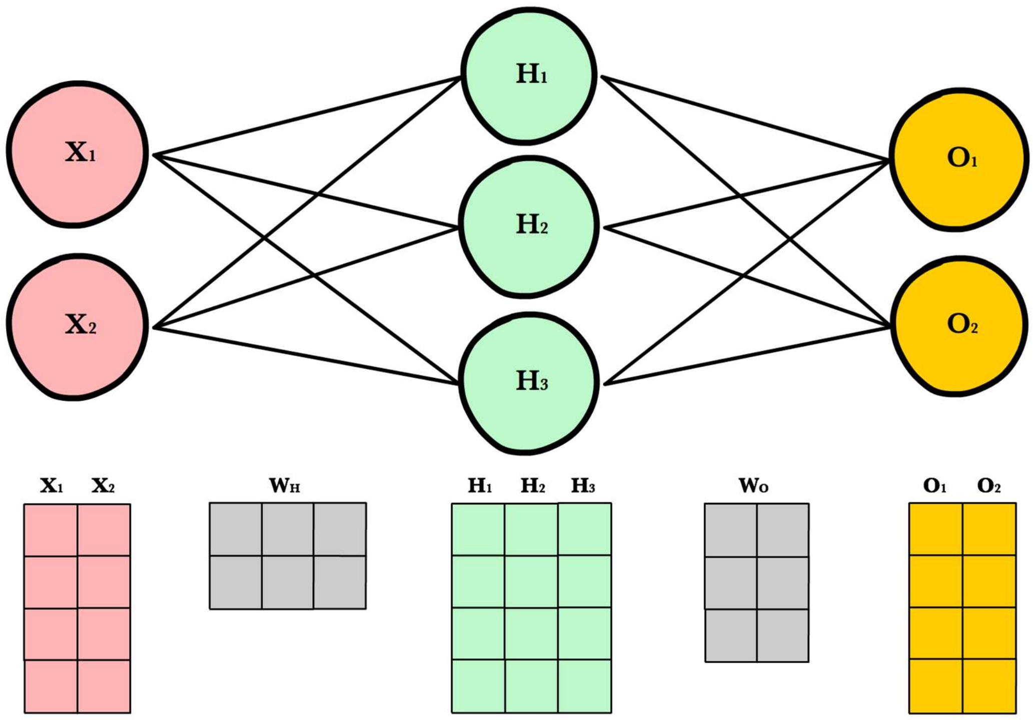

Figure 24. Illustration of a Neural Network and its matrix representation (From SB3 Website)

5.4. Multi-Layer Perceptron with Long Short Term Memory (MLP+LSTM)

To make this algorithm more robust, a model with more memory can be used. The goal is for the memory of past observations and actions to allow the controller to correlate previous actions and states learn how to compensate for motor variance. Stablebaselines 3 allows us to train another model using LSTMs with MLP feature extraction. The LSTMs allow the model to look backwards in time for many time steps. Figure 25 shows the success rate of the resulting MLP + LSTM model (*PPO_LSTM_10*), we can see that compared to the *SimToReal_63* model, the success rates are almost identical.

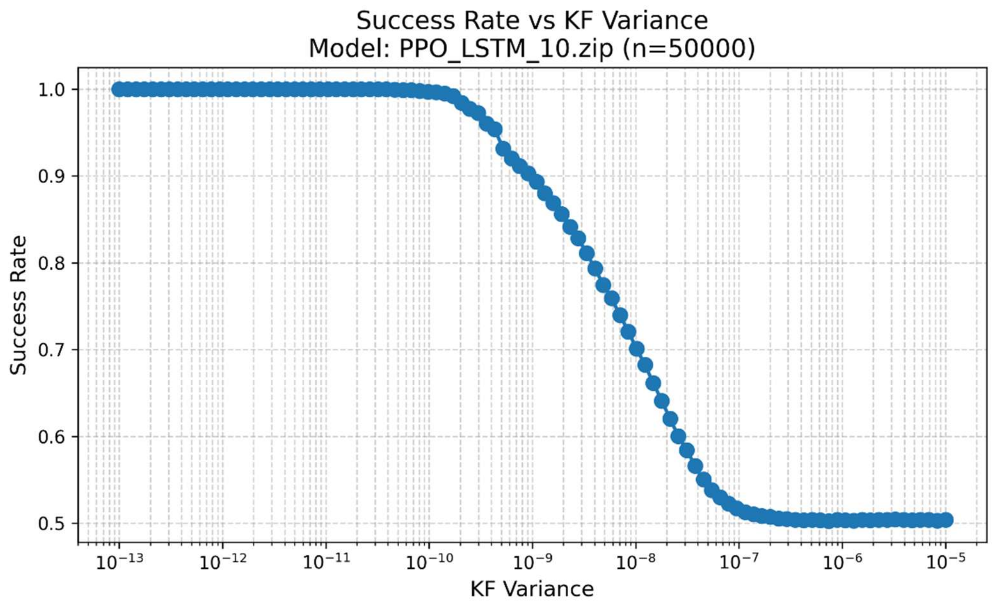

Figure 25. Success rate vs Thrust Variance for model PPO_LSTM_10

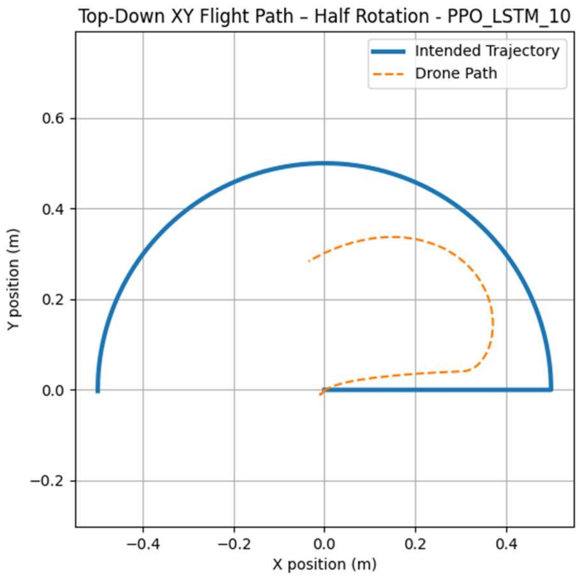

Figure 26. A sample trajectory of the quadrotor controlled by PPO_LSTM_10 starting at (0,0) and ending at (-0.5,0)

Figure 26 shows the trajectory of the quadrotor controlled by the *PPO_LSTM_10* controller in a semicircle. Like the previous controllers this controller was able to maintain stability throughout the entire trajectory. Unlike the previous controllers this controller was the slowest yet, only completing about 90 degrees of the 180 degree semicircle trajectory.

5.5. MLP + LSTM with more direct actions

The previous models were trained to control the system using the real1 action space. The real1 action space requires an estimate of the hover rpm for each motor. In practice the hover rpm was calculated by hovering the quadcopter using a simple PID controller and measuring the average commanded thrust. This estimate is reasonably accurate but changes when the battery voltage varies or if the weight distribution of the quadrotor changes. To test to quality of the hover rpm estimate, the hover rpm value in the flight program was varied. The result of these tests empirically showed that the estimate greatly affected the quality of the flight.

The previous MLP models were unable to control the quadcopter using the real2 action space. This is assumed to be because the MLP controllers were not dynamic controllers and could not dynamically learn the true hover rpm. A dynamic controller like an MLP with LSTMs could dynamically learn the true hover rpm. *PPO_LSTM_11* was trained using the real2 action and was the first model to learn how to control the simulated quadrotor using the real2 action. *PPO_LSTM_11* flew the simulated quadrotor excellently, and was able to control the system with a great deal more robustness than the previous models. Figure 27 shows the success rate vs motor variance for *PPO_LSTM_11*.

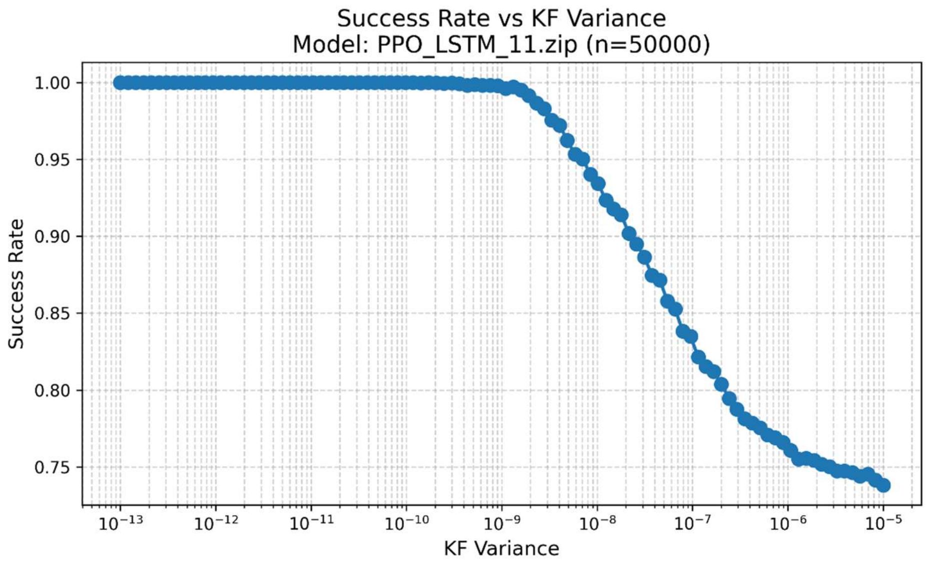

Figure 27. Success rate vs Thrust Variance for model PPO_LSTM_11

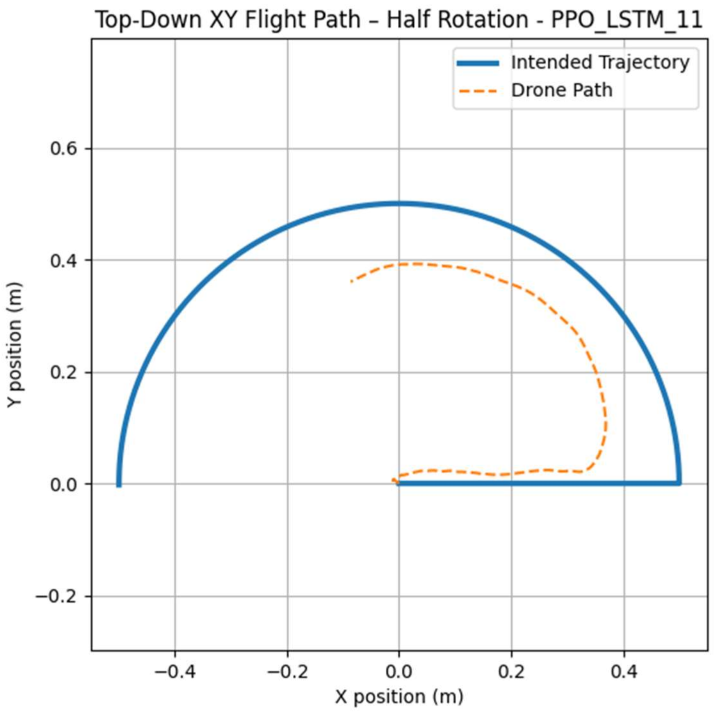

Figure 28. A sample trajectory of the quadrotor controlled by PPO_LSTM_11 starting at (0,0) and ending at (-0.5,0)

Figure 28 shows the trajectory of the quadrotor controlled by the *PPO_LSTM_11* controller in a semicircle. Like all previous controllers, this controller was able to maintain stability throughout the entire trajectory. *PPO_LSTM_11* controls the quadrotor slower than the previous non LSTM controllers, though it controls the quadrotor at approximately the same speed as *PPO_LSTM_11* while also being significantly more robust.

5.6. Simulated Controller Comparison

Figure 29 shows the success rate vs motor variance for all of the trained models, here the difference in robustness is really made clear. *SimToReal_61* and *SimToReal_60* both lack memory and the only difference between them is training with and without thrust noise, their different training methods result in effectively no difference in robustness. *SimToReal_63* iterates on the previous two models by including some memory into the observations (inputs) of the MLP, resulting in a controller which is measurably more robust than the previous models. *PPO_LSTM_10* includes memory into the actual network by using LSTM, resulting in a controller with about the same robustness as *SimToReal_63*. Finally, *PPO_LSTM_11* uses the previously uncontrollable action space of direct motor power control, resulting in a greatly more robust controller than the previous four.

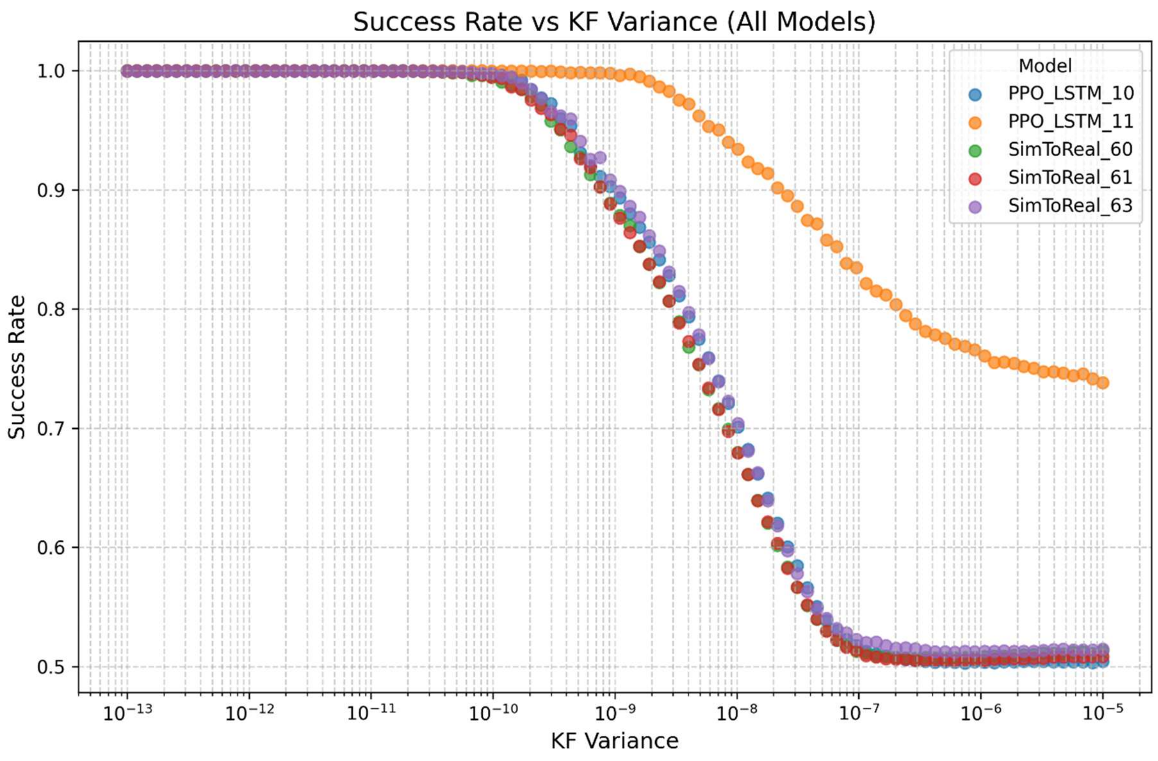

Figure 29. Success rate vs Variance for all models

The trajectories of the different controllers also paint an interesting picture, as all controllers were designed using the same reward structure they should all optimize to similar performances regarding velocity and positional error. The controllers with more memory moved slower than the models without, and the controllers with LSTM were the slowest.

Model	Ideal Simulation	Noisy Simulation	Real Environment	Action Type	Memory Level
SimToReal61	✓	✓	X	Hover RPM	no Memory
SimToReal60	✓	✓	X	Hover RPM	no Memory
SimToReal63	✓	✓✓	X	Hover RPM	some memory
PPO_LSTM_10	✓	✓✓	X	Hover RPM	memory
PPO_LSTM_11	✓	✓✓✓	X	0 - rpm Max	memory

Table 2. Performance of different models

6. Future work

Each controller was tested on the real quadrotor and motion capture system. None of the controllers were able to control the real system, although *PPO_LSTM_11* was the closest with what appeared to be underdamped oscillations. This follows the trend of SimToReal being extremely hard, and points to the need for future work in order to try closing this gap.

Future work in closing the SimToReal gap should include more accurate modeling of the system for simulation. This simulation, like all simulations, makes multiple assumptions which differ to the real world. One assumption is that the propeller rpm always matches the commanded rpm. This assumption was made to simplify the simulation and because the motors react so much faster than the controller. In reality the motors have inertia and do not spin up immediately, along with that there is motor slippage between the motor drive shaft and the propeller. Another assumption was that the radio communication was accurate and instant, while the setpoint architecture is extremely fast, it incorrect to say it is instant and the communication is always accurate with no packet drops. A future researcher should model these two aspects of the system in order to create a more accurate simulation.

Another method to close the SimToReal gap is to change the controller architecture. Typically, quadrotors are controlled by a cascaded controller, where the lower level controller controls the attitude of the quadrotor at 500Hz and the higher level controller controls the position at a slower rate. In this thesis the decision was made to create a controller to directly control the position at 120Hz. This decision was made in an effort to take advantage of the simulation and cover a gap in literature. While the simulation does show that the system is controllable, the real system may not be controllable because of the variable delay margin, and increased susceptibility to noise.

7. Conclusions

In this thesis we developed new open source firmware to control the Crazyflie 2.X quadcopter's motors individually and remotely using setpoint commands, characterized the Vicon Vero motion capture system using real data and modeling random variables, created python simulations of the Crazyflie quadcopter using OpenAI Gym interface, trained multiple different neural network controllers with different action and observation spaces to fly the quadcopter, measured the robustness of the controllers in simulation, and implemented the controller on the real system.

The framework was developed to create a direct control for the Crazyflie 2.X quadrotor using zero-shot learning, which would be run from a remote machine and not onboard the quadrotor for easier programming and implementation. This control problem comes with many built-in challenges. Zero-shot learning for control requires an incredibly accurate model of the quadrotor, which motivated the characterization of the motion capture system and motor variance. The Crazyflie 2.x does not have a method for direct control over the radio, which was solved by modifying the setpoint commands to send motor thrusts. The motion-capture system works at 120Hz and limits the direct controller to 120Hz but the default attitude controller for the Crazyflie 2.x runs at 500Hz which may cause controllability issues, so the simulation was set to a control frequency of 120Hz which shows that the system is controllable at 120Hz and validated other design decisions.

To produce a robust control for bridging the sim to real gap, many different controllers were designed and tested. The most robust controller architecture was using LSTMs with MLP feature extraction, and without hover velocity estimates. These controllers had excellent performance in simulation, allowing for an order of magnitude more motor noise before having the success rate fall below 100%, and having 25% better success rate at high motor variance when compared to other algorithms. This increase in robustness can be extremely useful for SimToReal applications which require a great deal of robustness. The most robust controllers were also found to be some of the slowest, that being said the most robust controller *PPO_LSTM_11* was the same speed as the next most robust controller *PPO_LSTM_10* which may suggest that the architectures with memory were slower because of the memory and not robustness.

It was shown that controllers with memory provided a great deal more robustness to action variance. The first controllers have memory in their observations, they observe velocity, which is a very low level of memory, this amount of memory allows for the creation of a controller. More memory was added into the observations by observing previous actions, resulting in measurably more robust controllers (SimToReal_63 vs previous models). Models with much more memory in the form of LSTM modules were then trained. Using this level of memory the controllers were much more robust to noise and was capable of controlling the motors without an estimated hover speed. I highly recommend designing controllers with memory for control of quadrotors.

Future research can utilize the developed Crazyflie firmware for direct control, simulation, training methods and information, and fly the real quadrotor using the code and information uploaded to the GitHub repository found at https://github.com/Nicholas-Navarrete/ML-controllers-with-memory-for-robust-quadrotor-control-and-research.

References

[1] S. L. Waslander, G. M. Hoffmann, Jung Soon Jang and C. J. Tomlin, "Multi-agent quadrotor testbed control design: integral sliding mode vs. reinforcement learning," 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton, AB, Canada, 2005, pp. 3712-3717, doi: 10.1109/IROS.2005.1545025. keywords: {Testing;Control design;Learning;Sliding mode control;Aerodynamics;Aircraft;Blades;Attitude control;Vehicle dynamics;Aerospace control},

[2] Kownacki, C., Romaniuk, S. & Derlatka, M. Applying neural networks as direct controllers in position and trajectory tracking algorithms for holonomic UAVs. *Sci Rep* **15**, 12605 (2025). https://doi.org/10.1038/s41598-025-97215-9

[3] Jack F. Shepherd and Kagan Tumer. 2010. Robust neuro-control for a micro quadrotor. In Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO '10). Association for Computing Machinery, New York, NY, USA, 1131–1138. https://doi.org/10.1145/1830483.1830693

[4] J. Dunfied, M. Tarbouchi and G. Labonte, "Neural network based control of a four rotor helicopter," 2004 IEEE International Conference on Industrial Technology, 2004. IEEE ICIT '04., Hammamet, Tunisia, 2004, pp. 1543-1548 Vol. 3, doi: 10.1109/ICIT.2004.1490796. keywords: {Neural networks;Helicopters;Intelligent robots;Intelligent networks;Attitude control;Aircraft navigation;Propellers;Drones;Mechanical variables control;Stability},

[5] Hwangbo, J., et al. "Control of a Quadrotor With Reinforcement Learning," in *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[6] S. Edhah, S. Mohamed, A. Rehan, M. AlDhaheri, A. AlKhaja and Y. Zweiri, "Deep Learning Based Neural Network Controller for Quad Copter: Application to Hovering Mode," 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), Ras Al Khaimah, United Arab Emirates, 2019, pp. 1-5, doi: 10.1109/ICECTA48151.2019.8959776. keywords: {Deep learning;Training;Computers;Supervised learning;Computer architecture;Autonomous aerial

vehicles;Feedforward systems;Long short term memory;Standards;Quadrotors;Deep Learning (DL);RUAV Control;Quadrotor Control;LQR;Deep Neural Network (DNN)},

[7] Jacopo Panerati, undefined., et al, "Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” arXiv.org, Aug. 28, 2017. https://arxiv.org/abs/1707.06347