Quadcopter Control with Deep Reinforcement Learning and PID Controllers in Simulated Environments

James Antonio Scoon González

A thesis submitted for the degree of Master of Science in Intelligent Systems and Robotics

Supervisor: Dr. Michael Fairbank
School of Computer Science and Electronic Engineering
University of Essex

August 2020

ABSTRACT

Quadcopters are interesting autonomous vehicles which are highly unstable systems. A common method of controlling quadcopters is with PID controllers, however researchers are actively investigating the feasibility of using deep reinforcement learning as an alternative method. This dissertation seeks to compare the differences between using the state-of-the-art deep reinforcement learning algorithm Proximal Policy Optimization to control a quadcopter against using PID controllers with an anti-windup mechanism. A virtual environment which was created from scratch by the author in Unity is used to simulate quadcopters while also serving as a framework to facilitate future work related to quadcopters, PID controllers and deep reinforcement learning.

This dissertation additionally presents and implements a novel method of tuning PID controllers through a genetic algorithm that finds the optimal gains for the controllers based on a reward function. This method allows the gains of the PID controllers to be learnt while optimising the same reward functions as deep reinforcement learning algorithms, minimising the possibility of discrepancies that can invalidate comparisons between PID controllers and deep reinforcement learning algorithms.

Three tasks are used to compare different behaviours for the quadcopters: one for hovering, one for following a path and one for landing on a target.

ACKNOWLEDGMENTS

I am deeply grateful to my supervisor Dr. Michael Fairbank for guiding me through the process of completing this dissertation, and for providing me with support at every step of the way. This support helped me overcome the hardest and most tedious challenges of this dissertation.

I would also like to acknowledge the use of the High Performance Computing Facility (Ceres) and its associated support services at the University of Essex in the completion of this work. Ceres has been an invaluable tool without which I would not have been able to complete this dissertation.

Finally, I would like to thank my family and friends who motivated me to work harder than ever before and surpass my own expectations.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1. INTRODUCTION

The objective of this dissertation is to measure how well Proximal Policy Optimization agents and PID controllers can control quadcopters in a simulated environment.

Quadcopters are unmanned aerial vehicles that achieve flight by spinning four rotors positioned around the craft. These vehicles are capable of vertical take-off and landing (VTOL), and can manoeuvre in all six degrees of freedom. One of the challenging aspects of controlling a quadcopter is that it is an underactuated system, meaning that certain movements cannot be done, such as moving horizontally without rotating.

There are many advantages with using a simulated quadcopter over a real one; buying a commercial quadcopter or the components used to make one could be expensive and building one can also be time consuming. Piloting a real quadcopter can also be dangerous, especially if it is being piloted by an agent or PID controllers that have not completed the learning process, because the quadcopter could fly at a high speed towards another object, potentially damaging the quadcopter or the object in question. Additionally, training can take a very long time to complete, however in a virtual environment the time within the simulation can be sped up drastically, and multiple processors or computers can be used to do training in parallel, reducing the training time.

Deep Reinforcement Learning is a relatively new area of machine learning which has gained a lot of traction due to the large amount of applications it has and its ease of implementation. An advantage of this area is that the reinforcement learning agent does not have to be told which decisions it must choose in order to achieve optimal behaviour, because it instead learns how to make the right decisions over time. This is especially convenient in environments where one knows what the agent has to achieve but not how to achieve it. This applies to the task of

piloting a quadcopter because it is extremely hard for a human to understand how to manipulate the four rotors of a quadcopter so that a specific task such as following a path can be performed.

There are a multitude of different deep reinforcement learning algorithms that have been developed, however in this dissertation Proximal Policy Optimization is analysed because it is a state-of-the-art algorithm developed by OpenAI [14] that has been used in many other works and papers (some examples have been included in Section 2.2).

PID controllers are commonly used to pilot quadcopters because they are simple and extremely effective at stabilising control systems. This means that they are good references for other methods of controlling the quadcopter, such as deep reinforcement learning. A problem with PID controllers is that there is no ideal method for finding the best values or gains for these controllers, so a novel technique is presented in this dissertation where a genetic algorithm is used to find these values through a training process.

This dissertation also presents virtual simulation environment for quadcopters which was built from scratch by the author with the Unity game engine. The simulation environment interfaces with the ML-Agents package [8] to allow one to train deep reinforcement learning agents within the environment.

In order to properly analyse the differences between the Proximal Policy Optimization algorithm and the PID controllers with the genetic algorithm in a virtual environment, one must first create or find an existing virtual quadcopter which is as close as possible to a real one. Subsequently, one must find a way to process and send the outputs of the PPO agent and PID controllers to this quadcopter. Next, one has to design a method of evaluating the performance of the quadcopter with quantitative result. This could be as simple as creating a task which gives a quadcopter a reward value of zero if it performed poorly and one if it performed well. With these results, one can then perform comparisons between the different methods of controlling the quadcopter and come to a set of conclusions. These steps are followed and described in detail in Chapter 3.

1.1 Outline

This dissertation is divided into five chapters and an appendix. Chapter 1 serves as an introduction to the reader. Chapter 2 contains a background reading section where the most important and relevant topics required to understand the dissertation are briefly summarised. This chapter also contains a related work section which shows how this dissertation is related to similar papers or dissertations, while also displaying the results obtained by these works.

Chapter 3 contains detailed descriptions of the methods and procedures followed by this dissertation in order to achieve the results. Chapter 4 displays the most relevant results obtained for each of the simulations. Finally, Chapter 5 discusses the results obtained in Chapter 4 and presents a set of conclusions based on these results. This chapter also goes over different opportunities for future work and the limitations and challenges that came with the dissertation.

Appendix A contains supplementary materials such as graphs and tables which complement the results in Chapter 4, but would be too large or not relevant enough to include in that chapter.

CHAPTER 2. LITERATURE REVIEW

2.1 Background Reading

2.1.1 Quadcopters

Quadcopters are a popular type of Unmanned Aerial Vehicle (UAV) that have four rotors which allow them to move and rotate in three-dimensional environments that have an atmosphere. The rotors of these vehicles are designed to produce a force of lift which will oppose the force of gravity, allowing them to hover or fly upwards.

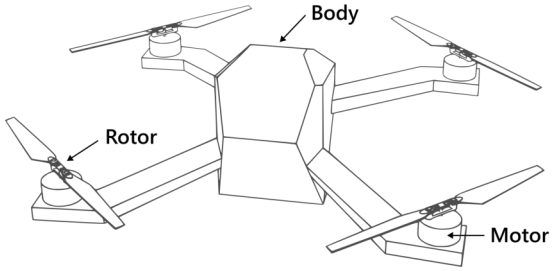


Figure 2.1: Structure of a quadcopter.

2.1.1.1 Dynamics

Each rotor of a quadcopter is physically connected to a motor, which when spun allows the rotor to cut through the air. When the rotor does this, the air resists this action due to Newton's Third Law and applies a force on each of the blades of the rotor. This force can be divided into two components: a lift component and a drag component.

Figure 2.2 displays a diagram of the cross-section of the blade of a rotor, along with the opposing force of the air and its two components. The component of the force that points upwards is generally referred to as the *lift* or *thrust* force, while the component that points against the velocity of the blade is referred to as the *drag* force.

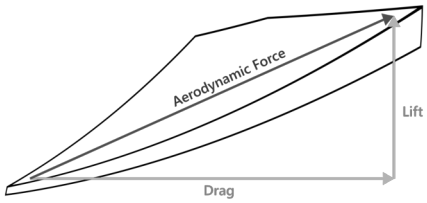


Figure 2.2: When the blade spins the air applies an opposing force.

When the motor is actuated, it generates an electromagnetic torque that allows the rotor to spin. Because the blades are distributed at equal angles around the centre of the rotor, the drag forces generate a torque that opposes this electromagnetic torque. This drag torque must always be less than or equal to the electromagnetic torque otherwise the rotor would spin on its own or in the reverse direction, which would violate the law of conservation of energy. The result is a net torque on the rotor which is equal to the difference between the electromagnetic torque of the motor and the drag torque. The lift forces of each of the blades also add up to create a force that points upwards and acts upon the centre of the rotor. This means that the dynamics on each of the rotors can be represented by a lift force pointing upwards, a net torque around its axis in the direction of rotation, and the force gravity. Figure 2.3 shows a diagram of these effects.

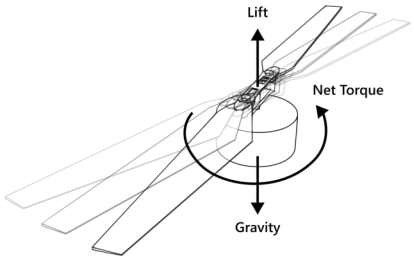


Figure 2.3: Forces and torques that are applied on a rotor.

Knowing this, the dynamic effects on a quadcopter can be represented with five forces and four torques, corresponding to the four lifts of the rotors, gravity (the mass of each of the rotors

and the body is constant, so gravity can be represented as a single force) and four torques on each of the rotors. In the real world there are many more forces and torques that act on a quadcopter such as ground effect and turbulent flow, however their effect is negligible in most contexts, so the aforementioned ones are the only dynamic effects that are relevant in this document.

2.1.1.2 Control

Quadcopters are underactuated mechanical systems because the number of rotors or actuators they contain are less than the number of degrees of freedom that they can navigate in (there are six degrees of freedom in three-dimensional space), which means that certain actions such as moving sideways cannot be done in such a way that the quadcopter translates without rotating. This, as well as the influence of gravity and the fact that quadcopters are only designed to thrust in one direction makes the task of controlling a quadcopter difficult.

The quadcopter is designed so that varying the forces and torques on each of the rotors in specific manners will cause it to either rotate around its vertical axis (also known as changing the yaw of the quadcopter), rotate around its horizontal axes (changing the roll and pitch of the quadcopter) or move up and down its vertical axis (changing the thrust of the quadcopter). Figure 2.4 shows these actions. By carefully combining these actions a pilot can move the quadcopter to any position and rotation in three-dimensional space.

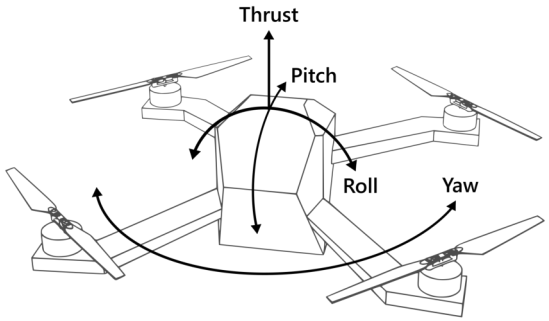


Figure 2.4: The relationship between thrust, yaw, pitch and roll.

In order to get the quadcopter to remain motionless in the air (hover), the forces on each of the rotors must counteract the force of gravity while the quadcopter is facing parallel to the ground. This is achieved by controlling the quadcopter in such a way that the magnitudes of each of the rotor forces are equal to a quarter of the magnitude of gravity. When this occurs, the sum of the forces on the quadcopter equals zero and the quadcopter does not accelerate. Figure 2.5 shows the free body diagram of a quadcopter that is hovering.

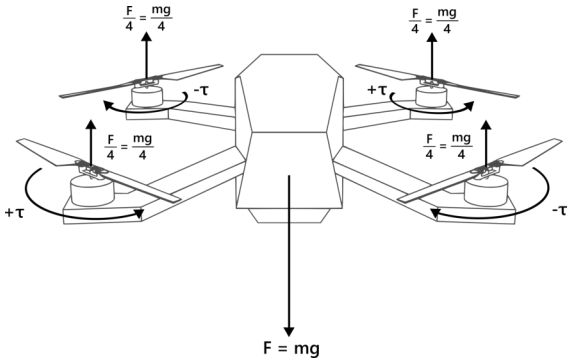


Figure 2.5: The forces and torques on the quadcopter have to be in equilibrium in order to achieve hovering. In this diagram, τ is the net torque of each of the rotors, F is the force of gravity, m is the mass of the quadcopter and g is the acceleration due to gravity. One can notice that all the torques and forces cancel each other out.

Of the motions mentioned previously, thrust is the first motion that will be discussed as it can be used to make the quadcopter move up or down its vertical axis. Upwards acceleration occurs when the signals on all four motors are increased in equal amounts above the values used while hovering, while downwards acceleration occurs when the signals are decreased in equal amounts below the values used while hovering. Figure 2.6 shows a diagram of these motions.

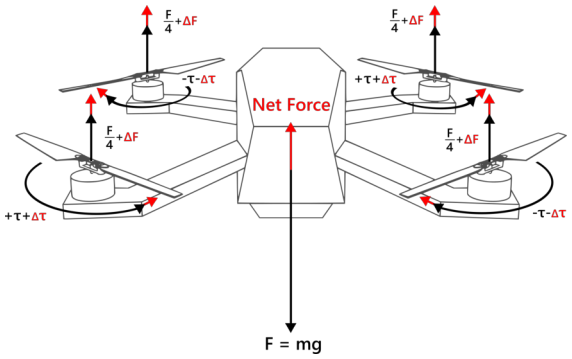


Figure 2.6: Varying the speeds of all four motors by equal amounts on top of the hovering speeds creates a net force which moves the quadcopter up or down. The torques cancel each other out, however changing the rotor speeds makes the sum of their lift forces either larger or smaller than the force of gravity, which makes the net force either point up or down.

Yaw is when the quadcopter rotates in the clockwise or anticlockwise direction around its vertical axis. Yaw is achieved by increasing the signals of two motors diagonal to each other while decreasing the signals of the other two motors. Figure 2.7 displays a diagram demonstrating how yaw can be controlled.

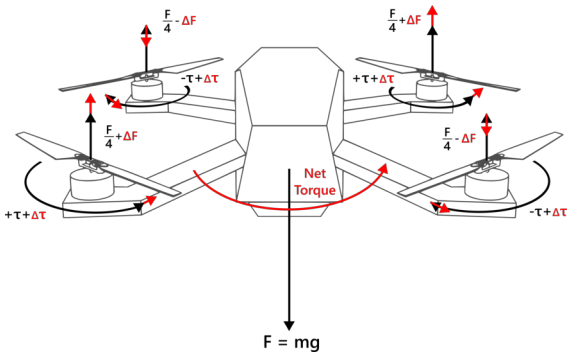


Figure 2.7: Varying the speeds of diagonal motors by equal amounts on top of the hovering speeds creates a net torque which spins the quadcopter clockwise or anticlockwise around its vertical axis. Because the speeds of the motors are added and subtracted diagonally by equal amounts, the sum of all the forces stays at zero, meaning that the quadcopter will only rotate.

Pitch is when the quadcopter rotates in the forwards or backwards directions. Pitch is achieved by increasing the signals of both the front motors while decreasing those of the back motors by equal amounts, or vice versa. This can be seen in Figure 2.8.

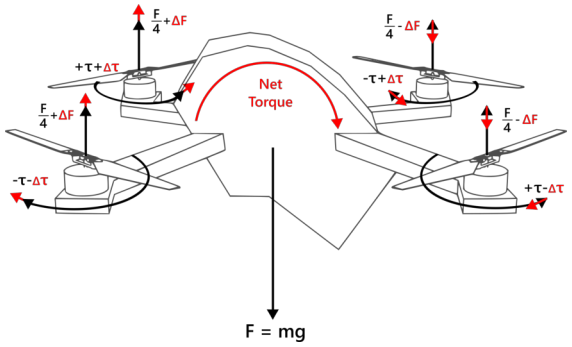


Figure 2.8: Varying the speeds of the front and back motors on top of the hovering speeds creates a net torque which spins the quadcopter forwards or backwards. Because the forces are both added and subtracted by equal amounts, the net force on the quadcopter remains at zero.

Finally, roll is when the quadcopter rotates sideways to the left or to the right. Roll can be done by increasing the signals of either the left motors while decreasing those of the right motors by equal amounts, or vice versa. Figure 2.9 demonstrates this.

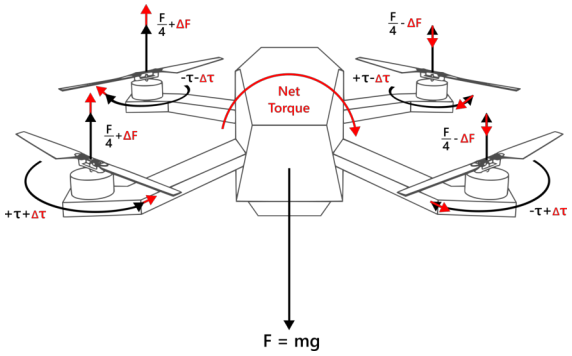


Figure 2.9: Varying the speeds of the left and right motors on top of the hovering speeds creates a net torque which spins the quadcopter sideways. Because the forces are both added and subtracted by equal amounts, the net force on the quadcopter remains at zero.

Knowing how these actions change the translation and rotation of the vehicle is vital, as these terms will regularly be mentioned throughout the rest of the document. Yaw, pitch and roll are used in this dissertation to refer to either one of the actions described above or to the angles on the axes of these motions. The context where these terms are being mentioned should make it clear to the reader which one of the two definitions is being used.

2.1.1.3 Motor Mixing

While designing a controller for a quadcopter it is often convenient to separate it into four algorithms: one for controlling thrust, one for controlling yaw, one for controlling pitch and one for controlling roll. For each algorithm, output signals for all four rotors will be generated that will only perform the action of that algorithm. These sixteen output signals can be combined into four output signals (one for each motor) with motor mixing to generate a response that performs the appropriate thrust, yaw, pitch and roll actions at the same time. Motor mixing is characterised by Equations 2.1 through 2.4, where S refers to the value of a particular motor signal and the CLAMP function is a function which clamps the value of S between its minimum and maximum values. The clamp function is given in Equation 2.5.

$$S_{\text{FRONT LEFT}} = \text{CLAMP}(S_{\text{THRUST}} - S_{\text{YAW}} + S_{\text{PITCH}} - S_{\text{ROLL}}) \tag{2.1}$$

$$S_{\text{FRONT RIGHT}} = \text{CLAMP}(S_{\text{THRUST}} + S_{\text{YAW}} + S_{\text{PITCH}} + S_{\text{ROLL}}) \tag{2.2}$$

$$S_{\text{BACK RIGHT}} = \text{CLAMP}(S_{\text{THRUST}} - S_{\text{YAW}} - S_{\text{PITCH}} + S_{\text{ROLL}}) \tag{2.3}$$

$$S_{\text{BACK LEFT}} = \text{CLAMP}(S_{\text{THRUST}} + S_{\text{YAW}} - S_{\text{PITCH}} - S_{\text{ROLL}}) \tag{2.4}$$

$$\text{CLAMP}(S) = \begin{cases} S_{\text{min}} & \text{if } S < S_{\text{min}} \\ S_{\text{max}} & \text{if } S > S_{\text{max}} \\ S & \text{if otherwise} \end{cases} \tag{2.5}$$

2.1.2 Deep Reinforcement Learning

The term reinforcement learning refers to a subset of machine learning where an algorithm learns by exploring and obtaining rewards or punishments that *reinforce* how good or bad a previous decision was. Reinforcement learning is a good approach for training quadcopters because it learns without any supervision or micromanagement, and in most cases there is no model or supervisor that knows the optimal way to fly a quadcopter.

The entity that takes decisions (in this case the quadcopter) is called the *agent*. This agent lives in an environment and is capable of making decisions depending on the current state of the environment. Once the agent takes a decision (this can be a change in the voltage of one of the motors), it receives a reward at that time or at some time in the future which is an indication of how well the agent is performing. The agent can then learn to adjust its behaviour based on this reward. Once the action has been completed the environment changes and the whole process is repeated until an end condition is met. Each process is referred to as an episode. Figure 2.10 displays this in the form of a diagram. This is more formally known as a Markov Decision Process.

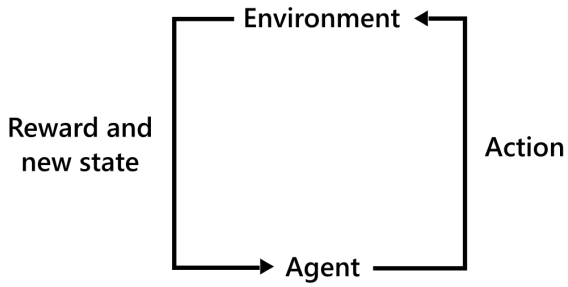


Figure 2.10: A simplified diagram of how the agent iteratively interacts with the environment.

In reinforcement learning the person that seeks to train the agent designs a reward function, which will reward or punish the agent based on the current state of the environment, so that the agent can learn which actions are good and which are bad. Rewards can be sparse, meaning that the reward is zero most of the time until something significant occurs, or rewards can be dense meaning that feedback is given all the time. It is desirable to have dense rewards because they are

easier for a reinforcement learning agent to learn from, however this is not always possible due to the nature of the environment. Sparse rewards can also be learnt from but take more time to do so because it is not immediately obvious to the agent if an action could lead to a better reward, while with dense rewards it is immediately apparent when one action is better than another. For this reason, all of the reward functions used in this dissertation are dense.

While referring to reinforcement learning, the policy of an agent is usually mentioned. According to [15], "A policy defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus–response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action."

In the case of deep reinforcement learning, the policy is a complex multi-dimensional function with inputs that represent information about the current and previous states of the environment, and one or more outputs that determine which actions to take. Such complex functions can be extremely hard to represent with mathematical tools and lookup tables, so they are instead often represented by deep neural networks.

Deep reinforcement learning uses deep neural networks to represent policies because they are capable of learning complex relationships between inputs and outputs with only a few parameters and lots of data. An additional advantage of deep neural networks is that if they are trained properly, they can generalise, meaning that they can learn to produce the correct outputs for inputs they have never seen before.

The goal of reinforcement learning is to improve the policy of the agent over time so that it can make better decisions. Neural networks are typically trained by feeding them data from a fixed dataset, however in reinforcement learning this is challenging because the data changes over

time. This occurs because different behaviours will lead to new experiences which can completely disrupt the distributions of the data. Reinforcement learning algorithms such as Q-Learning and policy gradient methods employ different strategies in order to make the deep neural networks adapt to these changes.

2.1.2.1 Policy Gradient Methods

Policy gradient methods are a subset of reinforcement learning algorithms where the goal is to maximise the performance of the current policy by performing gradient descent on the neural network behind that policy. It is very hard to compute the exact gradient of the policy's neural network with respect to its weights, so an estimate of this gradient is calculated instead. This estimate of the gradient is dependent on a value called the *advantage estimate*, which is the difference between the reward obtained by the agent and the output of a second neural network, which returns the reward it expects the agent to receive when the next action occurs.

If the advantage estimate is positive, this means that the reward obtained by the agent was better than the expectation produced by the second neural network, and if it is negative then the reward obtained was worse than the expectation.

Because the estimate of the gradient depends on the advantage estimate, decisions that lead to better than expected results become likelier to be selected by the policy's neural network, while decisions that lead to worse than expected results become less likelier to do so.

The neural network that estimates the reward that the agent will get is also trained over time. This is done by providing it with the previous state as the input and the reward that was obtained as the ground truth.

2.1.2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a state-of-the-art reinforcement learning algorithm by OpenAI designed to improve on traditional Policy Gradient Methods. [14]

PPO is based on an algorithm called Trust Region Policy Optimization (TRPO) which limits the size of the steps taken during gradient descent of the neural network of the policy using KL-divergence. TRPO limits this step size because the advantage estimate and the estimate of the gradient can be noisy, which can lead to divergence during gradient descent while taking large steps.

PPO innovates on TRPO by using a clipping function instead of KL-divergence to limit the size of the steps taken during gradient descent. The clipping function has significantly less overhead than KL-divergence and is easier to implement.

The objective function of PPO is provided in Equation 2.6 [14]. The goal is to maximise this function. The symbol L^{CLIP} refers to the PPO objective function, θ is the set of parameters that controls the policy or the weights of the policy's neural network, $\hat{\mathbb{E}}_t$ is the expected value function (which can be considered an averaging function), $\text{MIN}(x_0, x_1)$ is the minimum function that returns the smallest of its inputs, $\text{CLIP}(x, x_{\min}, x_{\max})$ is the clipping function where the first parameter is restricted between the second parameter and the third parameter (see Equation 2.7), r_t is the ratio between the probability of the action occurring under the current policy and the probability of the action occurring under the old policy, $\hat{A}_t$ is the advantage estimate and ϵ is a PPO hyperparameter used to control how far a new policy can deviate from an old one. It is worth noting that the minimum and clipping functions work together to constrain the value of $r_t(\theta)$ between $1-\epsilon$ and $1+\epsilon$. This is the part of PPO which limits the step size taken during gradient descent.

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{CLIP}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.6}$$

$$\text{CLIP}(x, x_{\min}, x_{\max}) = \begin{cases} x_{\min} & \text{if } x < x_{\min} \\ x_{\max} & \text{if } x > x_{\max} \\ x & \text{if otherwise} \end{cases} \tag{2.7}$$

One can observe that when the function is not clipped, the loss can be simplified to $r_t(\theta)\hat{A}_t$, which is the product of the ratio r_t and the advantage estimate. The probability ratio r_t can be seen as a metric of how probable an action is likely to occur, so by multiplying this value by the advantage estimate, if the advantage estimate is positive (the agent performed better than expected) then r_t will be increased, making the action likelier to occur, and if the advantage is negative (the agent performed worse than expected) then r_t will be decreased, making it less likelier to occur.

Choosing larger values for ϵ will allow the PPO algorithm to deviate more from old policies with the risk of diverging, while reducing this value will cause the clipping function to act, which can lead to more stable changes at the risk of updating too slowly or converging early. For these reasons, this hyperparameter has to be chosen carefully in order to achieve an optimal functionality.

2.1.3 Proportional-Integral-Derivative Controllers

Proportional-Integral-Derivative controllers, commonly known as PID controllers, measure the value of an output $y(t)$ of a system and vary the input $x(t)$ to the system until $y(t)$ is equal to a desired value $r(t)$. The input $x(t)$ is adjusted based on the difference between $y(t)$ and $r(t)$. The desired value is commonly known as the *setpoint*.

The PID algorithm first calculates the error or difference between the setpoint and the current output $e(t) = r(t) - y(t)$ at the current time t, its integral $\int_0^t e(T)dT$ and its derivative $\frac{de(t)}{dt}$. These three values are then multiplied by a set of constants K_p, K_i and K_d, and added together to give the output value. This is the reason why it is called a Proportional-Integral-Derivative controller. The equation of a PID controller is displayed in Equation 2.8, where $u(t)$ is the output value of the controller.

$$x(t) = K_p e(t) + K_i \int_0^t e(T)dT + K_d \frac{de(t)}{dt} \tag{2.8}$$

The proportional term $K_p e(t)$ of the PID controller corrects the output depending on how large the error is; a larger error will lead to a greater change in the input and consequently the output, while a smaller error will lead to a smaller change. This method works well in practice but on its own it is not enough to produce a stable output, as in most cases it will cause the output to oscillate above and below the desired value while never converging to it.

The derivative term of the PID controller $K_d \frac{de(t)}{dt}$ helps solve this problem because it adjusts the output proportional to its error. This term usually acts as a dampener, making the output converge to the desired output.

A final problem that tends to appear is that with only the proportional and derivative terms, the output eventually converges to a value slightly above or below the setpoint and stays there. The error at this point is referred to as the *steady state error*. This is solved with the integral term $K_i \int_0^t e(T)dT$, which adds up the error over time to ensure that the system moves towards the setpoint even if the proportional and derivative terms have settled down.

The constants that multiply each of the error terms are called the *gains* of the controller. The optimal value for these gains varies depending on the system. There is no ideal method for finding these gains, however there are several methods such as the Ziegler–Nichols, Tyreus Luyben and Cohen–Coon methods which can be followed to find gain values that follow a certain behaviour or that meet certain constraints.

When one of the gains of a PID controller is zero, people often refer to these controllers by the initials that correspond to non-zero gains. For example, a PD controller has an integral gain of zero, while a PI controller has a derivative gain of zero.

2.1.4 Genetic Algorithms

A genetic algorithm optimises a set of parameters by mimicking behaviours seen in genetics and nature. Each set of parameters is called a chromosome, and multiple chromosomes form a population. The idea behind a genetic algorithm is to breed sets of two chromosomes by combining their values in order to create child chromosomes. These child chromosomes then

undergo mutation which involves modifying their values with a small random probability, followed by inserting these children into the population.

All of the chromosomes are evaluated and assigned a different fitness, which is a metric of how good that chromosome/set of parameters is. The least fit chromosomes are removed from the population (survival of the fittest), leaving chromosomes with increasing values of fitness after each iteration or generation. In order to ensure that good children are produced, chromosomes are selected to breed with each other with a higher likelihood if they have better fitness values.

At the beginning of the genetic algorithm, chromosomes are initialised with random values in order to ensure there is enough variation between chromosomes. As the algorithm runs, these chromosomes will slowly converge to an optimal solution, while also exploring different possibilities due to mutation. This is useful because it can prevent the genetic algorithm from falling into local minima.

A genetic algorithm is used in this project to optimise the gains of several PID controllers. This is explained in more detail in Section 3.3.2.

2.2 Related Work

Cano Lopes et al [2] show that Proximal Policy Optimization can be used to train a deep reinforcement learning agent to fly a quadcopter. They analyse the behaviour of the quadcopter under three situations, one where the policy of the quadcopter is stochastic and the quadcopter must reach a fixed target, one where the policy of the quadcopter is deterministic and must also reach a fixed target, and one where the quadcopter must reach a moving target with a deterministic policy.

Their results show that Proximal Policy Optimization did not work well for the stochastic policy and the deterministic policy with a moving target because the quadcopter had high error values. Nevertheless, the Proximal Policy Optimization agent showed desirable results for the deterministic policy with a fixed target.

Hsiao, Chiang and Hou [6] present in their report a comparison between two deep reinforcement learning algorithms, Advantage Actor-Critic (A2C) and Proximal Policy Optimization, and PID controllers for the task of hovering a quadcopter. Their results show that PID controllers obtain a reward of almost 600 points, while PPO obtains a reward of slightly more than 500 points and A2C finishes with a reward of slightly less than 300 points.

The reward function used in their report is displayed in Equation 2.9, where R_t is the reward at a time t, $\mathbf{x}$ is the position of the quadcopter, $\mathbf{x}_{goal}$ is the position of the target, θ is a vector representing the rotation of the quadcopter, and ω is a vector of the angular velocity of the quadcopter. C_θ and C_ω are constants used to change the importance of reducing the final rotation and angular velocity of the quadcopter.

$$R_t = \max(0, 1 - ||\mathbf{x} - \mathbf{x}_{goal}||) - C_\theta||\theta|| - C_\omega||\omega|| \tag{2.9}$$

The reward function proposed in this dissertation seeks to improve on the one presented above by scaling the distance, angle and angular velocity values by exponents, which changes the shape of the reward function to ease learning.

Tiwari [16] uses a combination of PID controllers with motor mixing to control a quadcopter which must track a moving ground vehicle. Tiwari implements an attitude controller which consists of four PID controllers that each adjust the height, yaw, pitch and roll of a quadcopter given desired values for these parameters. Tiwari then uses a position controller which given the X and Y distance errors to the target returns values for pitch and roll that are sent to the attitude controller. This has the effect of turning the quadcopter towards the target in order to move to it. This dissertation uses the same structure of PID controllers to control the quadcopter.

Tiwari tunes their PID controllers with the Ziegler-Nichols method, which allows one to change the characteristics of the PID response such as overshoot, however this dissertation seeks to innovate on this by using a novel implementation of a genetic algorithm to find the gains which maximise the reward or fitness of the quadcopter automatically. An advantage of using a genetic algorithm is that it can avoid local minima.

The results obtained by Tiwari show that using a Feedforward PID controller with Kalman filters can lead to very high accuracies while following a path that varies in the X, Y and Z components. While varying all three of these values, Tiwari showed that mean squared errors of 0.0363, 0.5864 and 0.0649 can be obtained for these three coordinates respectively.

Koch et al. [9] compare the Deep Deterministic Gradient Policy, Trust Region Policy Optimization and Proximal Policy Optimization algorithms for controlling a quadcopter against each other and PID controllers. Their results show that Proximal Policy Optimzation obtains better rewards and converges faster than the other algorithms. The results also show that in many cases the Proximal Policy Optimzation agent outperforms the PID controllers.

Hu and Wang [7] use Proximal Policy Optimization with an integral compensator to train a reinforcement learning agent which is compared to PID controllers. Their results show that the use of an integral compensator makes the Proximal Policy Optimzation agent learn at a faster rate than without the compensator while also converging to a higher reward value. Hu and Wang apply a set of tests where the quadcopter must fly to target positions while carrying loads of different weights and show that the PID controlled quadcopters have more instability as the weight of the load increases, while the PPO with integral compensator agents show stability regardless of the load.

CHAPTER 3. METHODS AND PROCEDURES

3.1 Simulation Environment

The first requirement was to create or find an environment capable of simulating a virtual quadcopter. One problem with any simulation is verifying that it is accurate to the real world, because inaccuracies will inevitably lead to false conclusions. The ideal method of ensuring that the simulation is correct is by comparing the simulation to reality and making the appropriate adjustments to mitigate the differences. For this project, this could not be done due to the limitations mentioned in Section 5.2. An alternative method is taken where realism is ensured by using scientific models and equations that have been proven to be accurate.

Unity, a free game engine that makes use of the Nvidia PhysX physics engine, was used to create the simulation. This program contains many useful features which allow developers to quickly test and deploy games. Despite being marketed as a game engine, Unity has its own set of tools that can be used to simulate robots, those of which are being used in the industry by companies such as Siemens [1] and Ready Robotics [12].

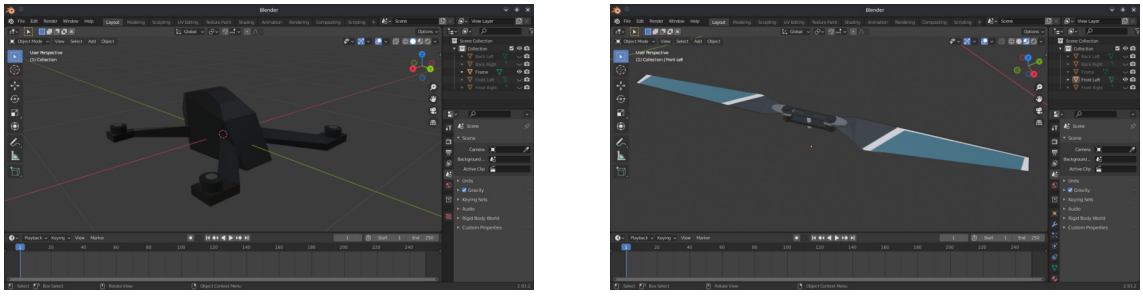
One tool used in this project is a package called ML-Agents. It is capable of training deep reinforcement learning agents within the Unity game engine. This package is open source and maintained by a group of researchers and collaborators from the Unity community [8]. It is worth mentioning that this package is not included in Unity by default, however it can be downloaded from within the program.

In order to understand how the simulation works, a few concepts will have to be explained. Every physical object in Unity is called a GameObject, and each of these objects has their own mesh which is the 3D model that is visualised in the environment. This 3D model can be used to test for collisions between other GameObjects and to calculate geometric properties such as

moment of inertia, therefore it is important that an accurate model is used. For this dissertation the author used the Blender modelling program to create the mesh models.

The mesh of the body of the quadcopter is based around DJI's FPV quadcopter [4] because it is commercially popular and there are many reference documents provided by DJI that specify physical parameters that are used by the simulation. One strange thing about this quadcopter is the fact that it does not have a flat bottom, but rather a triangular one. This was changed in the model the author created in order to allow the quadcopter to land upright on the ground. An image of the mesh the author created without the propellers can be seen in Figure 3.1a.

One design choice involved changing the propellers because there is little information on the ones used by the DJI FPV quadcopter and the motor that actuates them. DJI has a well-documented propeller and motor combination called the E5000 [3], which is used for this project instead. The mesh the author created for this propeller can be seen in Figure 3.1b. One should note that special care was taken to ensure that the dimensions of each of the meshes match those of the real-life models.



(a) DJI FPV body mesh.

(b) E5000 propeller mesh.

Figure 3.1: Quadcopter models created in Blender.

All of the dimensions in Unity are in SI units, meaning that distance is measured in metres, time in seconds, angles in radians, mass in kilograms and voltages in volts.

Another concept is that of steps, which are used for the reinforcement learning agents. ML-Agents considers the agent to have entered a new state after a step, and steps can be configured to occur every certain amount of physics updates. For this project one step has been

configured to happen every time the physics engine updates in order to increase the performance of the quadcopter. Each physics update can also be configured to occur every certain amount of milliseconds inside the simulation; in this case this value was chosen to be every 20 milliseconds because increasing the time step leads to inaccurate results with very little benefit.

3.1.1 Code Overview

One of the biggest advantages of Unity is that it uses a scripting API based on the C# programming language which allows one to create scripts that can run algorithms and control objects in the environment. Practically all the code created for this dissertation was written for this API.

In order to understand how the simulation works, the diagram in Figure 3.2 shows a simplified model of how the scripts are connected between each other and the physics engine, and what kind of data they send between each other.

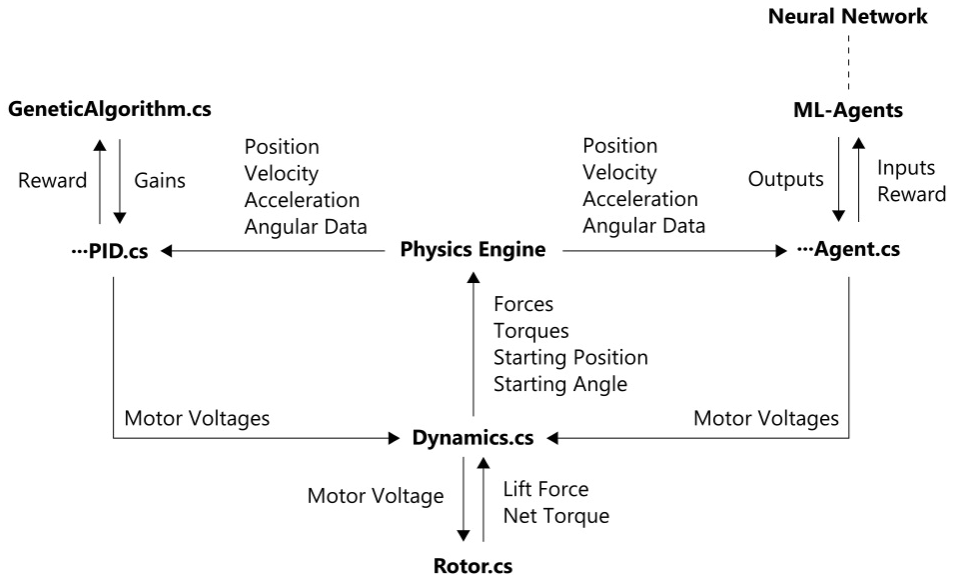


Figure 3.2: This diagram displays how each of the scripts interact with each other in order to update the simulation and work with the PID and PPO agents.

There are a few scripts that have been omitted from this diagram either because they are not very relevant to the simulation or because they are small helper scripts. The names `···Agent.cs` and `···PID.cs` refer to all scripts that end in `Agent.cs` and `PID.cs`. For example, `···PID.cs` can refer to `HoverPID.cs` and `PathPID.cs`.

The `Dynamics.cs` script communicates directly with the physics engine, sending it the forces and torques to be applied on the quadcopter. When the simulation starts it also sends the starting position and angles for the quadcopter.

The `···Agent.cs` scripts correspond to different deep reinforcement learning agents. These scripts receive the state of the environment from the physics engine and use this state to calculate the reward at the current step and the inputs that it will send to the neural network. These values are sent through ML-Agents, which acts as an interface between the script and the deep neural network. ML-Agents sends the inputs to the neural network and receives the outputs, which it sends back to the `···Agent.cs` scripts. These scripts then convert the outputs into motor voltages which are sent to `Dynamics.cs`. When specified to do so, ML-Agents will also train the neural network by updating it based on the inputs and rewards received.

A similar thing occurs with the `···PID.cs` scripts, which represent the PID controllers. These controllers receive data from the physics engine that is used to calculate the current yaw, pitch, roll and height of the quadcopter. The scripts then compare these values to the desired ones and use the PID and motor mixing algorithms in order to produce motor voltages which are sent to `Dynamics.cs`.

The PID controllers are trained through the `GeneticAlgorithm.cs` script, which is a genetic algorithm that optimises the proportional, integral and derivative gains of the controllers based on reward. The genetic algorithm first creates many agents with different gains which are sent to the `···PID.cs` scripts. Subsequently the agents interact with the environment and each return rewards, which are sent back to `GeneticAlgorithm.cs`. The genetic algorithm compares the rewards and uses them to create the gains for the next episode.

One should note that · · ·`Agent.cs` scripts are only used in copies of the quadcopter controlled by reinforcement learning agents and · · ·`PID.cs` scripts are only used in copies that are controlled by PID controllers. Only one of these two sets of scripts is used at any time.

The · · ·`Agent.cs` and · · ·`PID.cs` scripts send the motor voltages for each of the four propellers to `Dynamics.cs`. This script keeps four copies of the `Rotor.cs` script, one for each rotor, and sends the voltages to this script. The `Rotor.cs` script then uses the motor and aerodynamic equations from Sections 3.4.1 and 3.4.2 to calculate the lift force and net torque on that rotor. These values are then sent to `Dynamics.cs` and processed before being sent to the physics engine.

3.2 Tasks

The PID controllers and deep reinforcement learning agents are quantitatively analysed with a set of three tasks; a hovering task where the quadcopter must hover and stay still at a precise point in space, a path following task where the quadcopter must move around a curved path with as little deviation as possible and a landing task where it must learn to land at a precise point while minimising its speed when it touches the ground. Each of these three tasks have different inputs and reward functions, which is why there are multiple · · ·`Agent.cs` and · · ·`PID.cs` scripts. These tasks are explained in detail in the following sections.

3.2.1 Hovering

For this task the quadcopter is given a target position at which it must hover. In the case of the deep reinforcement learning agent, the inputs to the neural network are listed below:

- **Yaw** - The yaw angle of the quadcopter normalised between -1 and 1.
- **Pitch** - The pitch angle of the quadcopter normalised between -1 and 1.
- **Roll** - The roll angle of the quadcopter normalised between -1 and 1.

- **Distance** - The scalar value of the distance between the quadcopter and the target, divided by a value which represents the maximum distance the quadcopter can travel in order to normalise the value between 0 and 1.
- **Speed** - The scalar value of the speed of the quadcopter (magnitude of the velocity vector), divided by the maximum value the quadcopter can travel in order to reduce the value to near the 0 to 1 range.
- **Angular Speed** - The scalar value of the angular speed of the quadcopter (magnitude of the angular velocity vector), divided by the maximum value the quadcopter can travel in order to reduce the value to near the 0 to 1 range.
- **Error Vector** - A vector pointing from the quadcopter towards the target, normalised by component.
- **Velocity Vector** - The vector of the velocity of the quadcopter, normalised by component.
- **Angular Velocity Vector** - The vector of the angular velocity of the quadcopter, where the direction is the axis of revolution and the magnitude is the speed it rotates in that direction, normalised by component.

The distance and component normalised error vectors are favoured over providing the agent with the positions of the quadcopter and the target because it is easier for the reinforcement learning algorithm to teach the agent to reduce the former two values than to find a relationship between the latter two vectors. Additionally, in the first case only four values are used as inputs (distance, error x, error y and error z) while in the second case six values have to be used (x, y, and z values for both positions), which makes the process simpler. One might consider omitting the distance and using the error vector without normalisation as an input to further reduce the number of inputs, however ML-Agents strongly advises the use of normalisation as this leads to faster convergence. [10]

One can note that the vectors are normalised by component. This is different to traditional vector normalisation, where all the components are divided by the magnitude of the vector. Normalisation by component in this case refers to rescaling all the components by the same factor so that the smallest component has a value of -1 or the largest component of the vector has a value of 1. This is recommended by ML-Agents in their Agent class documentation [10].

Angular data such as yaw, pitch, roll, angular speed and the component normalised angular velocity vector are all sent to the agent in order to help it learn to adjust its angle. Additionally, the speed and component normalised velocity vectors are sent as inputs in order to help the quadcopter learn to manipulate the direction its position changes.

The positional and angular accelerations are not used as inputs to the agent because these are directly proportional to the net force and torque on the quadcopter, which are in turn related to the outputs of the agent. The environment satisfies the Markov property, meaning that the quadcopter does not gain any additional information from previous positional or angular data.

The reward function for hovering is designed to be equal to one if the quadcopter acts perfectly and equal to zero if it performs infinitely bad. A reward of one can only be achieved if the quadcopter moves instantly to the target, staying completely still and facing the right angle. In practice it is not possible for the quadcopter to get a reward of zero or one, however this function can be used to obtain a metric of how ideal the performance of the quadcopter is.

By maximising the reward function the quadcopter must learn to minimise the distance, the yaw angle and the angular speed of the quadcopter. The reward function for hovering at a time step t is given in Equation 3.1, where d is the distance between the quadcopter and the target in metres, $|\varphi|$ is the absolute value of the yaw error of the quadcopter in radians and ω is the angular speed of the quadcopter in radians per second. These three values are positive, which when put in the exponential term e^x makes the reward function map all values from 0 to infinite between 0 and 1.

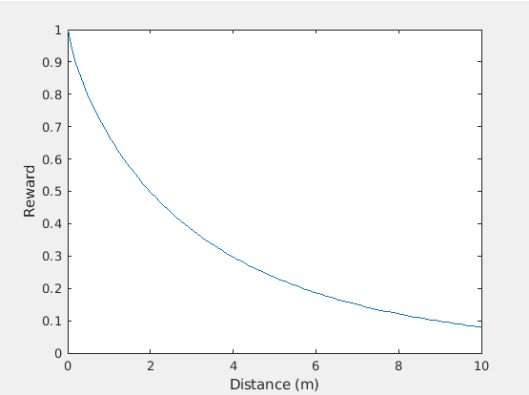
$$R(t) = e^{-0.4d(t)^{0.8} - 1.0|\varphi(t)|^{1.4} - 0.8\omega(t)^{1.2}} \tag{3.1}$$

This reward is added to the total reward at every time step and divided by $t_{\max}$, the number of time steps in the simulation, in order to ensure that the total reward is between 0 and 1.

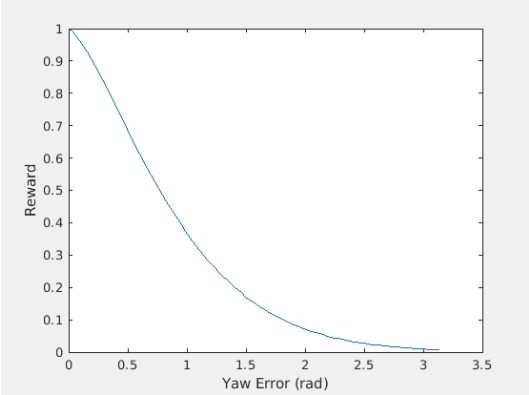
$$R = \sum_{t=0}^{t_{\max}} \frac{R(t)}{t_{\max}} \tag{3.2}$$

This style of reward function was chosen for several reasons. One can observe that distance, yaw and angular speed are all raised to different powers less than one. This has the effect of changing the shape of the reward function, making it either sharper if the exponent is closer to zero or smoother if it is closer to one. Sharper reward functions are preferrable in cases where it is more important to achieve the best performance possible, while smoother reward functions are better in cases where decent performance is good enough.

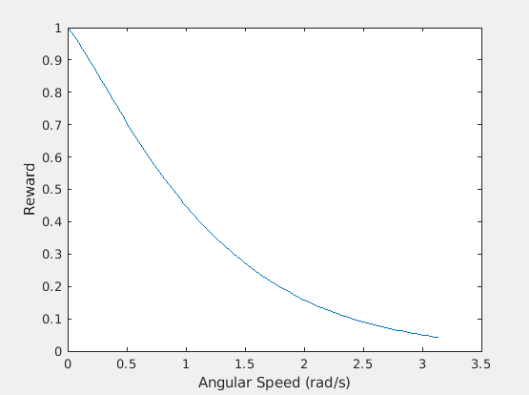
Visualising the reward function is difficult because it has three inputs and an output, however one can get a general idea of the shape of the reward function by looking at plots of the output where only one parameter is changed at a time. The graphs in Figure 3.3 show these plots.



(a) Plot of the reward obtained by varying distance when the yaw error and angular speed are zero.



(b) Plot of the reward obtained by varying yaw error when the distance and angular speed are zero.



(c) Plot of the reward obtained by varying angular speed when the distance and yaw error are zero.

Figure 3.3: One can get a general idea of the shape of the reward function by visualising the reward when only one parameter is changed.

Distance has the smallest exponent of the three variables in the exponential term of the reward function, meaning that the reward changes the sharpest while changing this value. Because the function is sharper, the agent will receive a small reward if it optimises its distance from being very far to quite far, but it will receive a significant reward if it optimises its distance from very close to extremely close. This has the effect of prioritising the minimisation of the distance function above all other tasks. One can note that the exponent of the yaw is the highest, because minimising the yaw is not the highest priority of the reward function.

The distance, yaw and angular speed values are also multiplied by different constants, which have the value of stretching or squashing the reward function. These constants have been selected to make the reward function span different ranges of distance, yaw and angular speed appropriately. These terms are all in a negative exponent because when all the terms are optimised to zero then the reward is one, and when any or all the terms are infinite then the reward is zero. This is ideal because the reward function is smooth and has a descending gradient, which should make training easier.

In the case of the PID controllers, a genetic algorithm is used to select a set of gains in such a way that the same reward function from Equation 3.1 is maximised. The genetic algorithm is explained in detail in Section 3.3.2. The quadcopter makes use of six PID controllers listed below:

- **Thrust Controller** - This PID controller receives a height in metres as the setpoint, and uses the measured height of the quadcopter (the Y coordinate in Unity) in order to make the quadcopter hover at the setpoint's height. The output of this PID controller is a voltage signal which is mixed with the other signals before being sent to the motors.
- **Yaw Controller** - This PID controller adjusts the yaw of the quadcopter. The setpoint of this controller is always zero, which corresponds to a yaw where the quadcopter is facing towards the X axis. This means that the goal of this PID controller is to minimise the yaw error. The output of this PID controller is a voltage signal that is mixed with the other signals.
- **Pitch Controller** - This PID controller adjusts the pitch of the quadcopter. The setpoint of this controller is not zero but rather the output of another controller which is used to change the position of the quadcopter relative to the XZ plane in Unity. It is not possible for the quadcopter to change its horizontal speed without first rotating towards the direction it wants to move in. This means that two more controllers have to be used to adjust the pitch and roll of the quadcopter so that it rotates towards the target while also converging at it. The output of this controller is a voltage signal which is mixed with the other voltages.
- **Roll Controller** - Similarly to the pitch controller, this PID controller adjusts the roll of the quadcopter based on the output of another controller used to move in the XZ plane. The output of this controller is a voltage signal mixed with the rest of the signals.
- **Positional Pitch Controller** - This controller is used to generate an angle which is sent to the pitch controller mentioned above. This controller receives a measurement of the X distance to the target relative to the body frame as input and compares it to a setpoint of zero. The body frame of the quadcopter is a frame of reference or coordinate system which

is aligned to the position and rotation of the quadcopter, which means that the X axis in this frame points in the same direction the quadcopter is facing. The body frame has to be used because the quadcopter will occasionally have a yaw which is not facing forward, which affects the absolute X distance between the quadcopter and the target.

- **Positional Roll Controller** - This controller is used to generate an angle which is sent to the yaw controller. It receives a measurement of the Y distance between the quadcopter and the target in the body frame and compares it to a setpoint of zero.

The positional pitch and roll controllers have their output angles clamped between -0.2 and 0.2 radians, which becomes the maximum pitch and roll allowed by the quadcopter, because inclining the quadcopter at angles outside of this range can cause it to lose control.

The thrust, yaw, pitch and roll controllers all generate voltage signals which are combined using the motor mixing equations (see Section 2.1.1.3) in order to generate the voltages for all four of the motors.

The PID gains used in the yaw, pitch and roll controllers are shared because these are all almost identical in functionality and produce angles. The PID gains in the positional controllers are also shared because of similarity. This means that only three sets of gains (nine gains in total) have to be found by the genetic algorithm.

In order to stop the reinforcement learning and genetic algorithms from overfitting, randomness is introduced into each of the testing environments. In the case of hovering, the target is placed at the origin and the quadcopter is placed at a random point on the surface of a sphere centred around the origin. The radius of this sphere is of five metres. The quadcopter is also initialised with a random yaw from 0 to 2π radians, a random pitch from $-\frac{\pi}{4}$ to $\frac{\pi}{4}$ radians, and a random roll from $-\frac{\pi}{4}$ to $\frac{\pi}{4}$ radians as well.

3.2.2 Path Following

The objective of this task is to create an environment where the quadcopters capacity to follow a path can be measured. There are several approaches that can be followed to do this,

however the one used for this project is relatively simple because it involves following a target which moves steadily along the path.

The path is represented by a parametric curve or equation, which has three components $x(t)$, $y(t)$ and $z(t)$ that are all dependent on an independent variable t, for all $0 \leq t \leq 1$. In order to introduce randomness into this environment, the quadcopter is not placed at a random position but rather at the start of the parametric curve, and the shape of the curve is instead randomised by varying a set of parameters.

The shape of the curve is based around an inclined figure of eight because this shape has several sections with different curvatures, and it intersects itself at the centre which makes following it accurately somewhat complex. There are three parameters a, b and c that are introduced into the curve to allow randomisation. The a and c parameters stretch and compress certain parts of the curve, while the b parameter changes its vertical incline. The parametric equation for the curve is defined by the equations below. Note that $y(t)$ refers to the Y coordinate in Unity which is the vertical coordinate.

$$x(t) = 5\cos(2\pi t + a) \tag{3.3}$$

$$y(t) = 5b\min(t, 1-t) \tag{3.4}$$

$$z(t) = 5\cos(2\pi t)\cos(2\pi t + c) \tag{3.5}$$

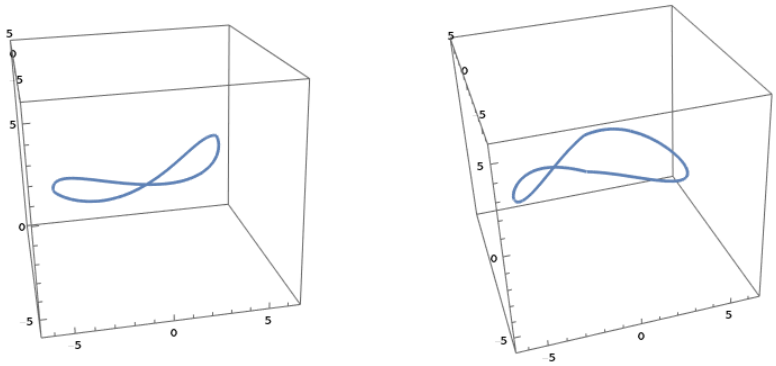
At the beginning of each episode, random values for a, b and c are generated based on the following constraints:

$$0 \leq a \leq \frac{\pi}{2} \tag{3.6}$$

$$-1 \leq b \leq 1 \tag{3.7}$$

$$\frac{\pi}{4} \leq c \leq \frac{3\pi}{4} \tag{3.8}$$

These constraints are defined because values outside of these ranges may create asymptotes or curves which are too large for the quadcopter to follow. Figures 3.4a and 3.4b show how the path can vary by changing only one parameter.



(a) The parametric curve when $a = 0$, $b = -1$ and $c = \frac{\pi}{4}$.

(b) The parametric curve when $a = \frac{\pi}{2}$, $b = -1$ and $c = \frac{\pi}{4}$.

Figure 3.4: Different randomisations of the curve change its shape.

The deep reinforcement learning agents and PID controllers for this task are trained with the same equation as the hovering task (Equation 3.1), because the objective of the quadcopter is quite similar; to reach a target point (however this time the target is constantly moving).

The PID controllers for this task are also the same as with hovering, receiving the same input and having controllers for thrust, yaw, pitch, roll, positional pitch and positional roll. Training these controllers for path following instead of hovering should yield a different set of gain values that are more suited to following the moving target.

The reinforcement learning agent on the other hand has the advantage of being able to receive more inputs from which it can learn. For this task the agent receives the same 15 inputs as with hovering and an additional six inputs in the form of two vectors; the direction of the velocity of the target and the direction of the acceleration of the target, both normalised by component. In theory, the agent should be able to make use of these vectors to predict where the target will move to, and subsequently learn how to improve its performance with that prediction.

3.2.3 Landing

The objective of the landing task is for the quadcopter to descend from an elevated point towards the ground and land on a specific point while minimising the speed at which it hits the ground. The quadcopter is initially placed in a random position on a ring with a five metre radius that is also five metres above the ground.

The reward function for this task is the same as the one used in the hovering task (Equation 3.1) with an extra term added for the speed v:

$$R(t) = e^{-0.4d(t)^{0.8} - 1.0|\varphi(t)|^{1.4} - 0.8\omega(t)^{1.2} - 1.1v(t)^{0.7}} \tag{3.9}$$

This way, the objective of this reward function is to minimise the distance d to the target point on the ground, the absolute value of the yaw error $|\varphi(t)|$, the angular velocity when it touches the ground $\omega(t)$ and the speed when it touches the ground v. The reward obtain by varying this additional term is plotted in Figure 3.5 to give the reader an idea of how this term affects the shape of the reward function.

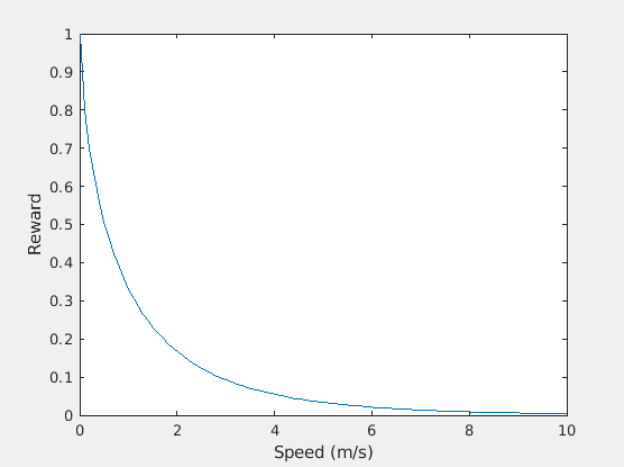


Figure 3.5: Plot of the reward obtained by varying speed when the values of distance, yaw error and angular speed are zero.

The episode ends when the quadcopter touches the ground, however this creates a problem because the quadcopter will lose any points that it can gain by waiting before touching the

ground. This can lead to the quadcopter learning to hover above the ground without ever actually touching it. In order to disincentivise this behaviour, the reward at the last time step when it touches the ground is saved and multiplied by the number of time steps that were remaining until the end of the episode before being added to the rest of the reward.

The PID controllers used for landing are the same as the ones used for hovering, with the same inputs, and the deep reinforcement learning agents also have the same inputs as the hovering task.

3.3 Training

This section goes over the specifics of the training methods used.

3.3.1 Deep Reinforcement Learning

The deep reinforcement learning agents are trained through ML-Agents, which uses the Python programming language and the Tensorflow library in order to create the neural network and perform the appropriate calculations. Agents can be trained in parallel within a single environment, doing this has the advantage of decreasing the training time.

Training can be done within the Unity editor or by creating a headless binary file. For this project, several binary files were created and uploaded to the High Performance Computing Facility (Ceres) provided by the University of Essex, so that multiple environments could be trained at the same time on different processors. Within each simulation environment, 1024 quadcopters are trained in parallel because this speeds up the training process and 1024 is roughly the largest amount of quadcopters that can be placed in the simulation before affecting performance.

Different combinations and values of hyperparameters were tested before training the agents in order to find the optimal set of values that would maximise the reward. More information about these hyperparameters can be found in Section A.1 of the appendix.

3.3.2 Genetic Algorithm

In order to obtain an optimal set of gains for each of the PID controllers while maximising for reward, a genetic algorithm was created. This genetic algorithm was configured to create a population of 64 quadcopters which are all run in parallel within the simulator, with chromosomes of nine genes which are used to hold the gains for the six PID controllers.

In order to ensure that the best individuals of each generation survive, the genetic algorithm keeps the 40% of the population that have the highest fitness values. These individuals are selected to breed with each other using roulette wheel selection, which makes chromosomes with higher values of fitness more likely to breed than others. Chromosomes are bred using uniform crossover, where two children are generated and for each of their genes they have a 50% chance of inheriting a gene from either parent.

All of the new children then undergo two types of mutation, uniform mutation (a gene is replaced with a brand new random gene) and gaussian mutation (a random increment is added to an existing gene), with each gene having a 20% chance of being mutated with gaussian mutation followed by a 20% chance of being mutated with uniform mutation. This allows the algorithm to introduce a large amount of variation into each of the individuals, which helps to prevent the population from being saturated with copies of a single chromosome.

The 60% of the population that did not perform as well as the other individuals are eliminated, and children are added until they fill the space left by the eliminated chromosomes. Once this new generation is complete, each of the chromosomes are assigned to one quadcopter, which updates the gains of its PID controllers to the values of the chromosome. The quadcopters are simulated and at the end of the episode they are all assigned rewards, which become the fitness values for their chromosomes. The chromosomes are then sorted by how fit they are and the genetic algorithm moves on to the next iteration until 250 generations have passed.

A problem with using the genetic algorithm to train the quadcopter is that the randomness used to avoid overfitting can lead to noise in the rewards obtained by the PID controlled quadcopters. As an example of why this is a problem, if a quadcopter gets lucky due to

randomness it may produce a reward which is very good despite having a bad set of gains. A quadcopter with a very good set of gains can also get very bad luck which could prevent it from getting a high enough reward to stay in the gene pool. Figure 3.6 shows a graph of the fitness at each generation while training the PID controllers of the landing task without any countermeasures for the noise. In this figure it is evident that the effect of noise is detrimental to the training of the system.

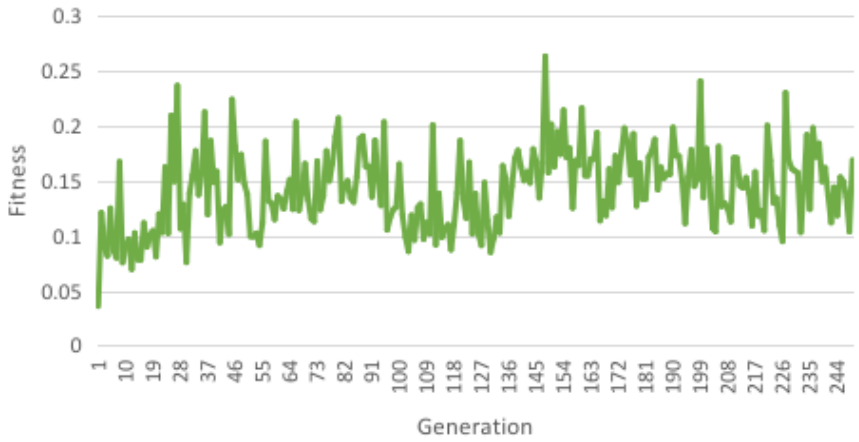


Figure 3.6: The fitness values of the genetic algorithm are too noisy for learning to occur.

The noise can be removed entirely by removing the randomisation from the simulations; however this could lead to biases in the final conclusions due to overfitting. Instead, an alternate method of supressing the noise was chosen where the quadcopter is simulated multiple times with the same gains and the average reward of these episodes is chosen. It is less likely for a set of gains to have a misleading reward by chance when the reward is sampled multiple times and averaged.

This however comes at a cost. Because the quadcopter is simulated N times in a row for each generation, the training time increases by a factor of N. For this reason, N should be given a value small enough to not affect the training time considerably while also being large enough to supress most of the noise due to luck. A value of 5 was chosen for N due to time constraints and because it appeared to perform decently well. Figure 3.7 shows the effect of $N = 5$ under the same conditions as Figure 3.6.

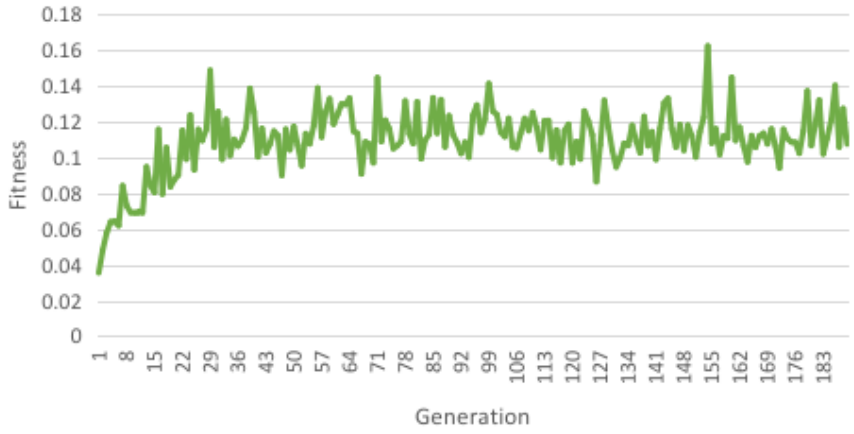


Figure 3.7: The effect of noise is mitigated by repeating the simulations 5 times, and learning can occur.

3.4 Quadcopter Simulation

This section explains how the dynamic and internal effects of the quadcopter are simulated.

3.4.1 Motor Simulation

This dissertation seeks to innovate on other related works by using an accurate motor model instead of directly using the signals of the controllers as the speeds of the rotors. In reality, the relationship between the rotors, the motor speeds and the input voltages are quite complex, so the work presented in this section and in Section 3.4.2 aims to model this complex relation to ensure accuracy in the simulation.

The motor model that was used for this project is based on the mathematical model for DC motors derived in Chapter 12 of Electrical Machines [17]. This model is quite special because it considers the effects of input voltage, current, resistance, inductance, moment of inertia, electromagnetic torque, load torque, friction and back electromagnetic force, and how these relate to the acceleration, velocity and total torque of the motor. This model has several parameters that change between motors; however these parameters are included in the datasheets of most commercial motors.

The block diagram for this model is displayed in Figure 3.8. This model is specifically for motors that use electromagnets instead of permanent magnets because one can notice that there are two inputs to the model, $U_{AB}(s)$ which is the voltage used to make the motor rotate, and $U_f(s)$ which is used to power the electromagnets inside the motor's stator.

Other important variables in the figure are s which is the complex variable of the Laplace transform, R_a which is the internal resistance of the motor, L_a which is the internal inductance of the motor, $I_a(s)$ which is the current of the motor, the product $k_m\Phi_f(s)$ which is a datasheet constant called the torque constant and will be referred to now as K_t, $T_{em}(s)$ which is the electromagnetic torque, $T_m(s)$ which is the load or drag torque, J which is the moment of inertia, k_F which is the coefficient of viscous friction, $\Omega_m(s)$ which is the angular velocity of the motor and the product $k_e\Phi_f(s)$ which is another datasheet constant referred to as the back EMF constant K_e.

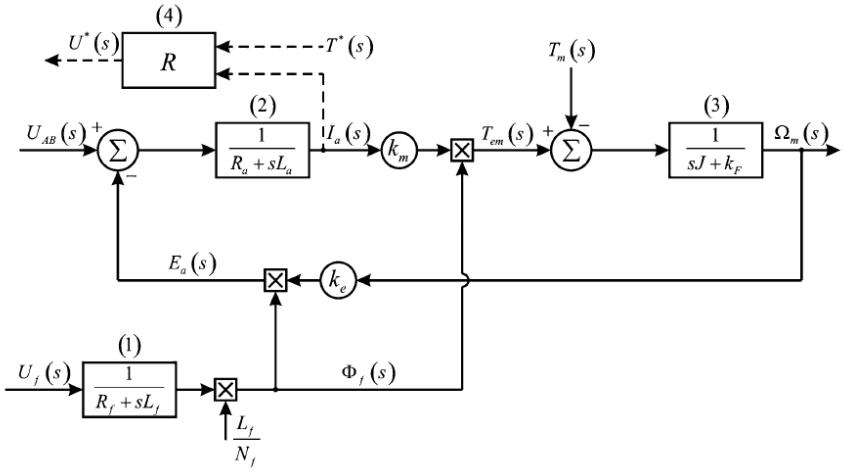


Figure 3.8: Block diagram for the mathematical DC motor model. Diagram from [17].

The first modification the author of this dissertation made was to remove the input voltage $U_f(s)$ related to the electromagnet (this is signalled as block 1 in Figure 3.8), because the rotor is based around the E5000 propeller, which uses an M10 motor that has permanent magnets. This modification does not affect the quality of the output of the model because this effect is usually negligible.

The second modification that was done was to remove the block signalled as block 4 in Figure 3.8 because the book states that it is not part of the mathematical model, but rather it is used for torque control. These modifications lead to the motor model used in the simulation, whose block diagram is observable in Figure 3.9.

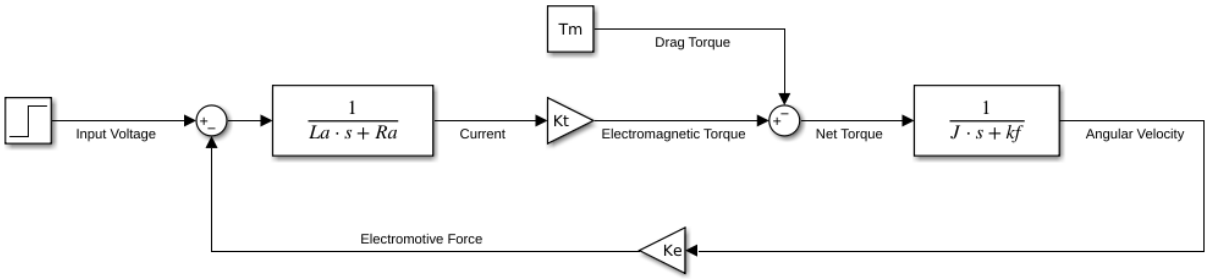


Figure 3.9: Motor model used in the simulation with slight variations from the one in [17].

The E5000 specification [3] specifies the values of L_a, R_a, K_t, J and K_e. The value of T_m, the drag torque of the rotor, is an input which is provided by the rotor model in Section 3.4.2, while the value of k_F is not specified. In other models, k_F is either neglected or given a value around 10 times smaller than J, so for the simulation it was assigned a value of $\frac{J}{10}$. This variable does not need to have a precise value because it has a small effect on the current of the motor.

This block diagram is implemented in the `Rotor.cs` script by using Euler's method, which approximates the state of each of the variables by taking 100 time steps with a simulated duration of 200 microseconds each during every physics update. The outputs of this model are used by the rotor simulation model, which is described in detail in Section 3.4.2. These outputs are the angular velocity and the net torque.

3.4.2 Rotor Simulation

The rotors are simulated by applying two aerodynamic equations that depend on the outputs of the motor simulation. Both these equations come from Chapter 9 - Propellers and propulsion from Aerodynamics for Engineering Students [5], and are used to calculate the lift and drag on

propellers. Equation 3.10 [5] relates the force of thrust or lift T on the rotor to the density of the air ρ, the angular speed n and the diameter of the rotor D.

The value k_T is a proportionality constant called the thrust coefficient which is calculated experimentally in real life. This constant is calculated in the simulation for the E5000 propeller by creating an instance of the rotor for testing, such that the mass of this rotor is equal to the maximum mass specified in the E5000 specifications [3] (14 kilograms), and the voltage it receives as input is also the one which provides the maximum amount of thrust (44.4 volts). The value of k_T is then found by varying it until the lift force is equal to the force of gravity. The value of k_T following this procedure is equal to 0.0026.

$$T = k_T \rho n^2 D^4 \tag{3.10}$$

Equation 3.11 [5] is used to calculate the drag torque Q on the rotor. This equation is similar to Equation 3.10, except that the constant of proportionality k_Q, also known as the torque coefficient, is different than the constant k_T, and the diameter of the rotor D is raised to a power of five instead of four. Unlike the constant k_T, there is no good way of determining k_Q without access to a real quadcopter. This constant was estimated by testing different values and observing the graphs produced by the equations until a behaviour that seemed reasonable was achieved.

$$Q = k_Q \rho n^2 D^5 \tag{3.11}$$

The net torque on the rotor is calculated by feeding the drag torque Q back into the motor simulation, where it replaces the variable T_m from Figures 3.8 and 3.9. This torque is subtracted from the electromagnetic torque of the motor to give the net torque. Once the lift force and net torque are known then these variables are sent to the physics engine in order to move the quadcopter. A block diagram for the combination of the rotor and motor simulations can be seen in Figure 3.10.

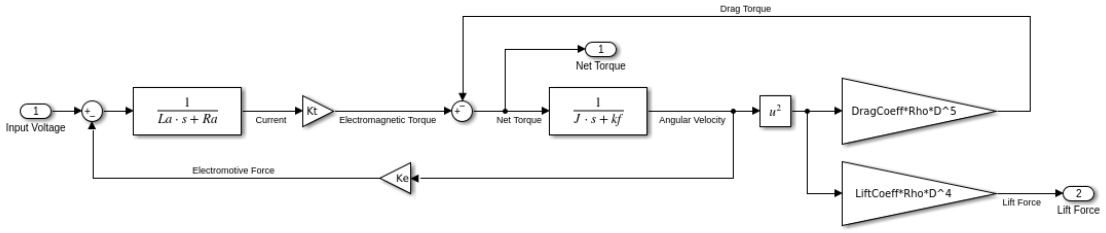


Figure 3.10: Block diagram which combines the motor model and the rotor equations, with lift force and net torque as outputs.

The code that handles the motor and rotor simulations can be found in the `Rotor.cs` script. The main algorithm of the script that follows the block diagram is shown in Algorithm 1, where the variable Δt is the time within the simulation that has passed between each iteration, I is the motor current, V is the input voltage, E is the electromotive force, R is the motor's internal resistance, L is the motor's internal inductance, T_{em} is the electromagnetic torque produced by the motor, K_t is the torque constant, T_{drag} is the drag torque, T_{net} is the net torque, Ω is the angular speed, k_F is the coefficient of viscous friction, J is the moment of inertia, K_e is the back EMF constant, k_T is the thrust coefficient, k_Q is the torque coefficient, ρ is the density of the air and D is the diameter of the rotor. One should note that the constants k_T and K_t are different.

Result: F_{lift} and T_{drag}

while *true* **do**

$I \leftarrow I + (V - E - R * I) * \Delta t / L$

$T_{em} \leftarrow K_t * I$

$T_{net} \leftarrow T_{em} - T_{drag}$

$\Omega \leftarrow \Omega + (T_{net} - k_F * \Omega) * \Delta t / J$

$E \leftarrow K_e * \Omega$

$F_{lift} \leftarrow k_T * \rho * \Omega^2 * D^4$

$T_{drag} \leftarrow k_Q * \rho * \Omega^2 * D^5$

wait Δt seconds

end

Algorithm 1: Combined rotor and motor simulation algorithm.

The large amount of parameters used in this algorithm allows for very different environments to be simulated. For example, one can simulate the quadcopter as if it were on Mars by varying the air density and the force of gravity in Unity's properties. One can also make the air density a function of height to simulate the effects of high-altitude flight.

3.5 Additional Methods

3.5.1 PID Anti-windup

Whenever a PID controller is used in a non-linear system such as with quadcopters, the integral term of the PID controller can increase in magnitude uncontrollably because the error is not corrected fast enough. An example of this is if the quadcopter is placed at a distance of hundreds of metres away from a target; because the quadcopter has a limit on the voltage it can send to update its angle and thrust, it will not be able to move at a rate which stops the integral term from growing. Once it eventually reaches the target, the integral term will be so large that the quadcopter will overshoot it and travel around the same distance as when it started because the integral term will take roughly the same amount of time to decrease.

Anti-windup can be used to remove this effect by placing limits on the integral term. If one were to observe the same example as before with anti-windup, one would notice that the quadcopter adjusts itself after overshooting the target and quickly converges at the target. While this is an extreme example, in practice anti-windup is helpful because its effects are present over shorter distances as well. For this reason, all of the PID controllers implemented in the code use anti-windup to control their integral terms.

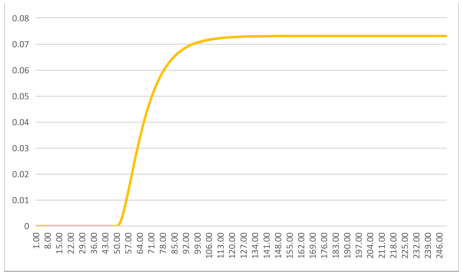
3.5.2 Debugging

In order to fix errors in the code, the scripts were debugged by stepping through the code or by comparing their outputs to other programs, such as Matlab's Simulink, which is capable of accurately simulating block diagrams.

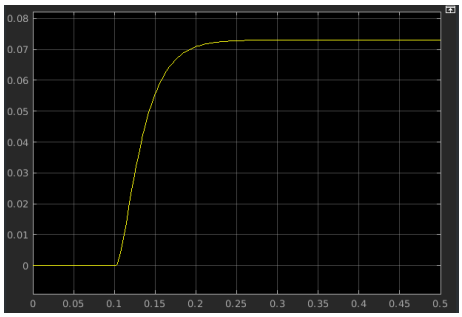
The `Rotor.cs` script, which uses Algorithm 1, is tested by comparing it to the block diagram in Figure 3.10, which is based around equations from [17] and [5]. A unit step response is sent to a copy of the block diagram made in Matlab's Simulink program and to the `Rotor.cs` script, both with the same parameters. The output graphs of the net torque and lift force are then compared to each other. If there are no discrepancies between both these graphs, then the code must not have any errors.

Figure 3.11a is the graph produced by the `Rotor.cs` script for lift force, while Figure 3.11b is the graph produced by Simulink for lift force. Similarly, Figure 3.12a is the graph produced by `Rotor.cs` for net torque and Figure 3.12b is the graph produced by Simulink for net torque.

One can observe that the graphs from the code are practically identical to those from Simulink, meaning that `Rotor.cs` accurately models the block diagram.

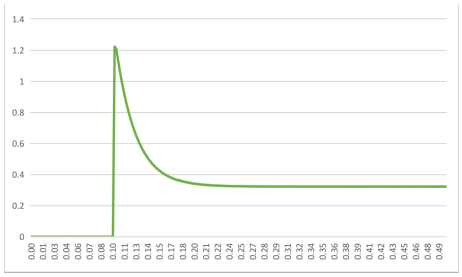


(a) The lift force produced by a unit step response voltage sent to the `Rotor.cs` script.

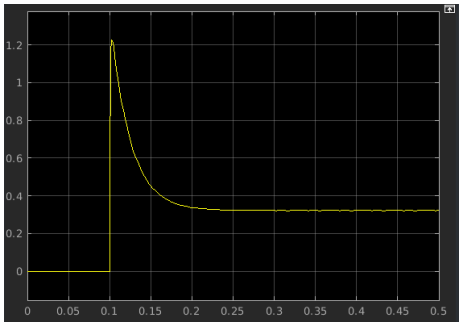


(b) The lift force produced by a unit step response voltage sent to the Simulink block diagram.

Figure 3.11: There are no observable differences between the lift force produced by Simulink's block diagram and the `Rotor.cs` script.



(a) The net torque produced by a unit step response voltage sent to the `Rotor.cs` script.



(b) The net torque produced by a unit step response voltage sent to the Simulink block diagram.

Figure 3.12: There are no observable differences between the net torque produced by Simulink's block diagram and the `Rotor.cs` script.

A similar comparison is done between the PID controller script `PID.cs` and the PID controller provided by Simulink. The block diagram in Figure 3.13 simulates a rotor which hovers at a desired height by using a PID controller.

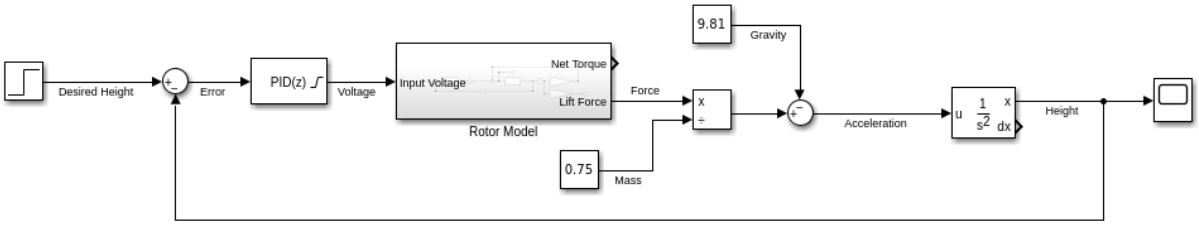
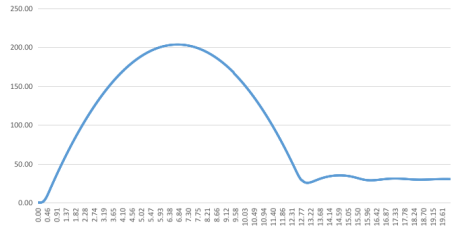
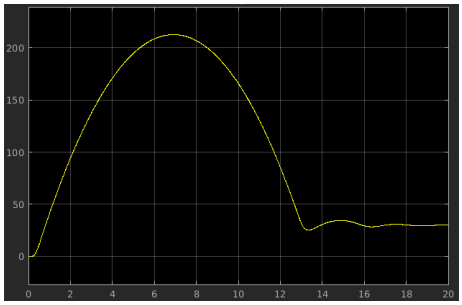


Figure 3.13: Block diagram which is used to test the PID controller from Simulink. The diagram uses the rotor model for added complexity.

This block diagram is recreated in Simulink and Unity, and the height is plotted when Simulink's discrete PID controller with anti-windup is used and when the `PID.cs` class is used. The results can be seen in Figures 3.14a which is the output produced by the `PID.cs` class and 3.14b which is the output produced by Simulink's discrete PID controller.



(a) The output which results from the PID controller from the `PID.cs` script.



(b) The output which results from the PID controller from Simulink.

Figure 3.14: There are no observable differences between the output produced by Simulink's PID controller and the `PID.cs` script.

One can notice that both graphs are practically identical, meaning that the `PID.cs` script represents an accurate PID controller with anti-windup.

Another tool used for debugging code was to use Unity's *Gizmos*. These Gizmos are visual aids which allow the developer to draw vectors, lines and other useful guides on the screen. In order to confirm that the forces are being applied on the correct spots on the quadcopter with the correct magnitudes, a set of lines are drawn to display the force vectors. Another line which is drawn is the angular velocity vector, as this allows one to see what torques are acting on the quadcopter. Figure 3.15 shows these Gizmos on the quadcopter.

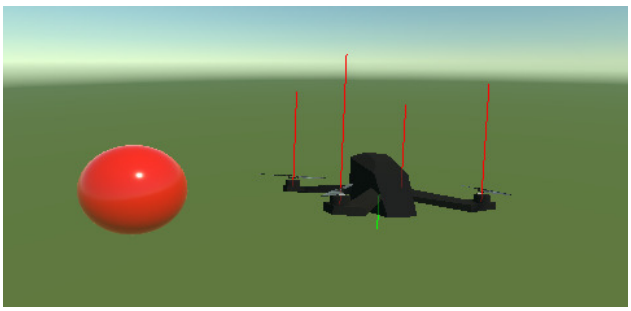


Figure 3.15: Gizmos are used to show the forces and torque vectors on a quadcopter. The forces are shown in red while the torque is shown in green.

For the path following task, the entire path is also drawn by a set of Gizmos that follow the parametric equation of the path. This can be seen in Figure 3.16

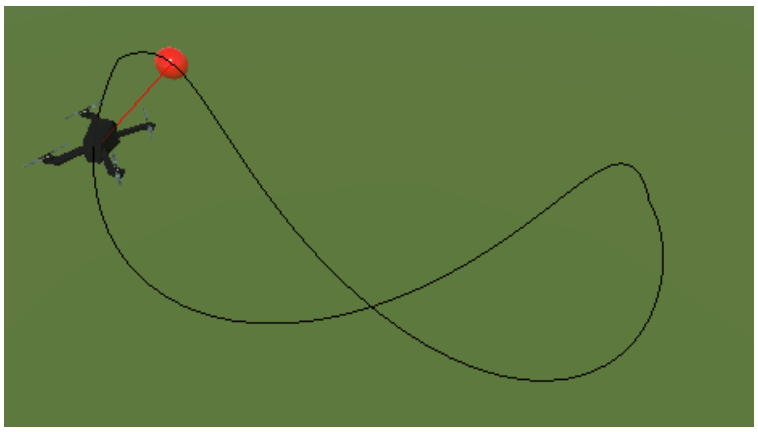


Figure 3.16: Gizmos are also used to help the viewer see the path the quadcopter must follow in black and the distance to the quadcopter in red.

CHAPTER 4. RESULTS

This section presents the results obtained for each of the trained quadcopters. Discussions of these results can be found in Chapter 5, and complementary graphs in appendix A.

4.1 Proximal Policy Optimization

4.1.1 Hovering

The Proximal Policy Optimization agent was trained to perform the task of hovering explained in Section 3.2.1, with the hyperparameters obtained in Section A.1.1. Table 4.1 displays the reward and several important variables such as the final distance, speed, angular speed, yaw error, pitch error and roll error of the quadcopter at the end of training.

Metric	**Value after training**
Steps Trained	150,000,000
Reward	0.4543
Distance	0.9499 m
Speed	0.9631 m/s
Angular Speed	0.1887 rad/s
Roll Error	0.01002 rad
Pitch Error	0.005667 rad
Yaw Error	0.0167 rad

Table 4.1: Metrics for the Proximal Policy Optimization agent for hovering at the end of training. These metrics represent the values obtained at the end of the final episode.

The agent was originally trained for 100 million steps of the simulation, however upon inspection of the results the reward function had a sufficiently large enough slope to justify that more training was needed until converging to the best reward. The agent had to be trained for an additional 50 million steps until convergence occurred.

Figure 4.1 displays the graph of the cumulative reward increasing during the training process, before finally stopping at a reward value of 0.4543 after 150 million steps. As a reminder, the minimum reward of zero is achievable by infinitely bad performance and the maximum reward of one is achievable by perfect movement. This means that the reward is a little bit less than half way between the rewards for these behaviours.

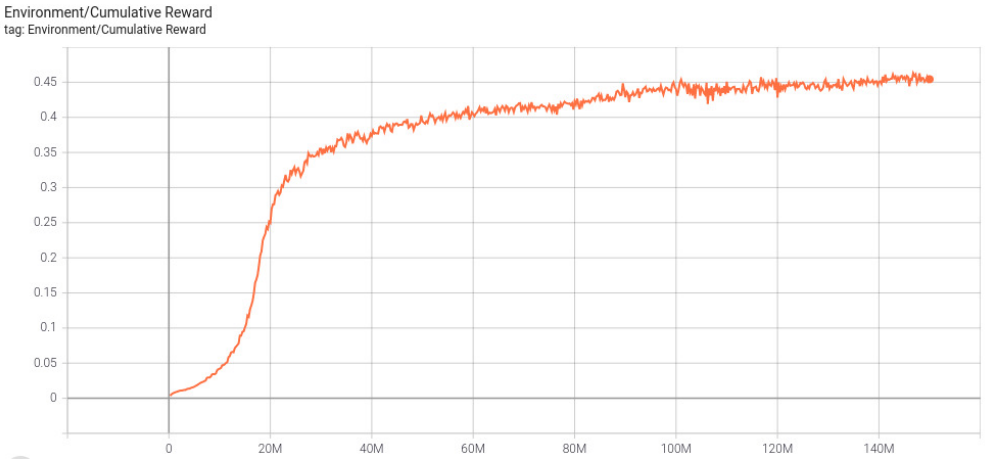
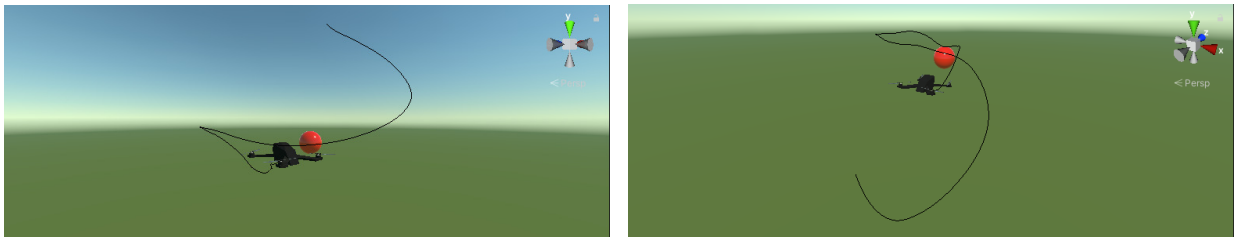


Figure 4.1: The reward obtained by the hovering agent increases over time before settling after around 150 million steps at a final value of 0.4543.

The graphs of the other variables over the training process can be found in Section A.2.1.1.

One should also visualise the behaviour of the quadcopter in the simulation to ensure that it is behaving as expected, because a problem with reinforcement learning agents is that they can either fall in local minima of the reward function or learn to perform an action that maximises reward while not completing the original task. Figures 4.2a and 4.2b both show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.2: Two runs of the simulation are shown. The black lines display the trajectory taken by the quadcopter for hovering over the entire episode, and the red sphere is the target.

One can observe that the quadcopters move somewhat smoothly towards the target but finish a small distance away from the quadcopter when the episode ends.

4.1.2 Path Following

The agent for the path following task was trained in the same way as the agent for hovering, with the hyperparameters obtained in Section A.1.1. Similarly to with the hovering task, the agent was originally trained for 100 million steps of the simulation but had to be trained for another 50 million steps before the reward converged.

Table 4.2 displays several metrics such as reward, final distance, angular speed and angle errors of the quadcopter at the end of training. The final speed has been omitted from this table because it is not optimised for this task and left as a free variable the quadcopter can control.

Metric	Value after training
Steps Trained	150,000,000
Reward	0.5176
Distance	3.843 m
Angular Speed	0.1277 rad/s
Roll Error	0.1721 rad
Pitch Error	0.01272 rad
Yaw Error	0.006802 rad

Table 4.2: Metrics for the Proximal Policy Optimization agent for path following at the end of training. These metrics represent the values obtained at the end of the final episode.

The graphs for these metrics over the training process can be found in Section A.2.1.2. One can notice that despite having a large reward, the distance at the end of training is rather high, suggesting that there is a large error between the quadcopter and the path.

Figure 4.3 displays the graph of the cumulative reward increasing during the training process, before finally stopping at a reward value of 0.5176 after 150 million steps.

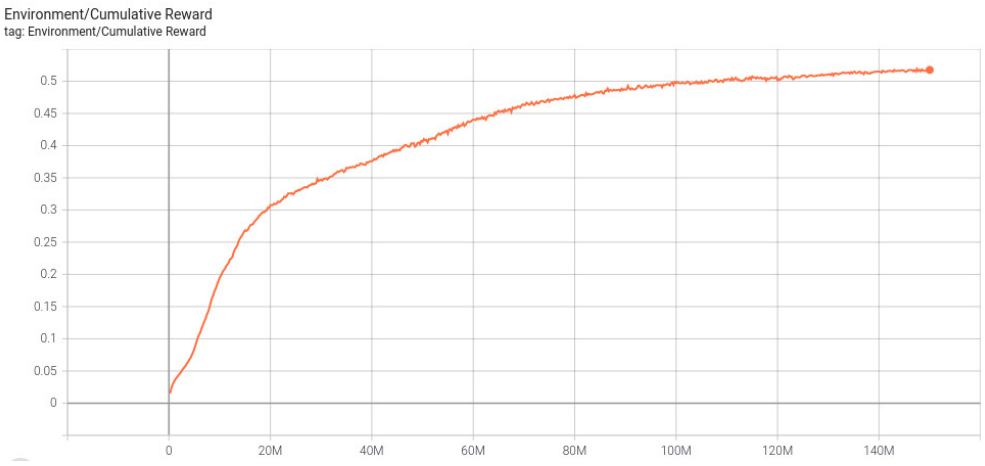
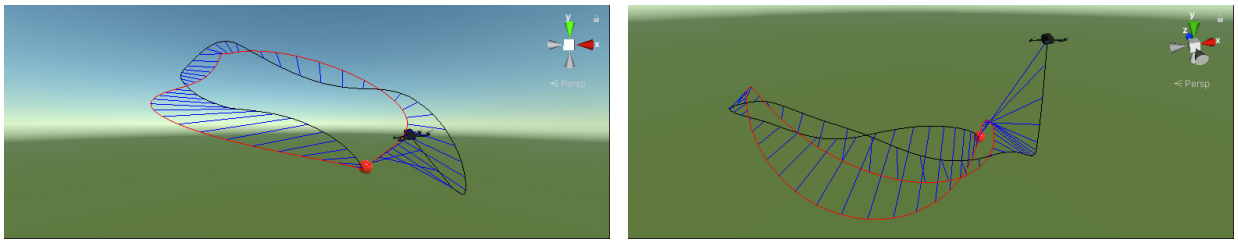


Figure 4.3: The reward obtained by the path following agent increases over time before settling after around 150 million steps at a final value of 0.5176.

In order to better understand the behaviour of the quadcopter, figures 4.4a and 4.4b show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.4: Two runs of the simulation are shown. The black line displays the trajectory taken by the quadcopter, the red line shows the path the quadcopter must follow, and the blue line shows the difference between where the quadcopter is and where the target point on the path is.

One can observe that the quadcopter attempts to follow the path in red, however its trajectory presents a significant amount of error (see the blue lines), and at the end of the simulation in 4.4b the quadcopter diverges from the path.

4.1.3 Landing

The agent for the landing task was trained in the same way as the other two PPO agents. Unlike other tasks, the reward obtained by this agent converged after training for 100 million steps.

Table 4.3 displays the reward and other important metrics at the end of training.

Metric	Value after training
Steps Trained	100,000,000
Reward	0.1341
Distance	7.066 m
Speed	0.01807 m/s
Angular Speed	0.06422 rad/s
Roll Error	0.0000115 rad
Pitch Error	0.0000401 rad
Yaw Error	0.0000943 rad

Table 4.3: Metrics for the Proximal Policy Optimization agent for landing at the end of training. These metrics represent the values obtained at the end of the final episode.

One can observe that the distance at the end of training is very large, while the other metrics are very small. This is not ideal because it indicates that the quadcopter is not landing on the target.

Figure 4.5 displays the graph of the cumulative reward increasing during the training process, before finally stopping at a reward value of 0.1341 after 100 million steps.

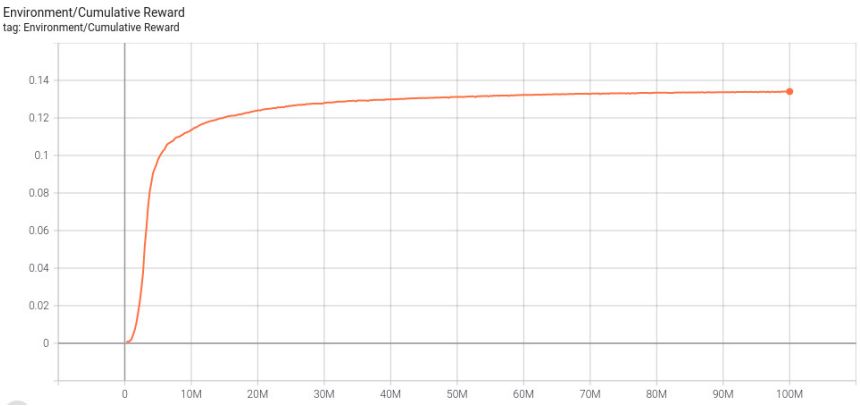
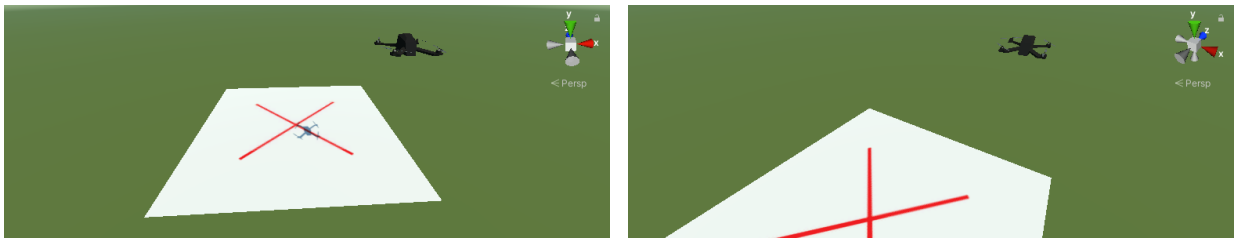


Figure 4.5: The reward obtained by the landing agent increases over time before settling after around 100 million steps at a final value of 0.1341.

To complement this information, figures 4.6a and 4.6b show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.6: Two runs of the simulation are shown, with the trajectory plotted as a black line. The quadcopter must land in the middle of the red cross.

One might notice that there is no black line in either of the simulations. This is because the quadcopter hovers in place during the entirety of the episode, therefore the trajectory line is not visible. This means that the quadcopter has learnt to stabilise itself at the starting point but has not learnt to land at the target.

4.2 PID Controllers

4.2.1 Hovering

The PID controllers for hovering were trained with the hyperparameters for the genetic algorithm in Section A.1.2. The controllers were trained by the genetic algorithm for 250 generations or 125,000 steps, which is 1200 times less than the 150 million steps required to train the PPO hovering agent.

Important metrics such as reward, distance, speed, angular speed and angle errors at the end of training can be seen in Table 4.4.

Metric	Value after training
Steps Trained	125,000
Reward	0.7469
Distance	0.03862 m
Speed	0.002666 m/s
Angular Speed	0.000359 rad/s
Roll Error	0.00000554 rad
Pitch Error	0.0000627 rad
Yaw Error	0.001235 rad

Table 4.4: Metrics for the PID controllers for hovering at the end of training. These metrics represent the values obtained at the end of the final episode.

One can notice that these values are very favourable because the reward is high (roughly three quarters of the way towards being perfect) while the distance, speed, angular speed and angle errors are very small.

Figure 4.7 displays the graph of the reward increasing during the training process, before finally stopping at a reward value of 0.74691 after 125 thousand steps.

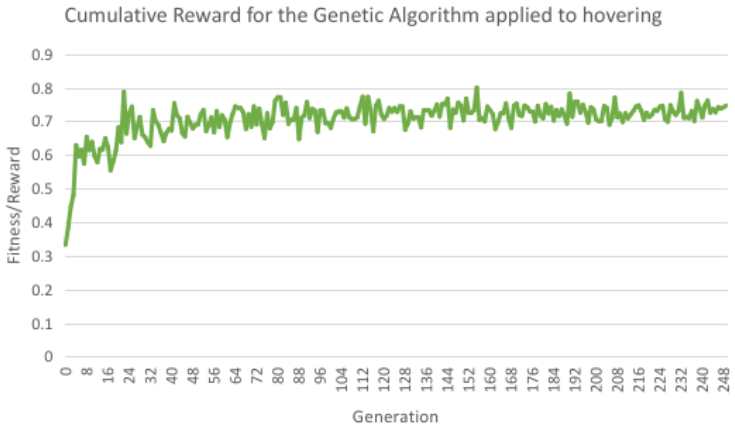
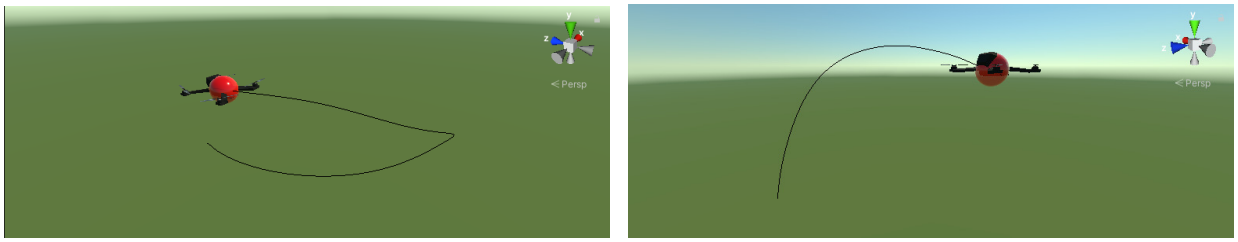


Figure 4.7: The reward obtained by the hovering PID controllers increases over time before settling at a final value of 0.74691.

Similarly to with the reinforcement learning agents, one should visualise the behaviour of the quadcopter in the simulation to ensure that it is behaving as expected. Figures 4.8a and 4.8b both show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.8: Two runs of the simulation are shown. The black lines display the trajectory taken by the quadcopter for hovering over the entire episode, and the red sphere is the target.

One can notice that the quadcopter moves smoothly towards the target and settles at that point before the episode ends.

4.2.2 Path Following

The PID controllers for path following were trained in the same way as the hovering task. Important metrics such as reward, distance, angular speed and angle errors at the end of training can be seen in Table 4.5. The speed has been omitted from this table because it is not optimised for this task and left as a free variable the quadcopter can control.

Metric	Value after training
Steps Trained	125,000
Reward	0.4315
Distance	1.1287 m
Angular Speed	0.04943 rad/s
Roll Error	0.1911 rad
Pitch Error	0.2024 rad
Yaw Error	0.002785 rad

Table 4.5: Metrics for the PID controllers for path following at the end of training. These metrics represent the values obtained at the end of the final episode.

One can notice that the distance at the end of training is relatively low, which suggests that the quadcopter follows the path relatively closely. The other values are quite small which is also favourable.

Figure 4.9 displays the graph of the reward increasing during the training process, before finally stopping at a reward value of 0.43153 after the 125 thousand steps.

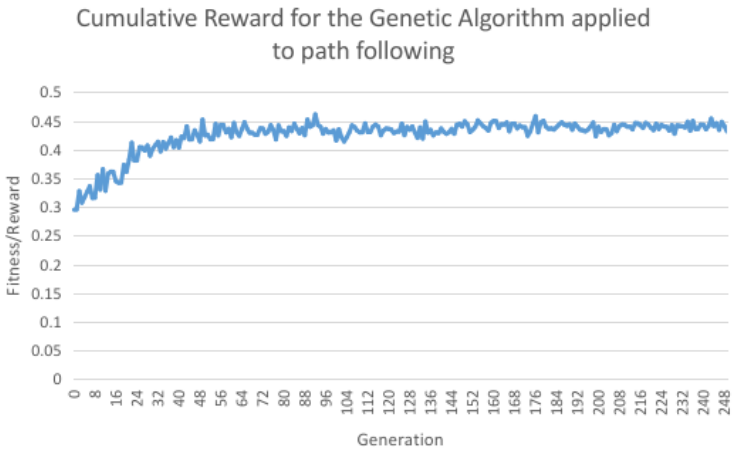
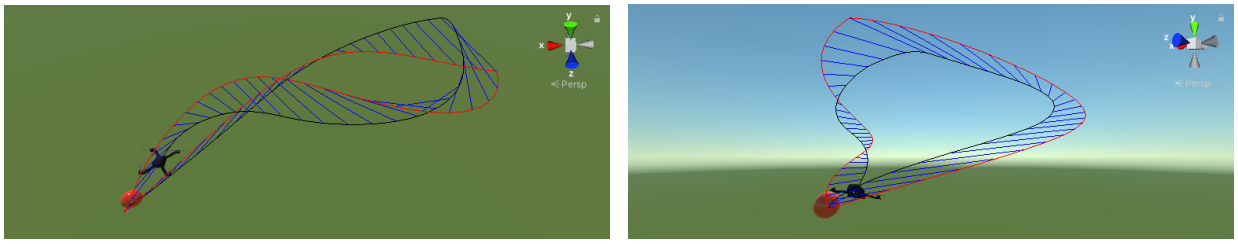


Figure 4.9: The reward obtained by the path following PID controllers increases over time before settling at a final value of 0.43153.

Figures 4.10a and 4.10b both show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.10: Two runs of the simulation are shown. The black line displays the trajectory taken by the quadcopter, the red line shows the path the quadcopter must follow, and the blue line shows the difference between where the quadcopter is and where the target point on the path is.

One can observe that the quadcopter follows the path in a smooth manner but with a noticeable steady state error.

4.2.3 Landing

The PID controllers for landing were trained in the same manner as for the other two tasks. Table 4.6 contains metrics for the reward, distance, speed, angular speed and angle errors for this task.

Metric	Value after training
Steps Trained	125,000
Reward	0.3121
Distance	0.08875 m
Speed	0.5057 m/s
Angular Speed	0.0629 rad/s
Roll Error	0.01142 rad
Pitch Error	0.009618 rad
Yaw Error	0.003819 rad

Table 4.6: Metrics for the PID controllers for landing at the end of training. These metrics represent the values obtained at the end of the final episode.

One can observe that all of the metrics are favourable. Because the distance is quite close to zero and the speed is small, one can consider that the quadcopter has learnt to land successfully.

Figure 4.11 displays the graph of the reward increasing during the training process, before finally stopping at a reward value of 0.31206 after the 125 thousand steps.

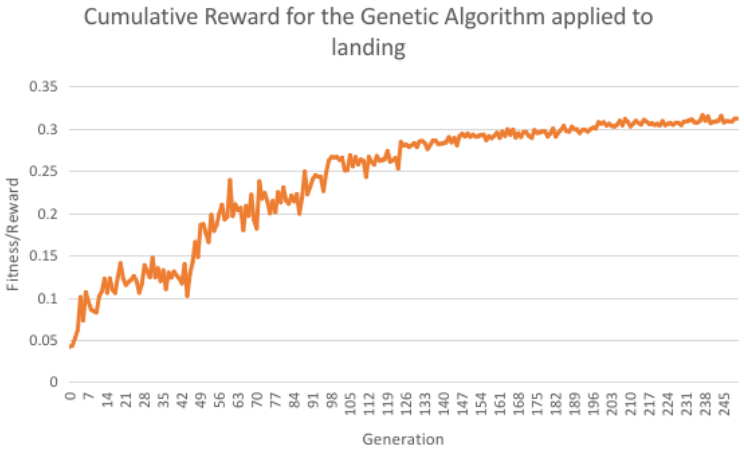
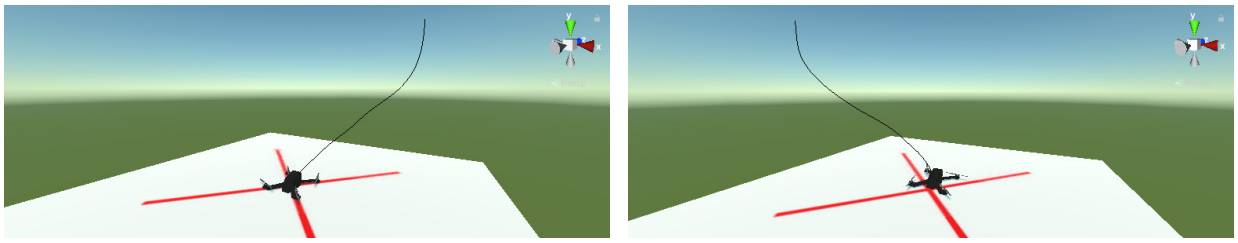


Figure 4.11: The reward obtained by the landing PID controllers increases over time before settling at a final value of 0.31206.

Figures 4.12a and 4.12b both show how the trained agent pilots the quadcopter in the simulation.



(a) First simulation

(b) Second simulation

Figure 4.12: Two runs of the simulation are shown. The black line displays the trajectory taken by the quadcopter.

One can observe that in both simulations the quadcopter does indeed land on the target marked by the red cross. One can also notice that the trajectory taken by the quadcopter is smooth, which is also ideal.

4.3 Comparisons

In order to fully contrast the differences between the Proximal Policy Optimization agents and PID controllers, Table 4.7 compiles all of the metrics obtained in the previous sections of this chapter. The metrics labelled as N/R are not included because they are not relevant.

	Hovering		**Path Following**		**Landing**	
	PPO	**GA+PID**	**PPO**	**GA+PID**	**PPO**	**GA+PID**
Steps Trained	150,000,000	125,000	150,000,000	125,000	100,000,000	125,000
Reward	0.4543	0.7469	0.5176	0.4315	0.1341	0.3121
Distance	0.9499	0.03862	3.8430	1.1287	7.0660	0.08875
Speed	0.9631	0.002666	N/R	N/R	0.01807	0.5057
Angular Speed	0.1887	0.0003590	0.1277	0.04943	0.06422	0.06290
Roll Error	0.01002	0.000005540	0.1721	0.1911	0.00001150	0.01142
Pitch Error	0.005667	0.00006270	0.01272	0.2024	0.00004010	0.009618
Yaw Error	0.01670	0.001235	0.006802	0.002785	0.00009430	0.003819

Table 4.7: Comparisons of the Proximal Policy Optimization (PPO) and the genetic algorithm + PID controller (GA+PID) metrics. The better value between PPO or the GA+PID algorithms is highlighted in blue. All values are in SI units.

One can observe that the GA+PID combination outperforms PPO for hovering in every aspect. For path following, the PPO and GA+PID controllers are comparably good, and for landing, PPO gets a better speed and final errors, but does not perform the task of landing, while the GA+PID combination gets a better reward, distance and angular speed while also landing.

CHAPTER 5. CONCLUSIONS

For the hovering task, the results in Section 4.1.1 show that the Proximal Policy Optimization agent was able to learn how to minimise distance, speed, angular speed and angle error to achieve hovering. One can also notice that the simulations displayed in this section show that the quadcopter adjusts its trajectory to move towards the target, irrespective of its starting position.

The genetic algorithm that trained the PID controllers for the same task obtained significantly better results in all of the aspects analysed compared to the PPO agent; The reward obtained by the genetic algorithm was higher with a value of 0.74691 compared to 0.5176, while the values that needed to be minimised, namely the final distance, speed, angular speed and angle errors were much lower than those of the PPO agent. Furthermore, the genetic algorithm requires less training steps in order to produce a better result than Proximal Policy Optimization.

One can also observe that the simulations of the PID controllers for hovering in Section 4.2.1 show that the PID controllers create a much smoother trajectory that converges exactly at the target, while the PPO agent is more erratic and does not converge exactly on the target.

For the path following task, one can notice that both PPO and the genetic algorithm with PID controllers were able to produce very similar results for most of their metrics.

The Proximal Policy Optimization agent obtained a final reward of 0.5176, while lowering its final distance to a value which is somewhat but not very low. The angular speed and angle errors also presented a somewhat favourable decrease.

The simulations show that the quadcopter moves towards the target but not in a consistent manner. Furthermore, the quadcopter in the second simulation in Section 4.2.1 diverges away from the target during the end of the episode. These results suggest that the PPO agent was able to learn to follow the target to a certain degree but was unable to optimise the way it does so adequately.

The divergence at the end of the second simulation could be an indicator of overfitting, meaning that the quadcopter did not learn to generalise its behaviour but rather to move in a specific manner which is not always adequate for every different variant of the path.

For the path following results in Section 4.2.2, the genetic algorithm was able to achieve a reward of 0.4315 for path following, which is only slightly lower than the reward obtained by the PPO agent. Other variables such as final distance and angular speed were better than the ones obtained by the PPO agent, however the angle errors were worse for the genetic algorithm.

One can notice that the trajectories in both simulations of the quadcopter with PID controllers for path following are smoother and follow the curve in a more consistent manner than the ones from the simulations of the Proximal Policy Optimization despite the fact that they obtained a lower reward.

This discrepancy suggests that the genetic algorithm was able to generalise to a better degree than the Proximal Policy Optimization agent, however the PPO agent obtained a better reward because there were certain points of the trajectory where it got much closer to the target at the expense of being more inconsistent.

Finally, for the landing task the Proximal Policy Optimization agent obtained a reward of only 0.1341. Looking at the simulations one can observe that the quadcopter did not learn to land but to rather hover at the exact place it starts.

The most probable cause for this behaviour is that the PPO agent reached a local minimum of the reward function during training, where it determined that the best policy was to stay still at the starting position. This is supported by the fact that the speed, angular speed and angle errors have to change in order to minimise the distance, meaning that the reinforcement learning algorithm learnt that good rewards could be achieved by optimising these variables at the expense of not optimising distance.

The genetic algorithm on the other hand was able to achieve a better reward value of 0.31206 while minimising the distance and actually landing. The fact that the quadcopter can get a

higher reward by landing instead of by hovering in place reinforces the idea that the PPO agent got stuck in a local minimum.

Furthermore, when trained for a sufficiently long period of time, genetic algorithms can avoid local minima because they always randomise a part of their search space, which is a desirable trait. This is one of the strengths of using a genetic algorithm to train the PID controllers.

The genetic algorithm was successfully able to lower the final distance of the quadcopter to the landing target to 0.08875 metres, which is very close to zero. The final speed of the quadcopter was also reduced to 0.50569 metres per second and the final angular speed was reduced to 0.0629 radians per second. Additionally, the three angle errors also decreased in magnitude over training.

An additional point of discussion is the time taken to train the different quadcopter controlling methods. The amount of training steps required by the genetic algorithm was between 800 to 1200 times smaller than the amount required by the Proximal Policy Optimization algorithm. This means that the genetic algorithm is much more favourable in this aspect.

In total, over 400 million training steps had to be done in order to obtain the results presented in this dissertation, which would take slightly more than 126 years to train in real life. This is a significant advantage of performing training in the simulation as opposed to a real quadcopter.

In summary, the novel combination of using genetic algorithm with PID controllers was able to perform significantly better than the Proximal Policy Optimization agents in most cases, and in the rest of the cases almost as good as the PPO agents.

5.1 Further Work

There are many opportunities for further investigation and improvements on the work presented in this dissertation. One of these opportunities is to test more deep reinforcement learning algorithms such as the OpenAI baselines [13] which include high quality python implementations of the Advantage Actor-Critic (A2C), Actor-Critic with Experience Replay (ACER), Actor Critic using Kronecker-factored Trust Region (ACKTR), Deep Deterministic Policy Gradient (DDPG), Deep Q-Network (DQN), Generative Adversarial Imitation Learning

(GAIL), Hindsight Experience Replay (HER) and Trust Region Policy Optimization (TRPO) algorithms.

Another area of research is how deep reinforcement learning agents and PID controllers respond in environments which have not been tested in this dissertation, where the quadcopter must perform more complex tasks such as avoiding obstacles, moving objects and navigating turbulent atmospheres. These environments can be further complicated by having the quadcopter receive noisy or limited information about its surroundings which must be processed. One example could be to add cameras to the quadcopter and have it apply computer vision and stereographical techniques to infer the nature of its environment.

One could also test the effects that different reward functions have on training quadcopters. Variations in the shapes and priorities of error terms can lead to differences in the quadcopter's performance and training results.

A different approach to finding the optimal hyperparameters for the deep reinforcement learning algorithms can also be taken. If the hyperparameter search is improved this could lead to decreases in training time while increasing the maximum reward the deep learning algorithms achieve.

Finally, PID controllers and deep reinforcement learning algorithms could be used to study how well multiple quadcopters can collaborate or compete with each other to perform tasks. Deep multi-agent reinforcement learning is a relatively new area of reinforcement learning which can be studied more profoundly by extending the work done in this dissertation to this area.

5.2 Limitations and Challenges

There were many limitations on the work that could be done for this dissertation due to external circumstances or the lack of resources. The biggest challenge was that of the SARS-CoV-2/COVID-19 pandemic, which meant that I could not visit the University of Essex in person and make use of the resources on campus. This was also problematic because there was a

time zone difference of six hours between me and my supervisor, which complicated the matter of scheduling meetings.

The fact that I did not have access to physical resources meant that I could not work with a real quadcopter nor use one as a reference to compare the simulations to. This meant that there was no way of quickly verifying if the quadcopter works as intended, which led to the inclusion of several bugs which went undetected for long periods of time.

A big limitation was the amount of time I had to complete this dissertation. Several desirable features could not be included in the final version of dissertation because they would take too long to complete. An example of this is the inclusion of other deep reinforcement learning algorithms aside from Proximal Policy Optimization. These algorithms took a significantly longer amount of time to train than Proximal Policy Optimization (which already required more than a day to train), which meant that they had to be discarded.

Finally, the laptop used to work on the dissertation has 8 gigabytes of RAM, an Intel i5 processor and an AMD Radeon R5 M330 GPU. These specifications are good enough to run ML-Agents, however they are far from optimal for training. Furthermore, ML-Agents only interfaces with CUDA compatible GPUs, meaning that the GPU in the laptop cannot be used to reduce the training time. These problems meant that all instances of training had to be done on the High Performance Computing Facility (Ceres) instead of locally.

REFERENCES

[1] Barnard, A. Gaming system test: New impulses for developers of autonomous systems - Siemens. https://new.siemens.com/global/en/company/stories/research-technologies/digitaltwin/robotics-simulation.html, 2021. [Online; accessed 10-August-2021].

[2] Cano Lopes, G., Ferreira, M., da Silva Simões, A., and Luna Colombini, E. Intelligent Control of a Quadrotor with Proximal Policy Optimization Reinforcement Learning. In *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)* (2018), pp. 503–508.

[3] DJI. E5000 Pro Tuned Propulsion System User Manual. https://dl.djicdn.com/downloads/e5000/E5000_Pro_User_Manual_v1.0.pdf, 2016. [Online; accessed 10-August-2021].

[4] DJI. DJI FPV. https://www.dji.com/dji-fpv, 2021. [Online; accessed 10-August-2021].

[5] Houghton, E. L., Carpenter, P. W., and Elsevier. *Aerodynamics for Engineering Students*, 5 ed. Butterworth-Heinemann, 2003, ch. Propellers and propulsion, p. 527–562.

[6] Hsiao, F.-I., Chiang, C.-M., and Hou, A. Reinforcement learning based quadcopter controller. *AA288/CS238 Past Final Projects* (2019). Stanford University.

[7] Hu, H., and Wang, Q.-l. Proximal policy optimization with an integral compensator for quadrotor control. *Frontiers of Information Technology & Electronic Engineering 21* (05 2020), 777–795.

[8] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., and Lange, D. Unity: A general platform for intelligent agents, 2020.

[9] Koch, W., Mancuso, R., West, R., and Bestavros, A. Reinforcement Learning for UAV Attitude Control, 2018.

[10] ML-Agents. Agents. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md, 2021. [Online; accessed 10-August-2021].

[11] ML-Agents. Training with Proximal Policy Optimization. https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md, 2021. [Online; accessed 10-August-2021].

[12] Navarro, A., and Gibson, S. Making robots more accessible with Forge/OS and Unity - Unity Blog. https://blog.unity.com/manufacturing/making-robots-more-accessible-with-forgeos-and-unity, 2021. [Online; accessed 10-August-2021].

[13] OpenAI. OpenAI Baselines. https://github.com/openai/baselines, 2021. [Online; accessed 18-August-2021].

[14] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

[15] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*, 2 ed. Adaptive Computation and Machine Learning. MIT Press, 2018.

[16] Tiwari, A. Position control of an unmanned aerial vehicle from a mobile ground vehicle. Master's thesis, Michigan Technological University, 2017.

[17] Vukosavic, S. N. *Electrical Machines*, 1 ed. Power Electronics and Power Systems. Springer-Verlag New York, 2013.

APPENDIX A. SUPPLEMENTARY MATERIAL

A.1 Hyperparameters

An important part of obtaining the best results consists of finding the hyperparameters which lead to optimal learning. The following sections explain how hyperparameters were selected for the genetic algorithm that finds the PID controller gains and for the Proximal Policy Optimization algorithm.

A.1.1 Proximal Policy Optimization

There are many hyperparameters which influence the way Proximal Policy Optimization learns. ML-Agents allows the user to change thirteen main hyperparameters. These hyperparameters are listed below with their descriptions from the ML-Agents Training PPO document. [11]

- **Gamma** - "`gamma` corresponds to the discount factor for future rewards. This can be thought of as how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. In cases when rewards are more immediate, it can be smaller. Typical Range: 0.8 - 0.995" [11]

- **Lambda** - "`lambd` corresponds to the lambda parameter used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate (which can be high bias), and high values correspond to relying more on the actual rewards received in the environment (which

can be high variance). The parameter provides a trade-off between the two, and the right value can lead to a more stable training process. Typical Range: 0.9 - 0.95" [11]

- **Buffer Size** - "`buffer_size` corresponds to how many experiences (agent observations, actions and rewards obtained) should be collected before we do any learning or updating of the model. This should be a multiple of `batch_size`. Typically a larger buffer size corresponds to more stable training updates. Typical Range: 2048 - 409600" [11]

- **Batch Size** - "`batch_size` is the number of experiences used for one iteration of a gradient descent update. This should always be a fraction of the `buffer_size`. If you are using a continuous action space, this value should be large (in the order of 1000s). If you are using a discrete action space, this value should be smaller (in order of 10s). Typical Range (Continuous): 512 - 5120. Typical Range (Discrete): 32 - 512" [11]

- **Number of Epochs** - "`num_epoch` is the number of passes through the experience buffer during gradient descent. The larger the `batch_size`, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning. Typical Range: 3 - 10" [11]

- **Learning Rate** - "`learning_rate` corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase. Typical Range: 1e-5 - 1e-3" [11]

- **Time Horizon** - "`time_horizon` corresponds to how many steps of experience to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimate is used to predict the overall expected reward from the agent's current state. As such, this parameter trades off between a less biased, but higher variance estimate (long time horizon) and more biased, but less varied estimate (short time horizon). In cases where there are frequent rewards within an episode, or episodes are prohibitively large, a smaller number can be more ideal. This number should be large

enough to capture all the important behavior within a sequence of an agent's actions. Typical Range: 32 - 2048" [11]

- **Max Steps** - "`max_steps` corresponds to how many steps of the simulation (multiplied by frame-skip) are run during the training process. This value should be increased for more complex problems. Typical Range: 5e5 - 1e7" [11]

- **Beta** - "`beta` corresponds to the strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta. Typical Range: 1e-4 - 1e-2" [11]

- **Epsilon** - "`epsilon` corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process. Typical Range: 0.1 - 0.3" [11]

- **Normalize** - "`normalize` corresponds to whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems." [11]

- **Number of Layers** - "`num_layers` corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation. For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems. Typical range: 1 - 3" [11]

- **Hidden Units** - "`hidden_units` correspond to how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the observation inputs, this should be small. For problems

where the action is a very complex interaction between the observation variables, this should be larger. Typical Range: 32 - 512" [11]

Some of these hyperparameters are easier to pick than others. For example, the `normalize` parameter is set to true because the task of manoeuvring a quadcopter is continuous and complex. The `max_steps` hyperparameter is used to stop the simulation after a certain number of steps have occurred. During testing, the author was able to infer that 10 million steps is enough to compare rewards between different hyperparameters, while around 100 million steps is enough to make the training process stabilise to an optimal solution. Finally, `time_horizon` is set to the maximum number of steps in the episode (500 steps) because better functionality can be obtained if the algorithm learns over the entirety of an episode.

A problem with the other ten parameters is that they have numerical values which are either integers that span large ranges or real values, meaning that there is a near infinite amount of values that can be selected for these hyperparameters. A Proximal Policy Optimization agent has to be trained for every different variation in the hyperparameters in order to compare them, which makes it impossible to test the full range of values. Instead, these values are split into a set of finite values depending on the recommendations in [11].

In order to reduce the amount of time required to test the hyperparameters and conform to the time constraints of this dissertation, these ranges are split into the set of values displayed in Table A.1. An attempt was made to space these values evenly either linearly or logarithmically.

Note that one constraint of the hyperparameters is that `buffer_size` must be a multiple of `batch_size`. Therefore, all combinations of values where `batch_size` is bigger than or equal to `buffer_size` are not considered.

Hyperparameter	**Values**				
Gamma	0.8	0.9	0.995	-	-
Lambda	0.9	0.925	0.95	0.99	-
Batch Size	512	2048	8192	-	-
Buffer Size	2048	8192	32768	131072	524288
Number of Epochs	3	5	7	9	-
Learning Rate	0.00001	0.0001	0.001	-	-
Beta	0.0001	0.001	0.01	-	-
Epsilon	0.1	0.2	0.3	-	-
Number of Layers	1	2	3	4	-
Hidden Units	32	128	512	-	-

Table A.1: Table of different values used for the ten Proximal Policy Optimization hyperparameters.

Even after splitting the values into those in Table A.1, the amount of different combinations of these values for the ten hyperparameters grows exponentially. This is not ideal nor practical because 186624 different environments would have to be tested, which would take over 10 years to do on the Ceres cluster with 12 environments being trained in parallel.

A potential solution could be to modify each hyperparameter individually while leaving the rest the same, however this can lead to problems because there are multiple hyperparameters which are dependent on each other, such as `buffer_size` and `batch_size`. Instead, two hyperparameters are varied at a time depending on the relationship they have between each other in order to reduce the amount of tests that have to be done. This approach requires only 57 environments to be tested, which takes around 30 hours to do on the Ceres cluster.

Following this approach, the number of layers and hidden units were varied first. These two hyperparameters are related to each other because they determine the size or architecture of the neural network. Figure A.1 shows the difference in reward and episode length for the different combinations of values of these parameters in Table A.1.

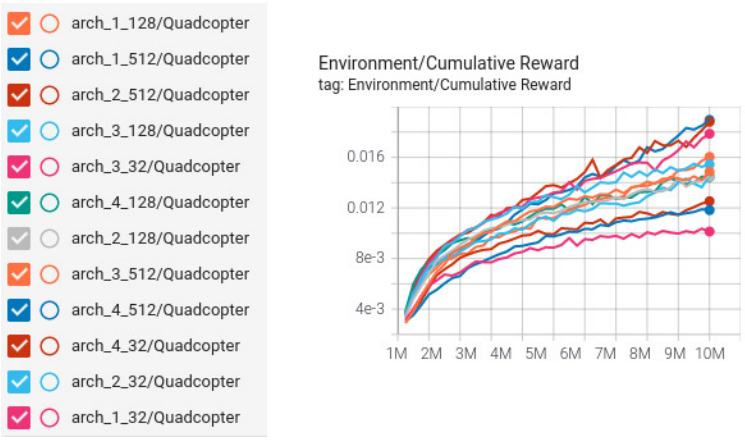


Figure A.1: The cumulative reward obtained over 10 million steps for different values of the `hidden_units` and `num_layers` parameters. Each curve is named arch_a_b, where a is the number of layers and b is the number of hidden neurons. Training is not completed because this is only a preliminary hyperparameter search.

Table A.2 displays the rewards and the amount of time taken to train the 10 million steps for the different combinations of parameters.

Number of Layers	Hidden Units	Final Reward
1	32	0.01013
1	128	0.01485
1	512	0.01182
2	32	0.01543
2	128	0.01451
2	512	0.01253
3	32	0.01786
3	128	0.01451
3	512	0.01603
4	32	0.01880
4	128	0.01445
4	512	0.01895

Table A.2: Table of results for different values of architecture hyperparameters.

One can notice that if the number of layers is one, then the neural network is not a deep neural network. Nevertheless, the results in Table A.2 show that deep neural networks were able to obtain higher rewards than shallow ones at the expense of more training time.

In Table A.2, one can observe that the combination of four hidden layers with 512 hidden units yielded the largest final reward.

The number of epochs and learning rate were varied next. These two hyperparameters are related because they control the rate at which the neural network learns. Figure A.2 shows the difference in reward for the different combinations of values of these parameters in Table A.3.

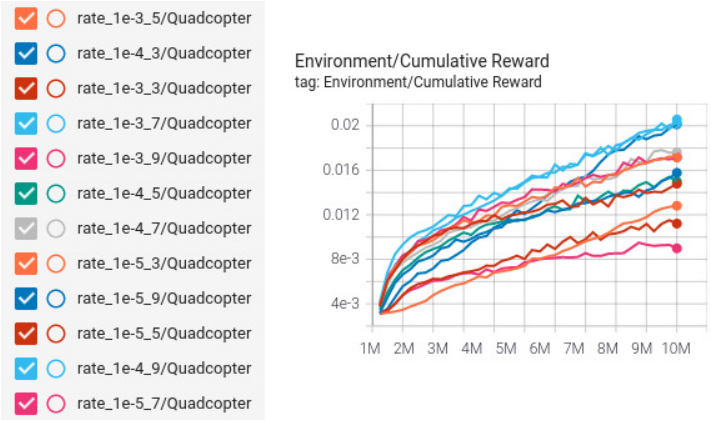


Figure A.2: The cumulative reward obtained over 10 million steps for different values of the `learning_rate` and `num_epochs` parameters. Each curve is named rate_a_b, where a is the learning rate and b is the number of epochs. Training is not completed because this is only a preliminary hyperparameter search.

Number of Epochs	**Learning Rate**	**Final Reward**
3	0.001	0.01479
3	0.0001	0.01577
3	0.00001	0.01281
5	0.001	0.01717
5	0.0001	0.01500
5	0.00001	0.01121
7	0.001	0.02020
7	0.0001	0.01762
7	0.00001	0.00899
9	0.001	0.01717
9	0.0001	0.02057
9	0.00001	0.02013

Table A.3: Table of results for different values of rate hyperparameters.

One can observe that the highest reward is obtained for 7 epochs and a learning rate of 0.001.

Subsequently, combinations of the batch and buffer sizes were tested. These two hyperparameters are related because the buffer size has to be a multiple of the batch size. Figure A.3 shows the difference in reward for the different combinations of values of these parameters in Table A.4.

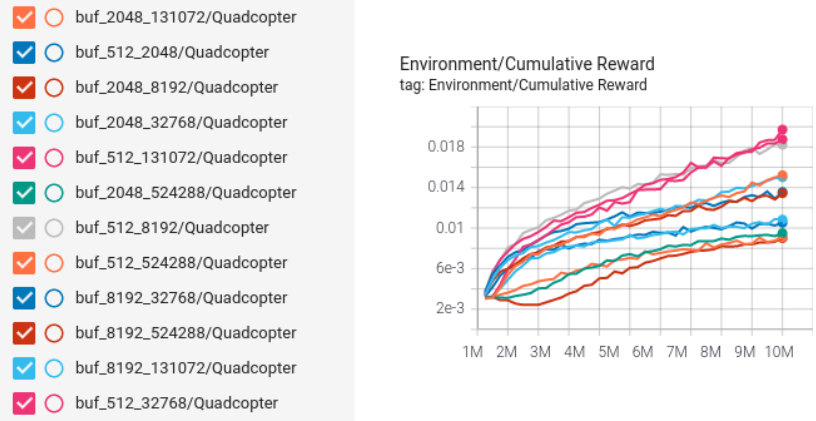


Figure A.3: The cumulative reward obtained over 10 million steps for different values of the `buffer_size` and `batch_size` parameters. Each curve is named buf_a_b, where a is the batch size and b is the buffer size. Training is not completed because this is only a preliminary hyperparameter search.

Batch Size	Buffer Size	Final Reward
512	2048	0.01355
512	8192	0.01822
512	32768	0.01972
512	131072	0.01875
512	524288	0.00908
2048	8192	0.01344
2048	32768	0.01501
2048	131072	0.01522
2048	524288	0.00950
8192	32768	0.01052
8192	131072	0.01084
8192	524288	0.00901

Table A.4: Table of results for different values of batch and buffer size hyperparameters.

In this case the highest reward is obtained when a batch size of 512 and a buffer size of 32768 are used.

Next, the values of beta and epsilon were varied. These two hyperparameters are related because they control the stability of the learning. Figure A.4 shows the difference in reward for the different combinations of values of these parameters in Table A.5.

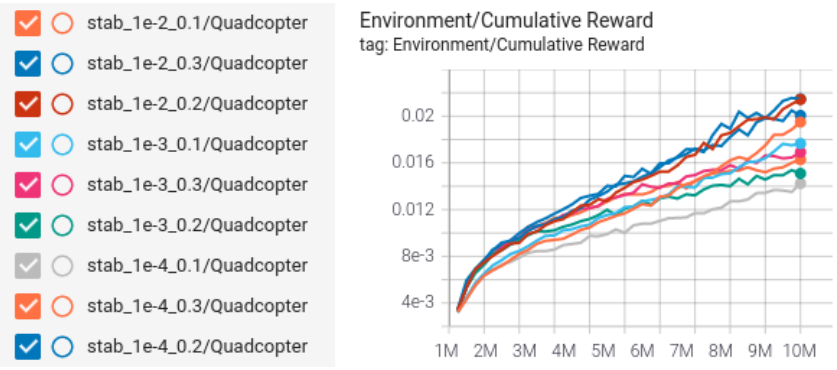


Figure A.4: The cumulative reward obtained over 10 million steps for different values of the `beta` and `epsilon` parameters. Each curve is named stab_a_b, where a is the beta value and b is the epsilon value. Training is not completed because this is only a preliminary hyperparameter search.

Beta	**Epsilon**	**Final Reward**
0.01	0.1	0.01950
0.01	0.2	0.02143
0.01	0.3	0.02147
0.001	0.1	0.01766
0.001	0.2	0.01509
0.001	0.3	0.01691
0.0001	0.1	0.01424
0.0001	0.2	0.02006
0.0001	0.3	0.01628

Table A.5: Table of results for different values of the stability hyperparameters.

One can notice that the highest reward was achieved for a beta of 0.01 and an epsilon of 0.3.

Finally, the values of gamma and lambda were varied. These two hyperparameters are related because they control the way the agent estimates the reward it will obtain by taking an action.

Figure A.5 shows the difference in reward for the different combinations of values of these parameters in Table A.6.

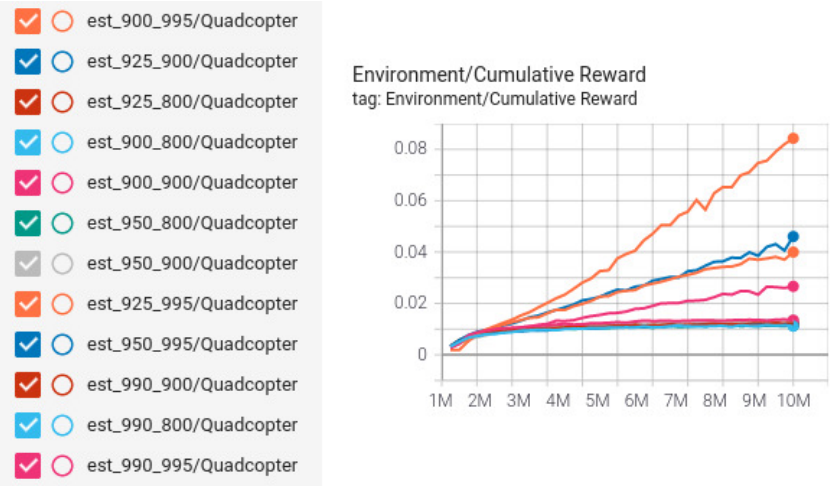


Figure A.5: The cumulative reward obtained over 10 million steps for different values of the `lambda` and `gamma` parameters. Each curve is named est_a_b, where a is the lambda value and b is the gamma value. Training is not completed because this is only a preliminary hyperparameter search.

Lambda	**Gamma**	**Final Reward**
0.9	0.8	0.01114
0.9	0.9	0.01344
0.9	0.995	0.08429
0.925	0.8	0.01176
0.925	0.9	0.01186
0.925	0.995	0.03999
0.95	0.8	0.01136
0.95	0.9	0.01341
0.95	0.995	0.04606
0.99	0.8	0.01153
0.99	0.9	0.01264
0.99	0.995	0.02665

Table A.6: Table of results for different values of the estimating hyperparameters.

One can observe that the rewards obtained for a lambda value of 0.9 and a gamma value of 0.995 are far greater than all the other ones, meaning that these are better hyperparameters.

With these analyses, the final hyperparameters for Proximal Policy Optimization are shown in Table A.7. These values are used in the subsequent sections to train all of the PPO agents.

Hyperparameter	Value
Gamma	0.995
Lambda	0.9
Buffer Size	32768
Batch Size	512
Number of Epochs	7
Learning Rate	0.001
Time Horizon	500
Beta	0.01
Epsilon	0.3
Normalize	Yes
Number of Layers	4
Hidden Units	512

Table A.7: Table of near optimal hyperparameters for training PPO for the quadcopter tasks.

A.1.2 Genetic Algorithm

There are a few hyperparameters for the genetic algorithm, mainly the population size, the proportion of the best chromosomes that survive after each generation, the probability of a mutation occurring, the range of values for the gains of the PID controllers and the amount of generations before training stops. Because the PID gains are tested several times in order to mitigate noise, the amount of times this repetition happens per generation is also a hyperparameter.

These hyperparameters were mostly found by trial and error rather than comparing the rewards obtained for different combinations of values because they tend to impact the training time more than the final reward obtained by the genetic algorithm. Training time does not need to be optimised for the genetic algorithm because the author was able to observe that the genetic

algorithm typically converges before 250 generations have occurred, which takes 125,000 steps or one eight-hundredth of the time PPO needs to converge.

The only hyperparameter changed between tasks is the range of the PID gains. The minimum is zero because negative gains will make the error of a PID controller increase over time, so the maximum gain value is what is changed. For hovering, a maximum value of 1 performed the best, while for path following and landing maximum values of 0.5 led to the best rewards.

For all the tasks, the remaining hyperparameters were given the values seen in Table A.8

Hyperparameter	**Value**
Population size	64
Survival proportion	40%
Probability of a gene mutating	20%
Repetitions to mitigate noise	5
Generations until training stops	250

Table A.8: Table of near optimal hyperparameters for training the genetic algorithm for the quadcopter tasks.

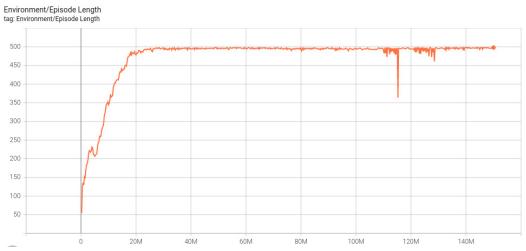
A.2 Result Graphs

This section displays the training graphs obtained for Proximal Policy Optimization and genetic algorithm with PID controllers for each of the tasks. These training graphs can be used to complement the results in Chapter 4.

A.2.1 Proximal Policy Optimization

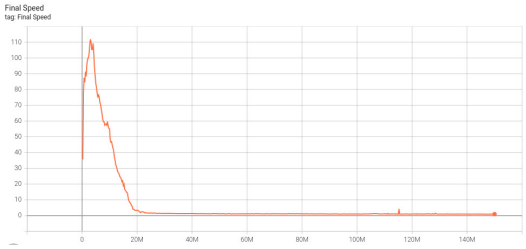
The graphs below were obtained from the Tensorboard utility, which displays useful information about the agents training process.

A.2.1.1 Hovering

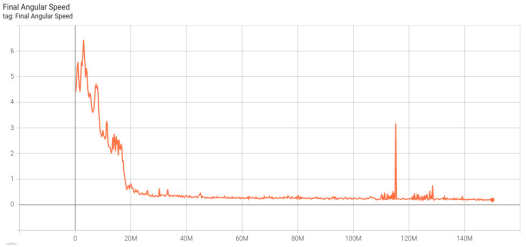


(a) The episode length increased over time until reaching a value of 500, which is equal maximum amount of steps in the episode.

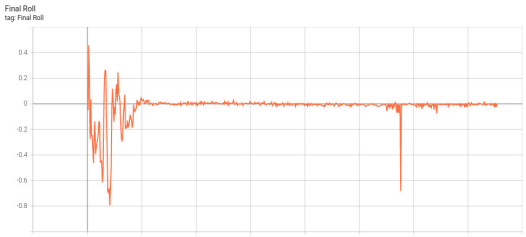
(b) The final distance between the quadcopter and the target decreased during training to 0.9499 metres.



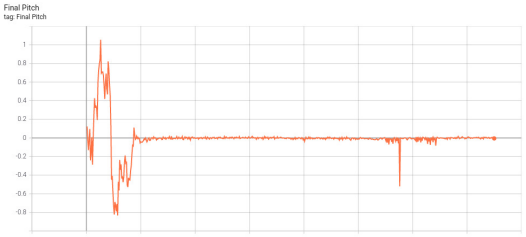
(c) The speed of the quadcopter at the end of each episode decreased to 0.9631 metres per second during training.



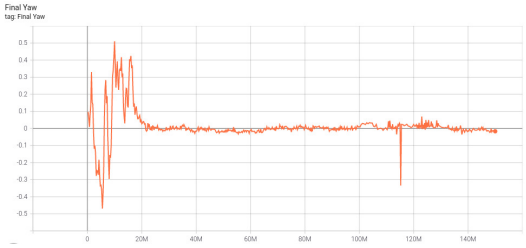
(d) The angular speed of the quadcopter decreased to 0.1887 radians per second over the training process.



(e) The roll error of the quadcopter decreased in magnitude to a value of 0.01002 radians.



(f) The pitch error of the quadcopter decreased in magnitude to a value of 0.005667 radians.



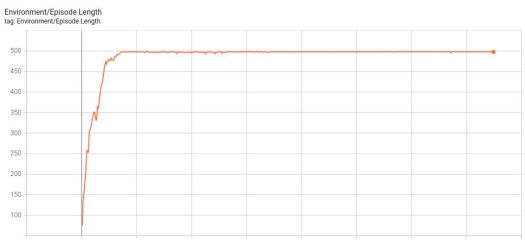
(g) The yaw error of the quadcopter decreased in magnitude to a value of 0.0167 radians.

Figure A.6: Training graphs for PPO hovering.

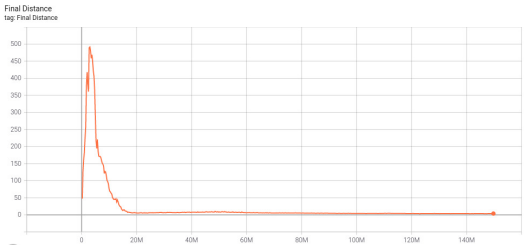
One can observe a few spikes in Figures A.6a through A.6g around 100 million steps. These spikes occur because the graphs display an average value of the variable being plotted at that time step, so because the training was resumed after having previously finished, the new average after resuming is filled with zeroes instead of the previously plotted data.

This is an error with the TensorBoard utility which was used to create the plots of these values. One can observe that a few steps after the point where training was resumed the graphs have the same values as before the spikes, which shows that the training process is not affected by this visual error, and that the spikes can be safely ignored.

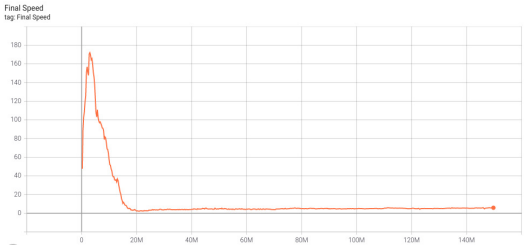
A.2.1.2 Path Following



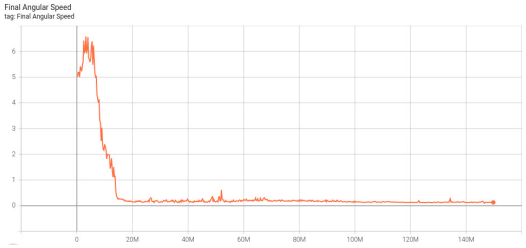
(a) The episode length increased over time until reaching a value of 500, which is equal maximum amount of steps in the episode.



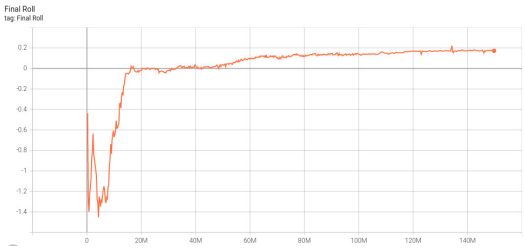
(b) The final distance between the quadcopter and the target decreased during training to 3.843 metres.



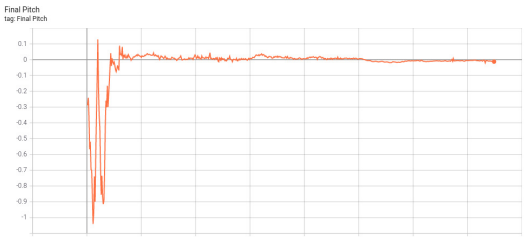
(c) The speed of the quadcopter at the end of each episode settled at 5.874 metres per second during training.



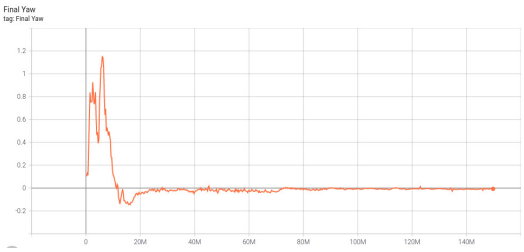
(d) The angular speed of the quadcopter decreased to 0.1277 radians per second over the training process.



(e) The roll error of the quadcopter settled in magnitude to a value of 0.1721 radians.



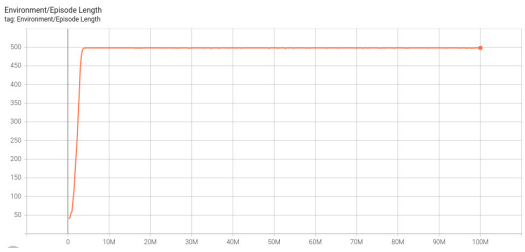
(f) The pitch error of the quadcopter settled in magnitude to a value of 0.01272 radians.



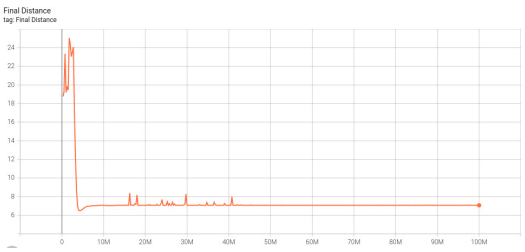
(g) The yaw error of the quadcopter decreased in magnitude to a value of 0.0068018 radians.

Figure A.7: Training graphs for PPO path following.

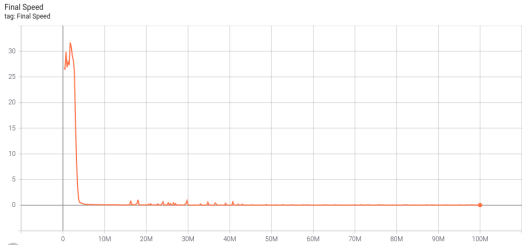
A.2.1.3 Landing



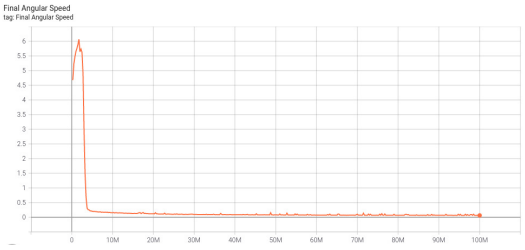
(a) The episode length increased over time until reaching a value of 500, which is equal maximum amount of steps in the episode.



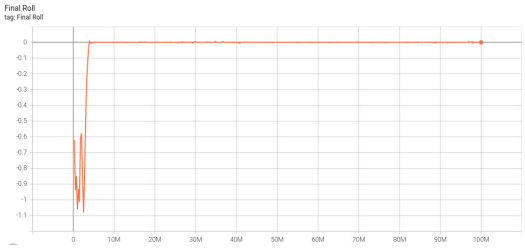
(b) The final distance between the quadcopter and the target settled during training to 7.066 metres.



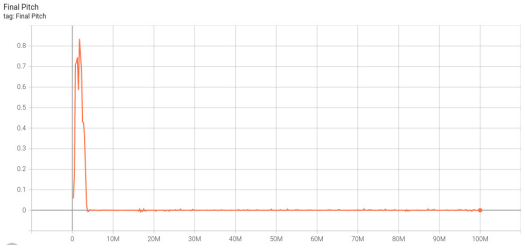
(c) The speed of the quadcopter at the end of each episode settled at 0.01807 metres per second during training.



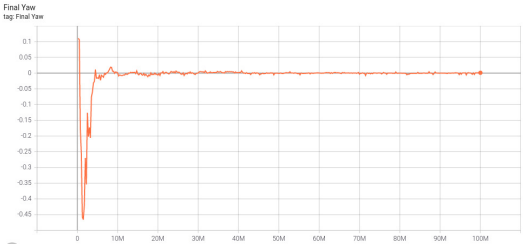
(d) The angular speed of the quadcopter decreased to 0.06422 radians per second over the training process.



(e) The roll error of the quadcopter decreased in magnitude to a value of 0.0000115 radians.



(f) The pitch error of the quadcopter decreased in magnitude to a value of 0.0000401 radians.

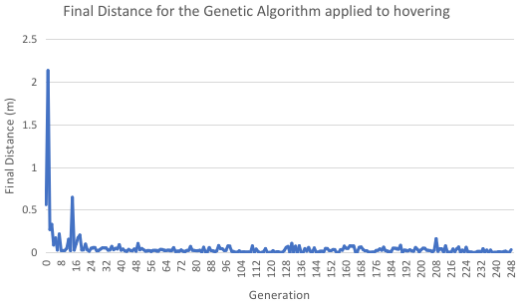


(g) The yaw error of the quadcopter decreased in magnitude to a value of 0.000943 radians.

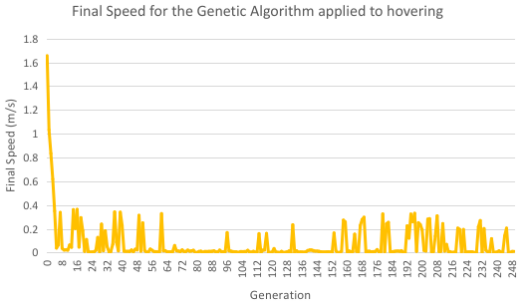
Figure A.8: Training graphs for PPO landing.

A.2.2 Genetic Algorithm

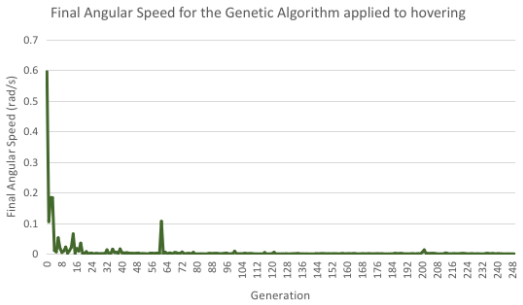
A.2.2.1 Hovering



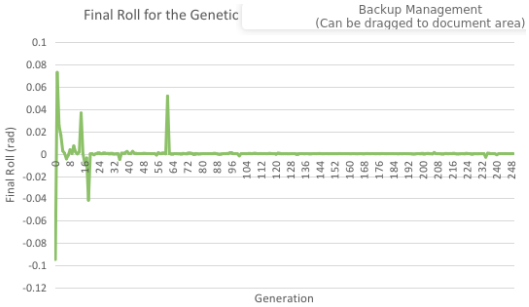
(a) The final distance between the quadcopter and the target decreased during training to 0.03862 metres.



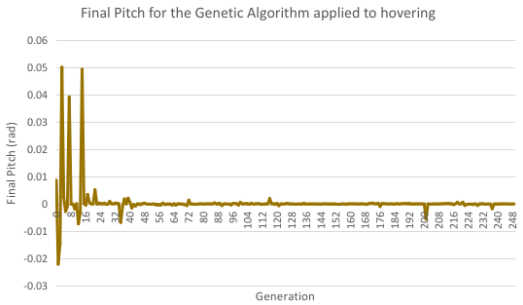
(b) The speed of the quadcopter at the end of each episode decreased to 0.002666 metres per second during training.



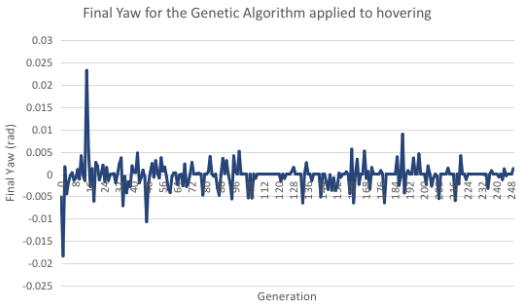
(c) The angular speed of the quadcopter decreased to 0.000359 radians per second over the training process.



(d) The roll error of the quadcopter decreased in magnitude to a value of 0.00000554 radians.



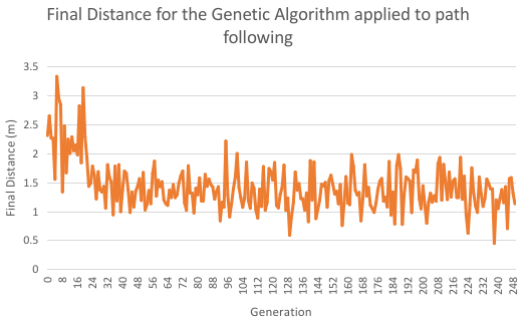
(e) The pitch error of the quadcopter decreased in magnitude to a value of 0.0000627 radians.



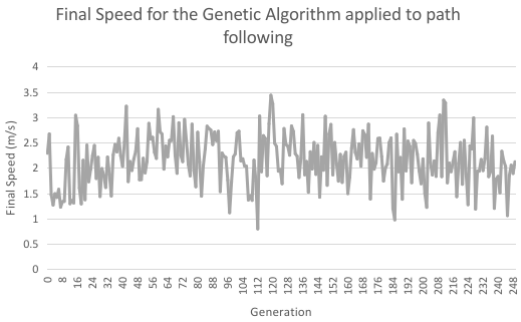
(f) The yaw error of the quadcopter decreased in magnitude to a value of 0.001235 radians.

Figure A.9: Training graphs for GA+PID hovering.

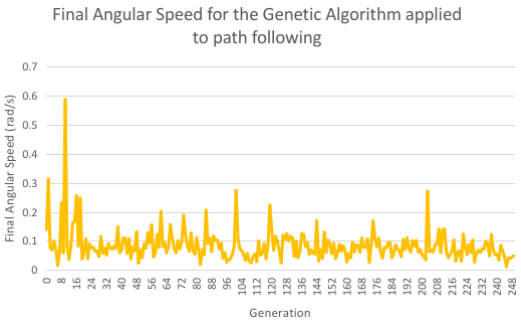
A.2.2.2 Path Following



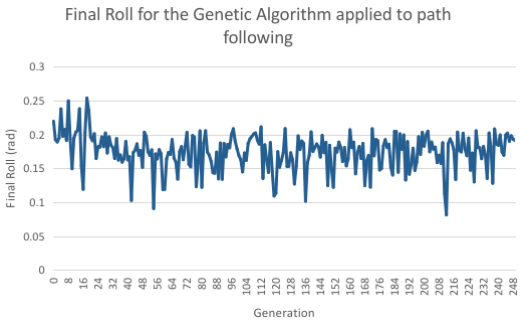
(a) The final distance between the quadcopter and the target decreased during training to 1.128716 metres.



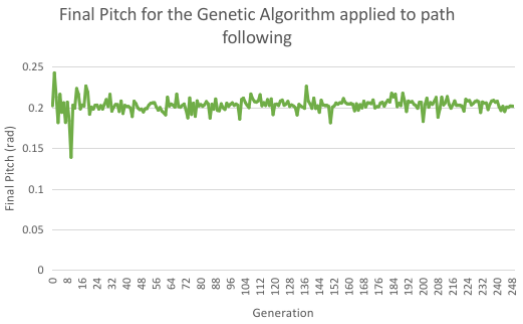
(b) The speed of the quadcopter at the end of each episode settles near 2.1219 metres per second during training.



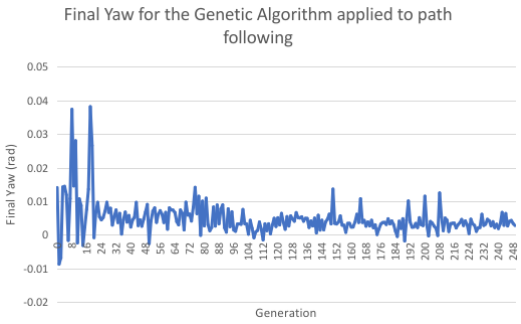
(c) The angular speed of the quadcopter settles to 0.04943 radians per second over the training process.



(d) The roll error of the quadcopter settles in magnitude to a value of 0.19113 radians.



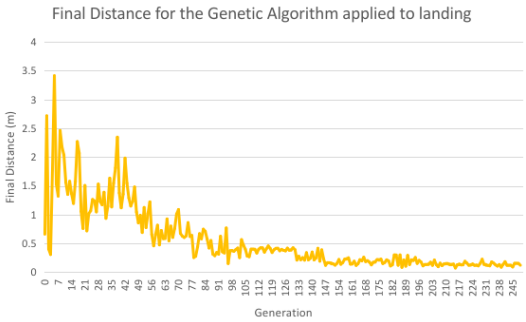
(e) The pitch error of the quadcopter settles in magnitude to a value of 0.20244 radians.



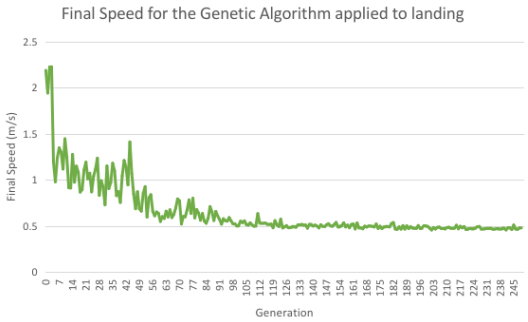
(f) The yaw error of the quadcopter settles in magnitude to a value of 0.002785 radians.

Figure A.10: Training graphs for GA+PID path following.

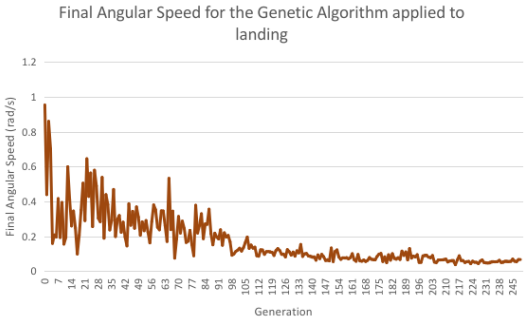
A.2.2.3 Landing



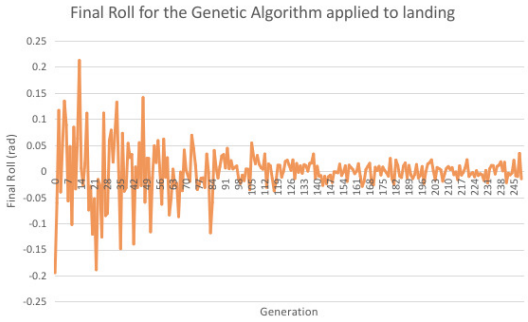
(a) The final distance between the quadcopter and the target decreased during training to 0.08875 metres.



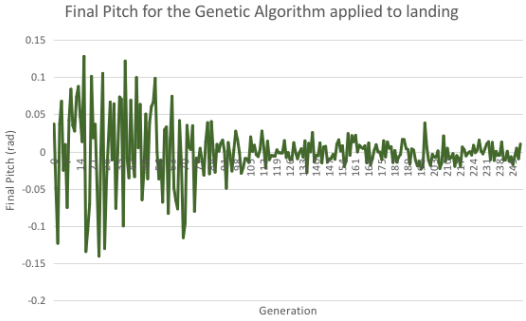
(b) The speed of the quadcopter at the end of each episode decreased to 0.50569 metres per second during training.



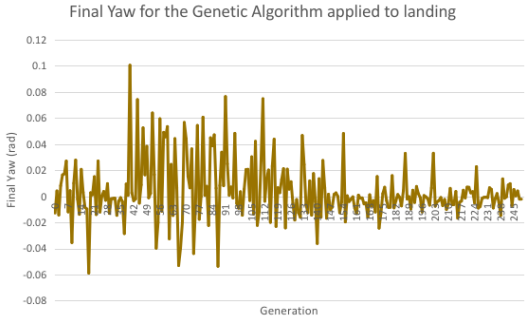
(c) The angular speed of the quadcopter decreased to 0.0629 radians per second over the training process.



(d) The roll error of the quadcopter decreased in magnitude to a value of 0.01142 radians.



(e) The pitch error of the quadcopter decreased in magnitude to a value of 0.009618 radians.



(f) The yaw error of the quadcopter decreased in magnitude to a value of 0.003819 radians.

Figure A.11: Training graphs for GA+PID landing.