NTNU
Kunnskap for en bedre verden

Robustness in Deep Reinforcement Learning for quadrotor control

Martin Aalby Svalesen

Autumn 2020

PROJECT THESIS
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor: Kostas Alexis

Preface

This project thesis presents a literature study on continuous state deep reinforcement learning algorithms, alongside the results when one such algorithm is implemented on a quadrotor helicopter in a simulator. It complements the work done by Eilef Olsen Osvik in his project thesis "Reward shaping in quadcopter control using Deep Deterministic Policy Gradients" as we will use the same reinforcement learning agent as baseline. We are both working towards the same core goal, namely to develop an understanding of how to navigate a quadcopter in geometrically confined environments without the aid of Global Navigation systems using deep reinforcement learning. Our master theses will explore how to integrate machine vision and LiDAR scanners to gain knowledge about the environment, so that an agent can efficiently navigate and explore said environment. Therefore, learning how to control the quadrotor on its own is a crucial step towards achieving this goal, and is what we will be exploring in this project thesis.

The core research on this topic is led by Prof. Kostas Alexis and the team at Autonomous Robots Lab [3] in Nevada, US, and it originates from the DARPA Subterranean Challenge [7].

Trondheim, 2020-12-17

Martin Aalby Svalesen

Executive Summary

In this project thesis, the necessary theoretical background about Deep Reinforcement learning algorithms is developed. Then the necessary mathematics to understand the reinforcement learning algorithm of choice, Deep Deterministic Policy gradients, is derived starting at the basic formulation for policy gradient methods. All necessary building blocks needed for the training of policy and value networks are introduced and explained. The reinforcement learning problem is then formulated, using the insights from the introduction of the dynamic system of a quadrotor and a reward function that encourages the agent to learn the optimal behaviour of navigating to a goal.

To test the significance of the depth of the neural networks used for parametrizing the value and policy functions, three different network topologies are trained and tested. In addition, a fourth network with an integral effect is introduced to test if its architecture improves robustness. The main goal was to find a network topology that matched the complexity in the policy features from the training data.

The key result of this study is that the agent with the most shallow network topology is able to generalize and understand in which general direction it is expected to go, but not scale its acceleration vector properly. This results in unstable behaviour which prevents the quadrotor from maintaining the goal position. The deepest network seems to be overparametrized, meaning that it did not generalize properly, and thus contains many sub-solutions and is sensitive to goal points spawned in regions it has not seen before. This is aligned with the fact that such a deep neural network presents an optimization space with multiple local optima.

The mass is then changed by +10% without retraining the networks, to see how robustly they can handle uncertainties in the model parameters. All of the network topologies experience a drop in performance, with the exception of the shallowest network. It seems that making the quadrotor heavier, and thus diminishing the magnitude of the acceleration vector, made this agent more stable. The network topology which proved to be the most accurate in normal conditions, experienced the biggest drop in performance by an order of magnitude. However, the network topology with integral effect experienced a smaller drop in performance, suggesting that having integral effect in the state-space and reward function actually does improve robustness when mass is increased.

Lastly the mass was changed by -10%, which corresponds to decreasing the gravitational pull and increasing the magnitude of the acceleration vector. All networks produced a trajectory diverging in the z-axis, making the quadrotor rise uncontrollably. The increased magnitude of the acceleration vector made all networks overshoot the goal coordinated in the x and y-axes without recovery within the simulation time limit. The network with integral experienced a comparable deterioration in performance as the other networks, suggesting that having integral effect in the state-space and reward function when the dynamic behaviour is overshooting is not as helpful.

Contents

Preface i
Executive Summary ii

1 Introduction **2**
1.1 Motivation 2
1.2 Machine Learning 2
1.2.1 The perceptron 3
1.2.2 Artificial Neural Networks 4
1.2.3 Gradient Based Training 4
1.2.4 Deep Learning 5
1.3 Reinforcement Learning 6
1.3.1 Markov Decision processes 6
1.3.2 Dynamic programming 8
1.3.3 Model-based dynamic programming algorithms 10
1.3.4 Model-free dynamic programming algorithms 11
1.3.5 Problem of Continuous State and Action Space 13
1.4 Deep Reinforcement Learning 14
1.4.1 Policy gradient methods 14
1.4.2 The Deep Deterministic Policy Gradient Algorithm 17
1.4.3 The Trust-Region Policy Optimization Algorithm 20
1.4.4 The Proximal Policy Optimization Algorithm 21
1.5 Quadrotor Dynamics 23
1.6 Problem Formulation 25
1.7 Related work 25
1.8 Experimental setup 27
1.8.1 High-level RL agent 27
1.8.2 Simulation environment 27
1.8.3 Software system overview 27
1.9 Reward function 28

2 Preliminary results and plots **30**
2.1 Deep Deterministic Policy Gradient (DDPG) . . . 30
2.1.1 Network topologies . . . 30
2.1.2 Training . . . 31
2.1.3 Adding an integral effect . . . 34
2.1.4 Results . . . 34
2.2 Testing for robustness . . . 36
2.2.1 Results +10% . . . 37
2.2.2 Results −10% . . . 38
2.3 Discussion . . . 40
2.3.1 Unmodified mass . . . 40
2.3.2 Modified mass +10% . . . 42
2.3.3 Modified mass −10% . . . 43
2.3.4 Advantages and disadvantages of having a learning-based controller . . . 44

3 Conclusions **46**
3.1 Summary . . . 46
3.2 Conclusions . . . 46
3.3 Further Work . . . 47

A Acronyms **48**

B Algorithms, training progress and computer specifications **49**
B.1 Algorithms . . . 49
B.2 Training progress . . . 51
B.3 Computer specifications . . . 53

Chapter 1

Introduction

1.1 Motivation

Autonomous flight of unmanned aerial vehicles (UAVs) in confined environments is an active field of research. The motivation for having robust, agile and responsive autonomous UAVs navigating confined environments is that it can have a big impact in problems where safety or accessibility is a concern. Examples of this use case are search and rescue in dangerous environments, surveillance of structural integrity and other situations where it is either difficult or dangerous to have a human operator.

Recent advances in computer processing power and power efficiency motivate us to try new approaches to UAV control which have been deemed infeasible until now. The control of aerial systems has until now largely been done by traditional control theory approaches like Model Predictive Control or the Proportional-Integral-Derivative controller (PID for short) and other fixed-gain schemes. However, as the field of machine learning is maturing, we need to ask ourselves if there is a learning-based approach to UAV control that may work better. After all, if a Fairyfly as seen in figure 1.1 with its 7400 neurons is able to fly [28], there exists a way to have learning-based controlled flight, which removes the need for solving complicated optimization problems in real time.

Figure 1.1: A fairyfly

1.2 Machine Learning

Machine learning is the field of computer science where one tries to fit statistical models to a set of data. The goal is then for the computer to be able to make good predictions or actions given a sample set of

data, without explicitly programming the computer how to do so. There are three main directions in the field of machine learning; supervised learning, unsupervised learning and reinforcement learning.

Supervised learning

In supervised learning, there is a labeled data set. This means that for every data tuple, there exist a label with the true association on it. When training a machine learning algorithm, this boils down to trying to fit a statistical model to the dataset in such a way that one can do inference on a random sample, and obtain the true answer [11, p.9].

Unsupervised learning

In this subset of machine learning, the goal is to spot trends in the data without explicitly knowing what to look for [11, p.485]. In this case the data is not labeled, in contrast to the supervised setting. Now, the computer is trying to discover trends and connections in the data that otherwise could not be seen by the naked eye. This field is dominated by clustering algorithms that finds clusters of data with similar characteristics, and classifies them accordingly.

Reinforcement Learning

Reinforcement learning, or RL for short, is the field of machine learning we will be looking at in closer detail throughout this report. The big difference here is that the data we are training our statistical models on, have to be generated by the agent itself, and that the training labels are continually updated. This means that both the distribution of training data and the distribution of labels for training are changing simultaneously, which is a much harder problem to solve in comparison to the supervised learning setting, where the distribution of training data and labels were fixed. Therefore, it is much harder to reach convergence in a reinforcement learning problem, and much care has to be taken to not end up in local maxima.

1.2.1 The perceptron

To begin the explanation of artificial neural networks, it is worthwhile to take a look at the perceptron [25]. This is a binary classifier that, given the input data, will either activate an output y or deactivate it. It works by doing a weighted sum of the input data and adding a constant bias, and then inputting everything into an activation function ϕ as shown in figure 1.2. The output y can thus be expressed as:

$$y = \phi(x_1\omega_1 + x_2\omega_2 + ... + x_m\omega_m + b) \tag{1.1}$$

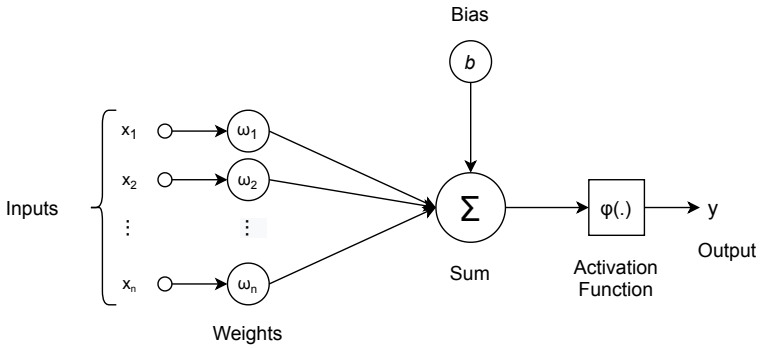


Figure 1.2: Illustration of the perceptron.

In the case of the perceptron, this activation function is just a step function which is either true or false.

1.2.2 Artificial Neural Networks

If we chain several perceptrons together into a fully-connected network and change its activation function, we now have an artificial neural network (ANN)[34]. Each node in this network is now called a neuron. The activation function itself can be changed according to the task at hand (examples are ReLU, leaky ReLU, Sigmoid, Tanh etc.)[31]. This type of classifier is able to learn more complicated classification problems than binary classification, as it easily can handle non-linearities in the input data. Depending on its size, it is also very good at finding features in the data without explicitly being told what they are, and thus can do robust inference based on these features.

1.2.3 Gradient Based Training

The training of an ANN is all about adjusting its weights and biases. For each neuron we have a set of weights for each input, and a bias. The goal now is to be able to accurately predict an output, given an input. The training algorithm used for ANNs is called Stochastic Gradient Descent. Stochastic because we are randomly sampling a batch of data to find a gradient to a loss function [10, p.177]. The loss function can be defined as following:

$$L(\vec{\omega}, \vec{b}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{1.2}$$

where $\hat{y}_i$ is the estimated output given by the network, and y_i is the true label of the input data. Notice that the number of outputs on this network is decided by the designer, and may as well be just one. In that case we have that $n = 1$ and the summation in equation 1.2 disappears. These outputs may be doing either some form of regression, for example predicting the price of a certain item, or classification, for example predicting what type of flower the input data suggests. Either way, the network learns the same way. We see that this loss function is small whenever the predicted $\hat{y}_i$ is close or equal to the real y_i. Therefore we want to do a gradient descent on this loss function, meaning we want to find the gradient of the function at each point, and do a step in the opposite direction to minimize the loss function.

An important concept in the training of neural network is *overfitting* [10, p.110]. This is when a neural network is so perfectly optimized for a training dataset, that it has basically begun to learn it by heart, instead of actually generalizing and seeing the bigger trends. The symptom of overfitting is when the *validation loss* starts to climb, instead of decrease like we want it to. The validation loss is a measure of the performance of a neural network on a piece of the data set it has not been trained for, which is a better metric of its true performance than the *training loss* which indicates the performance on its training data set.

Backpropagation

So how exactly do we minimize this loss function? This is where the concept of *backpropagation* [18] comes in. Let us focus on a single output $\hat{y}_1$ and its corresponding true value y_1. The magnitude of the gradient of this output is proportional to how far $\hat{y}_1$ is from the true value y_1, and the direction of the gradient tells us if $\hat{y}_1$ needs to be increased or decreased. We now know how much we want this output to change and the direction of the change, and we then look at the previous layer. The weights, activations and bias that cause this activation in $\hat{y}_1$ needs to be changed so that $\hat{y}_1$ either increase or decrease depending on the gradient direction. These weights are then changed proportionally to their corresponding activations, and we note by how much we want to change the activation of each previous neuron. Now this process repeats, as we know by how much and in which direction we want the activation of the neuron to change. We thus see a recursive pattern that propagates gradients backwards throughout the networks neurons, weights and biases.

When all these gradients have been propagated throughout the network, we can write the negative gradient as:

$$-\eta \nabla L(\vec{\omega}, \vec{b}) = [\nabla\omega_1, \nabla\omega_2, ..., \nabla\omega_m, \nabla b_1, \nabla b_2, ..., \nabla b_t] \tag{1.3}$$

where η is a proportionality constant. We now have found the gradient of the function 1.2 and can do a gradient descent step by changing the weights and biases in the direction of their negative gradient.

A challenge with deep networks, is that the magnitude of these gradients gets smaller and smaller the more layers the network has. This means that the first layers will learn slower than the last layers. This is termed the *vanishing of gradients* [10, p.289], and is the reason big neural networks require longer training times.

1.2.4 Deep Learning

When the width of each layer and number of layers increases, it is common to use the term "deep learning". This term denotes the network's ability to learn complex features and trends that shallower networks are unable to capture, and has led to many important advances in the field of computer science like computer vision [16].

It is easy to assume that the deeper the network, the more observant it is, and thus the better. However, this is not always the case. Apart from the problem of vanishing gradients, as explained in section 1.2.3, and the fact that larger networks require much more computation, there is also a trade-off between

generalization and overparametrization. The best size of a neural network is the smallest one where it can still accurately detect and learn the level of complexity in the features in the data set. As deep ANNs are considered black boxes, meaning we are not able to understand with any confidence what happens inside it, choosing the size of a network is more an art than a science.

1.3 Reinforcement Learning

Reinforcement learning (RL) is the study of how to teach an agent an optimal behaviour in a specific environment, where the only feedback consists of a scalar reward signal [33, p.1]. Through experience and repetition, the goal of the agent is to maximize this reward signal in the long run. The distinction between the agent and the environment may not always be clear cut, but a general rule is that everything the agent cannot control is considered part of the environment.

In this section we will briefly explain the fundamental building blocks of traditional Reinforcement learning algorithms and concepts. It is worth exploring these concepts, as the modern RL algorithms build on the intuition and terminology coming from this field. Historically, this field has been dominated by dynamic programming algorithms.

Terminology in reinforcement learning versus control engineering

Before delving into the field of reinforcement learning, it is worth making the connection between the traditional terminology in control engineering and their corresponding terms in reinforcement learning.

In RL, our goal is to learn a policy for an agent, i.e. given a state, produce an optimal action. Control engineers would use the word Controller instead of an Agent with a policy.

The environment discussed in RL is termed as the controlled system in control engineering.

And lastly, the word Action used by RL to describe what an agent does, is often termed Control Signal by control engineers.

1.3.1 Markov Decision processes

Let us consider a Markov chain. This is a graphical representation describing dynamic behaviour in a system, formalized through a transition matrix. In each node of the markov chain, the probabilities for transitioning into another node or staying in the same node is denoted with a probability. The probability of the different transitions in each node must sum to one.

From this we can define a Markov Decision Process (MDP). This consists in a Markov Chain when actions and rewards are added to the mix.

Reinforcement learning is presented with the difficult setting in which no prior knowledge about the MDP is available. It then has to interact and experiment in this unknown environment (i.e. the MDP) to learn about how to optimize its behaviour. This process is guided by the reward feedback signal. We now have a model-based setting, meaning that the full dynamics of transitions and distributions of rewards are known. This field is characterized by the use of dynamic programming, as we will discuss in a later section.

States

The set of states S in the environment is the finite set$s^1, s^2, ..., s^n$, with the corresponding size of the state space being $|S| = N$. Each state contains a unique description of everything that matters in a state of the given modelled problem. States can either be legal or illegal, with legal states being the ones the agent can explore, like empty space, and illegal states being the ones the agent is not able to explore, like inside a wall.

Actions

The set of actions A in the environment is the finite set $a^1, a^2, ..., a^k$, where the size of the action space is denoted $|A| = K$. The set of actions that can be executed in a given state $s \in S$ is denoted $A(s)$, where $A(s) \subseteq A$. An action can be used by the agent to control the system state. An important side note is that not all actions can necessarily be applied in every state.

The transition function

When applying an action $a \in A$ in a given state $s \in S$, the system makes a *transition* from the state s to a new state s'. This transition is based on a probability distribution over the set of possible transition from the original state. The transition function T is defined as follows: $T : SxAxS - - > [0,1]$. This symbolizes the probability of ending up in state s' after performing an action a in state s, and is denoted $T(s,a,s')$. This transition function must fulfill the following two conditions.

1. The probability of a given transition $T(s,a,s')$ must be in the set $[0,1]$
2. The sum of transition probabilities in each state must sum to one, i.e. $\sum_{s' \in S} T(s,a,s') = 1$

A very important property of markovian dynamics is that the current state s gives sufficient information about the past to make an optimal decision in the current state. In essence, the information of the past history is lumped into the previous state [33, p.49]. This is called the Markov Property, and this assumption is a fundamental building block in all methods discussed in this project thesis.

The reward function

The state reward function is defined as: $R : S \times A \times A \rightarrow \mathbb{R}$, and often denoted $R(s,a,s')$. This reward function is what the agent will use in the learning process to determine what actions to take, and thus it implicitly specifies the goal of the learning. It is up to the designer of the reward function to determine in which way the system, or rather the MDP, should be controlled.

This all wraps up into the definition of a Markov Decision Process (MDP). It is defined as a tuple $\langle S, A, T, R \rangle$ where S is a finite set of states, A is a finite set of actions, T a transition function and reward function R as specified earlier. We say that the transition matrix T and the reward function R make the *model* of the MDP.

Policies

Given a Markov Decision Process $\langle S, A, T, R \rangle$, a policy π is a function that maps states into actions, i.e. $\pi : S \rightarrow A$. The policy interacts with the MDP in the following way:

1. First an initial state s_0 is generated from the initial state distribution
2. An action $a_0 = \pi(s_0)$ is then performed as decided by the policy π
3. Based on our models for T and R, a transition is made to the next state s_1 with probability $T(s_0, a_0, s_1)$, and a reward $R(s_0, a_0, s_1)$
4. The process repeats and produces the tuples $\langle s_0, a_0, r_0 \rangle$, $\langle s_1, a_1, r_1 \rangle$ and so on
5. If we have an episodic setting, the process ends when the system reaches a pre-determined state s_{goal}.

1.3.2 Dynamic programming

The core goal of a reinforcement learning algorithm is to find an optimal policy $\pi(a|s)$ which for a given state s indicates an optimal action a.

Optimality criteria

Every time we run our algorithm for T time steps and receive a reward, we can compute the total reward as the sum of the rewards at each time step:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{1.4}$$

An additional concept we need to introduce is *discounting*. This can be formulated as follows:

$$G_t = R_{t+1} + \gamma^1 R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.5}$$

We can see that with $\gamma = 1$, we are weighing the future rewards the same as the rewards closer in time. However if we set $\gamma = 0$, the sum only contains the first element. This value for γ is called the *discount rate* and is always between zero and 1 ($\gamma \in [0, 1]$). Thus we can interpret this discount rate as how far into the future we are willing to look. With a small γ close to zero, we only see the very first time steps, but when it approaches 1 the sum includes more elements into the summation and the approach is more long-sighted.

However, this is not possible to calculate deterministically as our system is inherently stochastic. We can only compute the *expected* future reward:

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{1.6}$$

Value functions and Bellman equations

The *value* of a state s under policy π, denoted $V^\pi(s)$, is the *expected return* when starting at s and following π thereafter. Using the infinite-horizon discounted reward, it is expressed as:

$$V^\pi(s) = E_\pi\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \Big| s_t = s \Big\} \tag{1.7}$$

The value functions are used to link together optimality criteria and policies, meaning that learning algorithms for MDPs make their optimal policies by learning a value function. The value of being in a specific state represents an estimate of how good it is for an agent to be in said state. This is again expressed through the value function in 1.7, and is valid under a given policy π.

Another very important function is the Q-function. This is defined as the expected return of rewards starting from a given state s, taking an action a and following the policy π thereafter:

$$Q^\pi(s,a) = E_\pi\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \Big| s_t = s, a_t = a \Big\} \tag{1.8}$$

Both value functions and Q-functions are hugely important within the frameworks of RL, and their intuitive understanding is key for more complex deep RL methods we will look at later.

Value functions satisfy certain recursive properties:

$$V^\pi(s) = E_\pi\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \Big| s_t = s \Big\} = E_\pi\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s = s_t\} = \sum_{s'} T(s,\pi(s),s')(R(s,a,s') + \gamma V^\pi(s')) \tag{1.9}$$

This shows us that the value of a given state is defined in terms of the immediate reward and discounted values of possible next states weighted by their transition probabilities.

Thus, an optimal policy $\pi*$ can be written as:

$$V^{\pi\star}(s) \geq V^\pi(s) \;\; \forall \;\; s \in S \tag{1.10}$$

Now we are ready to formulate an important equation, the *Bellman Optimality Equation* [33, p.63]. This states that the optimal solution $V^\star = V^{\pi\star}$ satisfies:

$$V^\star(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma(V^\star(s')) \tag{1.11}$$

This tells us that maximizing the expected sum of rewards with respect to the action a will result in an optimal value function at that state. Note that the optimal value function also appears inside the summation, meaning that to be able to calculate the optimal value function at a state s, $V^\star(s)$, one must also know the optimal value function for the next state $V^\star(s')$.

Since we know that the *optimal action* is the one that maximized the expected sum of rewards, we can change the Bellman Optimality Equation to our own benefit by substituting the max()-operator with the argmax()-operator. This function will then return the action with which the expected sum of rewards is maximized:

$$\pi^\star(s) = \arg\max_a \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma(V^\star(s')) \tag{1.12}$$

This is also called a *greedy policy*, as it tries to maximize the reward at every time step.

In the same way we also have an optimal *state-action* value, or Q-value, as follows:

$$Q^\star(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q^\star(s',a')) \tag{1.13}$$

As we can see, $Q^\star$ and $V^\star$ are closely linked. Their relationship is formalized through the following two equations:

$$Q^\star(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^\star(s')) \tag{1.14}$$

$$V^\star(s) = \max_a Q^\star(s,a) \tag{1.15}$$

Combining equation 1.12 and 1.13 we get the following equation for the *optimal action*

$$\pi^\star(s) = \arg\max_a Q^\star(s,a) \tag{1.16}$$

1.3.3 Model-based dynamic programming algorithms

When we know what the transition matrix and reward function is, we can use model-based solutions. Here we will look at two important ones: *Value iteration* and *Policy iteration*.

Value iteration

This algorithm focuses solely on estimating the correct value function for each state. Value iteration is guaranteed to converge in the limit towards $V^\star$.

To make the value iteration algorithm, we just make the Bellman Optimality Equation 1.11 into an update rule:

$$V_{t+1}(s) = \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_t(s')) = \max_a Q_{t+1}(s,a) \tag{1.17}$$

This is run through all the states until some convergence property is met. Once we have converged on the optimal value function, we can calculate the optimal policy at each state using the equation 1.12.

Policy iteration

This algorithm consists of two steps: *Policy evaluation* and *Policy improvement*.

Policy evaluation

This is considered the prediction problem. It involves finding the value function V^{π} of a policy. This update rule is almost exactly the same as in 1.17, however we remove the $\max_a$ and replace a with the action given the current policy $\pi(s)$

$$V(s) := \sum_{s'} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V(s')) \tag{1.18}$$

This update rule is then run until convergence.

Policy improvement

Now that we know the value function V_{π} of a policy π, we can proceed to improving the policy. We compute the value of all actions in a given state $Q^{\pi}(s,a)$, and compare with the value function $V^{\pi}(s)$. If $Q^{\pi}(s,a) > V^{\pi}(s)$for some action $a \in A$, we know that we can improve our policy by choosing this action instead of the current $\pi(s)$, and the policy is updated.

To do this calculation we look back to the equation 1.12 and turn this into our update rule by replacing the optimal $V^{\star}(s)$ with our own estimate for the value $V(s')$:

$$\pi(s) = \arg\max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V(s')) \tag{1.19}$$

If the policy has not changed during this step, we know we have found our optimal policy. If the policy has changed however, we repeat the whole process from the start with policy evaluation.

We will later see that the idea of approximating a value function to which one can compare a range of actions to determine a policy is essential, as we will see in the section 1.4.1.

1.3.4 Model-free dynamic programming algorithms

Up until now we have looked at methods where we calculate an optimal policy given that a *perfect model is available*. However, when doing Reinforcement learning, one has to assume that such a model is not available. This means that we have to obtain an optimal policy by gathering information about the system ourselves. This adds a focus on approximation and incomplete information, which in extension means that we have to do sampling and exploration. Now we do not have *a priori known* transition and reward models.

In model-free algorithms one has a choice between two types of algorithms:

1. Learning the transition and reward model through interacting with environment. When a model is sufficiently accurate, we can use all the same methods from model-based section. This type of method is called indirect or model-based RL

2. Direct RL is to step right into estimating values for actions without estimating the model of the MDP itself

Temporal Credit assignment

When the agent reaches the goal state, which ones of the previous actions were truly responsible for it? How can we know the value of an action when the reward does not arrive until much later? This is the problem of temporal credit assignment.

To solve this problem, we can use similar mechanisms similar to what we saw in value iteration, i.e. adjusting the value of state based on the immediate reward and the estimated discounted value of the next state. However, this time we cannot allow the transition function T and the reward function R in our update rules. This is called *temporal difference learning*, and uses a concept called *bootstrapping*. When bootstrapping, we are using an estimate to compute another estimate.

The main difference between these methods and previous model-based methods is that we are no longer doing full sweeps through the whole state space. We are discovering paths through the state space ourselves, and finding the best ones through experimentation. This makes these algorithms much less computationally expensive. The other advantage of temporal difference learning is that the policy is developed in an online (meaning that it updates its estimates after each experience) and incremental fashion. This makes it learn continuously without having to wait until the end before updating its estimates.

Exploration vs Exploitation

When working with model-free algorithms, we have to explore the environment by interacting with it. This means that we have to balance between selecting the currently estimated best action, and taking a completely different action.

One such concept is the $\epsilon - greedy$ method. It chooses the currently best action a in a state s with probability $1-\epsilon$ and a completely random action with probability ϵ. The value for ϵ can thus be tuned to balance exploration and exploitation.

TD(0)

The simplest temporal difference method is the TD(0) method. It solves the prediction problem, meaning that it estimates V^{π} for some policy π, in an online and incremental fashion. Its update rule is:

$$V_{k+1}(s) = V_k(s) + \alpha(R + \gamma V(s') - V(s)) \tag{1.20}$$

where $\alpha \in [0,1]$ is the learning rate that determines the rate at which the Value function is updated. This update is done *after experiencing the transition* from state s to s', based on action a, while receiving reward r. While the policy π is not explicitly mentioned in the update rule, it is implicitly evaluated as it is the policy that decides the next step s'. We can see that inside the parenthesis behind α, there is a comparison. If $r + \gamma V_k(s')$ is larger than the current value $V_k(s)$, the new value $V_{k+1}(s)$ is increased. This means that the update rule both uses the experienced reward and the bootstrapped future value to update the value in the current state.

When it comes to finding the optimal policy itself, generally TD methods use some variation of *generalized policy iteration* (GPI), one example of which we saw earlier in section 1.3.3.

Sarsa

Sarsa is an on-policy temporal difference method that is a control method, meaning that is finds an optimal policy. It tries to estimate an action-value function $q_\pi(s,a)$ for the current behaviour policy π for all states s and actions a instead of a state-value function as seen earlier. The update rule is very similar to TD(0):

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha(r_t + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)) \tag{1.21}$$

The actions taken in each time step is taken using a policy derived from Q, for example using an $\epsilon - greedy$ strategy. It has been shown that Sarsa converges with probability 1 to an optimal policy and action-value function given that all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy.

Q-learning

Q-learning in contrast, is an off-policy TD control method, meaning it finds an optimal policy independent of the policy being followed. The big difference from Sarsa is that it introduces the max-operator. Thus the update rule can be formulated as follows:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma \max_a Q(s',a) - Q(s,a)) \tag{1.22}$$

The actions chosen are derived from Q using an $\epsilon - greedy$ method. The policy is still included in the algorithm even though it is an off-policy algorithm, as it determines which state-action pairs are visitied and updated. All that is required for convergence is that all pairs continue to be updated, which is a milder requirement than in Sarsa.

1.3.5 Problem of Continuous State and Action Space

We can quickly see the problem with the methods proposed so far; they are tabular methods. This means that for any given state there is a specific action. To implement this we have to have one memory cell for every combination of state and action, but what if we have continuous state spaces? Or what if the action space is continuous as well? It is evident that this would require infinitely many memory cells as the experience of the agent was gathered.

One solution to this problem is to discretize the state and action space. But this brings with it its own problems; how finely grained do we have to do the discretization? It is expected that the smaller discretization steps, the better the model. However, we quickly run into the problem of having an enormous table for the action-state values. It is clear that we need a radically new approach to solve this kind of problem.

1.4 Deep Reinforcement Learning

1.4.1 Policy gradient methods

When working with continuous state and action spaces, we need a new class of methods, as discussed in section 1.3.5. There we turn to a class of methods that learns a *parametrized policy* that can select an action based on observations without consulting with a value function, as previous reinforcement learning algorithms have done as explained in section 1.3.2. This parametrization of policy can be done with many types of models, but here we only consider models consisting of Artificial Neural Networks, as explained in section 1.2.2. We use the symbol $\theta \in \mathbb{R}^{d'}$ to denote the policy's parameter vector, in our case the weights and biases in the neural network. We can then write:

$$\pi(a|s,\theta) = Pr(A_t = a|S_t = s, \theta_t = \theta) \tag{1.23}$$

for the probability that an action a is taken at time t, when the environment is at state s at time t with parameter vector θ.

The core idea of policy gradient methods, is that we want to increase the performance of the parametrized policy. This performance is measured using some performance measure function $J(\theta)$ that returns a scalar with respect to the policy parameters. These methods work by maximizing the performance of a given parametrized policy, which is done by an approximate *gradient ascent* in J:

$$\theta_{t+1} = \theta_t + \alpha \Delta \hat{J}(\theta_t) \tag{1.24}$$

Here $\Delta\hat{J}(\theta)$ is a stochastic estimate whose expectation tries to approximate the gradient of the performance measure function $J(\theta)$ with respect to the policy's parameter vector θ_t.

This can be expressed by the following maximization problem:

$$\max_{\theta} E\left[\sum_{t=0}^{H} R(s_t) \middle| \pi_\theta\right] \tag{1.25}$$

where we want to set the parameters in the parameter vector θ such that we maximize the expected sum of rewards $R(s_t)$ for the whole trajectory given a policy π_θ.

An important property about the now parametrized policy $\pi_\theta(a|s)$ is that it is stochastic: we can read it as the probability of taking action a in state s. This makes it a smooth, continuous differentiable function, in contrast to the policies presented in section 1.3.2. Another important property of this policy changing smoothly as a function of the learned parameter, is at very sudden and dramatic changes are much less likely. This contrasts the previously mentioned methods where $\epsilon - greedy$ was used, where the selection of action probabilities could change dramatically for an arbitrarily small change in the estimated action values. This also brings with it better convergence guarantees.

How do we change the policy parameters in a way that ensures improvement? The problem is that the performance of a policy is dependent on both the action selection and the distribution of states in which those actions are made. Both of these are affected by the policy parameters. If we isolate one state, it is simple to see the effect of the parameter changes on the actions and thus the rewards, as this is just

a simple matter of feeding forward the different parameters in our policy approximation network. But knowing the effect of the policy on the state distribution, i.e. what states the agent will see if a parameter is changed, will be a function of an unknown environment.

The *policy gradient theorem* provides an answer to this problem. It gives us a theoretic and analytic expression of the performance gradient with respect to the policy parameters, and it does so by *not* including the derivative of the state distribution.

The *policy gradient theorem* for the episodic case is formulated as follows:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{1.26}$$

The gradients are column vectors of partial derivatives with respect to the components of θ, and π represents the policy corresponding to the parameter vector θ. We have that in the episodic case, this theorem is true up to proportionality, where the constant of proportionality is equal to the average length of an episode. In the continuous case, this theorem is an equality as the constant of proportionality is 1. The function $\mu(s)$ denotes how much we care about the error in each state of s, and is thus a function of the weight of the state, or rather how often a specific state occurs.

In the following subsections, the necessary algorithmic building blocks will be introduced one by one, until reaching the DDPG algorithm.

Monte carlo policy gradient

To make the Policy gradient theorem 1.26 applicable to our algorithm, we need to develop it further. We see that the Policy gradient theorem sums over all states weighted by how often they occur ($\mu(s)$). This term and the corresponding summation over all states s can be removed, as the agent following the policy π will visit these states with the same probabilities. We thus replace s with S_t in the other terms of the expression, which corresponds to the expectation under the policy π which will then be sampled. Another key observation is that our sampled policy gradient only needs to be proportional to the real gradient, since any proportional term will be absorbed into the learning rate α. We then have:

$$\nabla J(\theta) = E_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \tag{1.27}$$

We can integrate this straight into our general policy gradient ascent framework in equation 1.24 and have:

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(S_t,a) \nabla \pi(a|S_t,\theta) \tag{1.28}$$

However, we see that now we are summing over all actions in each state, which is physically not feasible in our use case. Therefore it needs to be changed such that we only consider the action actually taken in a given state. We do the same trick as we did in 1.27, but with the summation over the actions instead. The summation over possible values, is replaced with an expectation under π and then sampled:

$$\nabla J(\theta) = E_\pi \Big[q_\pi(S_t, A_t) \nabla \pi(A_t|S_t, \theta) \Big] \tag{1.29}$$

To compensate for the fact that the different actions have different probabilities, we will divide by $\pi(A_t|S_t,\theta)$. This will ensure that the gradient updates will not be skewed towards those actions that already are visited frequently. An action with twice the probability of being executed than a given action, will have a gradient half the length. We also use the fact that $E_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t)$ to arrive at the following equation:

$$\nabla J(\theta) = E \left[G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \right] \tag{1.30}$$

This is the update rule for the RL algorithm REINFORCE [37]. It is worth noting that this update rule is often presented using the more compact notation with a natural logarithm. Using the identity $\nabla ln(x) = \frac{\nabla x}{x}$, we can replace the fraction with $ln(\pi(A_t|S_t,\theta))$ resulting in the REINFORCE stochastic gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha G_t ln(\pi(A_t|S_t,\theta)) \tag{1.31}$$

The intuition behind this gradient ascent is the following: Every increment is proportional to the product of the return G_t and a vector denoting the gradient of the probability of taking a specific action divided by the probability of taking said action. Following the gradient of the probability of an action means increasing the probability of that action happening again, and when this is multiplied with the discounted sum of rewards, we can see that actions with higher returns will have their probability increased proportionately. Since this is an algorithm that samples the rewards through a whole episode, and then uses that sequence of rewards to calculate the discounted sum of future rewards at each time step, this is a Monte Carlo based algorithm.

Baseline

We can generalize the Policy gradient theorem (1.26) to include a comparison between the action value and an arbitrary *baseline* b(s):

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \big(q_\pi(s,a) - b(s) \big) \nabla \pi(a|s,\theta) \tag{1.32}$$

The baseline function can be any function as long as it does not vary with a. This changes the stochastic gradient ascent algorithm to the following:

$$\theta_{t+1} = \theta_t + \alpha \big(G_t - b(S_t) \big) \frac{\nabla \pi(A_t|S_t,\theta_t}{\pi(A_t|S_t,\theta_t)} \tag{1.33}$$

The reason one would want to do this, is to reduce variance as explained in [33, p.329]. The baseline is of approximately the same magnitude as the values of an action. This makes the magnitude gradient correspond to some *advantage* of an action compared to its baseline. Actions with high value have high baselines, and vice versa.

This is where the intuition from 1.3.3 comes in. By comparing the Q-value (or in this case the discounted sum of future rewards) with a baseline, or value of a given action, we have now arrived at the *advantage function*:

$$A(s,a) = Q(s,a) - V(s) \tag{1.34}$$

The baseline has now been replaced by a value function. This advantage function tells us how much better taking an action in a given state is, compared to the estimated value of that given state.

Actor-Critic methods

If we parametrize this value function $V(S_t)$, meaning we have a separate neural network approximating this function, we have arrived at the so called *Actor-Critic methods*. We now have two models, one for generating an action given a state $\pi(A_t|S_t,\theta_t)$, and one for estimating a value function which we will use as baseline, denoted $\hat{v}(S_t,w)$ where w denotes the parameter vector for the neural network. Again, as previously mentioned, this does not necessarily have to be a neural network, but in our case it is. The training of a value function is a standard supervised-learning problem, as it is trying to predict the value of a given state, and is corrected by the discounted sum of rewards. Since we know that the output of a neural network has some variance, depending on the time it has spent training, this value estimate will thus be a noisy estimate.

The introduction of the estimation of the value function brings *Bootstrapping* with it, as explained in section 1.3.4. The downside of introduction bootstrapping is that we now have introduced bias and an asymptotic dependence on the quality of the function approximation. However it also reduces the variance and accelerates learning [33, p.124]. We integrate these advantages into our Actor-Critic policy gradient method by using a bootstrapping critic. We further develop the general stochastic gradient ascent framework:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha\Big(G_t - \hat{v}(S_t,w)\Big)\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\ &= \theta_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},w) - \hat{v}(S_t,w)\Big)\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}\end{aligned} \tag{1.35}$$

The term G_t can be replaced by $R_{t+1} + \gamma\hat{v}(S_{t+1},w)$ where γ denotes the discounting factor. Note the similarities between equation 1.35 and the previously discussed TD(0) algorithm 1.20.

This is summed up in the method "One-step Actor-Critic episodic algorithm", shown in algorithm 1.

1.4.2 The Deep Deterministic Policy Gradient Algorithm

We are now ready to derive the Deep Deterministic Policy Gradient (DDPG) algorithm [17] which is used in this thesis.

Critic network

We begin by defining a critic network $Q_c(s, a|\theta^Q)$. This is an state-action value function, like the Q-function defined in 1.8. It denotes the value of taking an action a in a given state s and then following a policy π afterwards. However, since we are working in an actor-critic continuous setting, we will use a neural network to approximate this Q-function with the parameter vector θ^Q. The loss function for this neural network should look familiar to equation 1.2, but using the Q-value gradient from Sarsa 1.3.4:

$$L(\theta^Q) = E\Big[(Q(s_t, a_t|\theta^Q) - y_t)^2\Big] \tag{1.36}$$

where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \tag{1.37}$$

Actor network

Now it is time to evaluate the actor network. We want the actor network to learn a policy $\mu(s|\theta^\mu)$ parametrized by the parameter vector θ^μ, that gives an action that maximizes the Q-function $Q(s_t, a_t|\theta^Q)$ approximated by the critic network:

$$\max_{\theta^\mu} E\Big[Q(s_t, \mu(s_t|\theta^\mu)|\theta^Q)\Big] \tag{1.38}$$

To maximize this Q-function, we again turn to stochastic gradient ascent. In this case the gradient is calculated as:

$$\nabla_{\theta^\mu} J \approx E\Big[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\Big] \tag{1.39}$$

By applying the chain rule to this expression, we get the following *policy gradient*:

$$\nabla_{\theta^\mu} J \approx E\Big[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}\Big] \tag{1.40}$$

This is in fact very similar to the update rule of the basic actor-critic gradient ascent in equation 1.35, even though it may look quite different. We recognize that equation 1.40 is a product of the gradient of the Q-function with respect to the action and the gradient of the policy with respect to its own parameter vector. This means that actions that gives an advantage compared to the value provided by the critic-network will have its probability of being chosen again, increased, and vice-versa.

Replay buffer

An innovation introduced in the paper for Deep Q-networks [20], is the use of a replay buffer. This is a cache that stores each tuple $\langle s_t, a_t, r_t, s_{t+1}\rangle$ and has a finite size. When it fills up, each new tuple will delete the oldest one. Since we are dealing with an off-policy algorithm, this means that we separate the learning process from the actions taken. Whenever a neural network is trained, the training data should be independent and identically distributed. If one fed the experience tuples straight into training, there

would be a strong dependency between the data samples, which would cause problems when training the network. DDPG solves this by randomly sampling a minibatch of experience tuples from the replay buffer, ensuring that each experience tuple is as independent as possible. This replay buffer can be very large, which enables the algorithm to learn from a set which will be very uncorrelated.

Random processes

How should we make our agent explore the environment? In the tabular methods we could use the $\epsilon - greedy$ methods to choose a random action with a probability ϵ. In the continuous action-space setting however, this does not work. We need a way of perturbing the actions chosen by the agent so as to make the agent venture into unknown territory. To do this, we can add a sample from a noise process and add it directly to our actor policy:

$$\mu'(s_t) = \mu(s_t, \theta_t^{\mu}) + \mathbb{N} \tag{1.41}$$

This can be done independently from the learning algorithm because DDPG is an off-policy algorithm. An important property of the noise process $\mathbb{N}$ is that its variance does not grow infinitely, but is bounded. This is because we want to balance exploration and exploitation. If the random process returns 0, it will be exploitation of the agents policy, while any other perturbations away from zero will result in exploration of the state space. In our implementation we use a normal gaussian noise with expectation $\mu = 0$ and standard deviation $\sigma = 0.1$.

Target networks

A challenge when training the critic network is that the network $Q(s, a|\theta^Q)$ is being updated at the same time as it is used as a target in equation 1.37, and this has been shown to be unstable in many training environments [17, p.4]. To make sure there is a temporal gap between the network update and the calculation of the target, DDPG introduces "soft updates" by creating a copy of the actor and the critic, $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$ respectively, with parameter vectors $\theta^{Q'}$ and $\theta^{\mu'}$. These networks are used for the calculation of target values in equation 1.37, and their weights are constrained to change slowly, greatly improving the stability of learning. Their parameter vectors are updated using the following method:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{1.42}$$

with $\tau << 1$.

Batch normalization

To make sure that different observations with different units and magnitudes are considered equally, DDPG implements *batch normalization* to normalize each dimension across the observation samples in a batch to have unit mean and variance. To make sure that the data is normalized correctly during testing, the algorithm keeps track of a running average of the mean and variance. This makes the algorithm learn effectively in a range of different environments.

The DDPG algorithm is illustrated in figure 1.3, and the pseudo-code is found in algorithm 2.

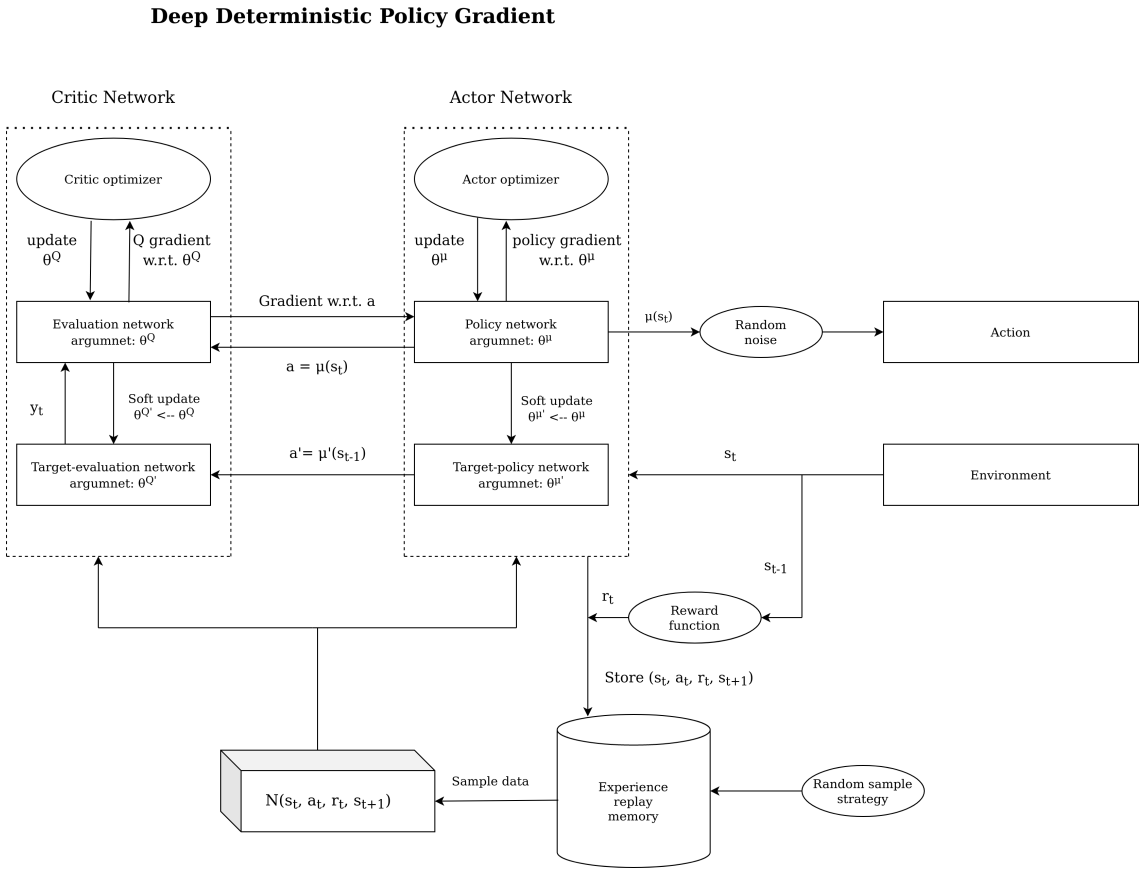


Figure 1.3: The structure of the Deep Deterministic Policy Gradient algorithm

In this thesis, the DDPG algorithm has been heavily utilized, and all the results presented later will be relying this algorithm. However, this is not currently considered to be a state-of-the-art algorithm even though it is relatively new (2016). Now, the focus has shifted more towards trust-region methods like TRPO and PPO [23]. A short introduction to each of these two methods are included below.

1.4.3 The Trust-Region Policy Optimization Algorithm

The general motivation behind trust-region policy optimization algorithms is that the choice of step length α when doing stochastic gradient ascent is a crucial element to having the algorithm converge properly. In DDPG, a constant step length was chosen, which meant that the gradient ascent could stride over and miss optimal solutions, or even diverge after having found an optimal solution. The rationale behind this is that the direction of the gradient usually is well calculated, but the step length is not chosen in such a way that it necessarily improves the overall function value. Trust-region methods chooses a step length α in such a way that it guarantees monotonic improvement. This is analogous to the Levenberg-Marquadt optimization algorithm, which takes the gradient from the Gauss-Newton method and chooses a step length such that the objective function is monotonically decreasing [21]. However, choosing a bad step size in reinforcement learning is more critical than in standard optimization problems, as out input

data is non-stationary and depends on the actions of the agent. A step too far in the wrong direction will lead the actor to a bad policy, which then will affect the next batch of observations.

Recall the general update rule from the stochastic gradient ascent framework in equation 1.35 which is using the advantage function A_t from equation 1.34. This can be compactly written as:

$$\nabla_\theta J(\theta) = \hat{E}_t\Big[\nabla_\theta log\pi_\theta(a_t|s_t)\hat{A}_t\Big] \tag{1.43}$$

where we use hats $\hat{}$ to emphasize that we are working with estimates.

We can rewrite this policy gradient as a loss function that we want to maximize:

$$L^{PG}(\theta) = \hat{E}_t\Big[log\pi_\theta(a_t|s_t)\hat{A}_t\Big] \tag{1.44}$$

which can be rewritten to use importance sampling where state-actions are sampled using θ_{old}:

$$L^{IS}_{\theta_{old}}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \tag{1.45}$$

The core idea of Trust-region policy optimization (TRPO) is that this policy update should not move too far from the old policy. To ensure that this is the case, we will include a constraint that limits the step size. Kullback–Leibler (KL) divergence is a way of measuring how one probability distribution is different from a second, reference probability distribution, and is exactly what we need for making sure that the policy does not change too rapidly. The update rule for TRPO can thus be written as follows:

$$\begin{aligned} \max_\theta \quad & \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \\ s.t. \quad & \hat{E}_t\big[KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(., s_t)]\big] \leq \delta \end{aligned} \tag{1.46}$$

The pseudocode is found in algorithm 3. For further details, see the original paper [30].

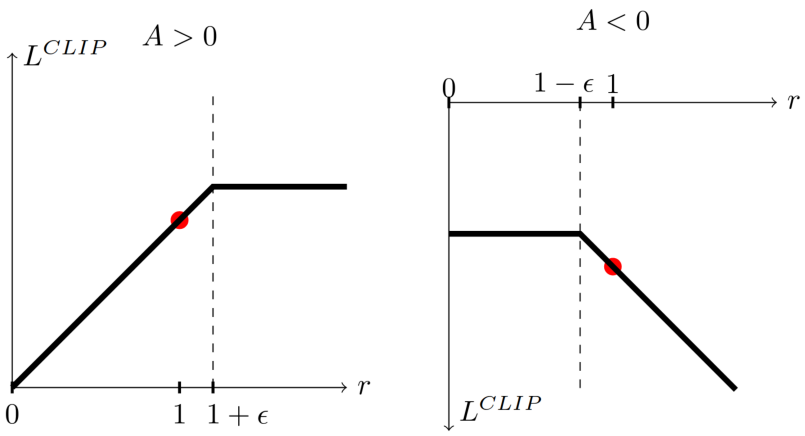
1.4.4 The Proximal Policy Optimization Algorithm

The Proximal Policy Optimization (PPO) algorithm is an iteration of the TRPO algorithm that simplifies implementation and decreases computational complexity in training, while having comparable performance to other state-of-the-art algorithms [29]. This is an on-policy algorithm, as there is no replay buffer as we saw in DDPG 1.4.2, and the agent learns directly from what it encounters in the environment.

The main contribution of PPO is eliminating the computationally expensive process of calculating the KL-divergence, and doing backtracking line search (see algorithm 3). It instead includes this constraint straight into the optimization objective, so that these extra steps can be avoided.

To derive this optimization objective we will define the probability ratio $r_t(\theta)$ which hopefully will look familiar:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{1.47}$$



(a) The case when the advantage estimate is positive

(b) The case when the advantage estimate is negative

Figure 1.4: Clipped objective function visualized with respect to the clipped probability ratio

Thus we have that $r_t(\theta_{old}) = 1$. We recall the objective function from TRPO:

$$L^{CPI}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{1.48}$$

As we saw in TRPO, maximizing this objective function may lead to an excessively large policy update. This may push the policy network into a region of parameter space where it is going to collect the next batch of data under a very poor policy causing it to never recover again. TRPO solved this by using the KL-divergence constraint to make sure that the new policy did not stray too much from the old one. In PPO, we will use a clipped objective function:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{1.49}$$

This may look simple, but it has some interesting properties which are not immediately obvious. We divide into two distinct cases: when the advantage estimate is positive 1.4a and when it is negative 1.4b. Beginning with the case of having positive advantage function, we see that when the probability ratio $r_t(\theta)$ is positive it is clipped so that it only can attain values which are $1+\epsilon$ larger than $r_t(\theta_{old})$. This limits how far the gradient ascent will step in the direction of increasing the probability of the given action.

Looking at the case of having a negative advantage estimate, we see that if the action becomes less likely, we limit the decrease in probability for that given action by making sure that the probability ratio $r_t(\theta)$ does not decrease more than $1-\epsilon$. When the action taken was both bad (negative advantage estimate) and also more probable, we see that we have now entered the case in equation 1.49 in which the first value $r_t(\theta)\hat{A}_t$ is returned by the min()-operator. This corresponds to undoing the update that made the bad action more probable in the first place, and the update step is proportional to how bad the update was in the first place

1.5 Quadrotor Dynamics

In this section, the dynamics of the quadrotor helicopter are discussed, as illustrated in figure 1.5. We will use two coordinate systems to describe the attitude and position of the quadrotor, the inertial reference frame $\{\vec{e}_{11}, \vec{e}_{21}, \vec{e}_{31}\}$ and a body-fixed frame $\{\vec{e}_{1B}, \vec{e}_{2B}, \vec{e}_{3B}\}$. The origin of the body-fixed frame is placed at the center of mass of the quadrotor, and we can see that the first and second axes ($\{\vec{e}_{1B}, \vec{e}_{2B}\}$) of the body frame points span out the plane on which all the rotors lie. The last axis $\vec{e}_{3B}$ is placed normal to this plane, and in the opposite direction to the thrust vector.

Each rotor contributes with a thrust F_i, and they are all placed with equal distances to the center of mass. To balance their torques, the rotors spin in the opposite direction of their neighbors.

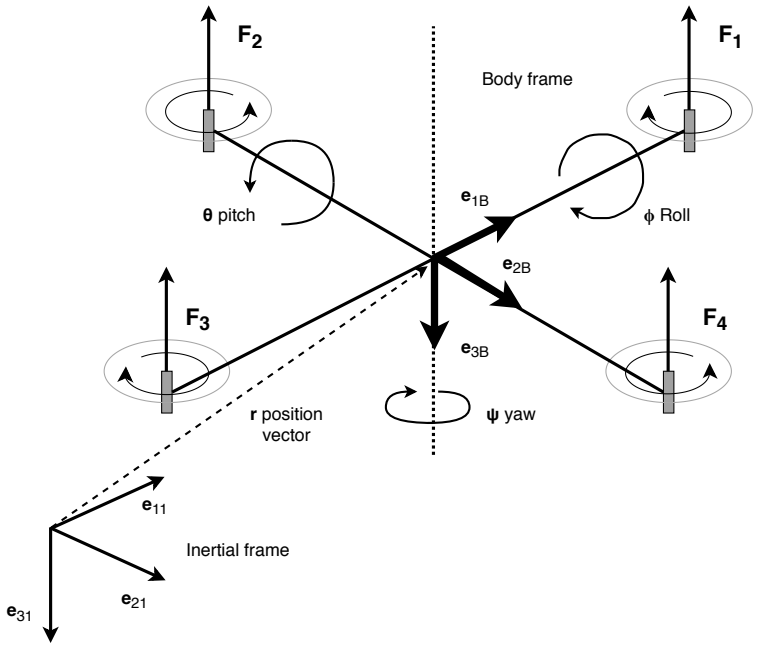


Figure 1.5: Dynamic model of the quadrotor

We define:

$m \in \mathbb{R}$	the total mass of the quadrotor
$J \in \mathbb{R}^{3\times3}$	the inertia matrix with respect to the body-fixed frame
$R \in SO(3)$	the rotation matrix from the body-fixed frame to the inertial frame
$\Omega \in \mathbb{R}^3$	the angular velocity in the body-fixed frame
$x \in \mathbb{R}^3$	the location of the center of mass in the inertial frame
$v \in \mathbb{R}^3$	the velocity of the center of mass in the inertial frame
$d \in \mathbb{R}$	the distance from the center of mass to the center of each rotor in the $\vec{e}_{1B}, \vec{e}_{2B}$ plane
$f_i \in \mathbb{R}$	the thrust generated by the i-th propeller along the -$\vec{e}_{3B}$ axis
$\tau_i \in \mathbb{R}$	the torque generated by the i-th propeller about the $\vec{e}_{3B}$ axis
$F \in \mathbb{R}$	the total thrust, i.e. $F = \sum_{i=1}^{4} F_i$
$M \in \mathbb{R}$	the total moment in the body-fixed frame

Table 1.1: Parameters of the quadrotor helicopter

The configuration of our system is thus defined by the location of its center of mass and its attitude with respect to the inertial frame. This means that it lies on the manifold of the special Eucledian group SE(3).

To derive the equations of motion for the quadrotor, we assume that the thrust of each propeller is directly controlled, and that the torque generated by each propeller is directly proportional to its thrust. This means that we are not considering the dynamics of the rotors and propellers. The direction of the thrust is defined to be normal on the quadrotor plane, and thus the total thrust f acts in the direction of -$\vec{e}_{3B}$. Since we now have the total thrust defined in the body frame, we only need to multiply the vector e_{31} with the rotation matrix R to get the total thrust in the inertial frame: $-fR\vec{e}_{31}$.

As previously mentioned, we have the propellers spinning in the opposite direction of its neighbors to balance the torque. This means that the first and the third propellers are rotating clockwise, and the second and fourth rotating counterclockwise when they are generating positive thrust. We write the torque of the i-th propeller as $\tau_i = (-1)^i c_{\tau f} F_i$ for a fixed constant $c_{\tau f}$. We are now ready to formulate the relationship between the total thrust F and the total moment M:

$$\begin{bmatrix} F \\ M_1 \\ M_2 \\ M_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -d & 0 & d \\ d & 0 & -d & 0 \\ -c_{\tau f} & c_{\tau f} & -c_{\tau f} & c_{\tau f} \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \tag{1.50}$$

This matrix is invertible when $d \neq 0$ and $c_{\tau f} \neq 0$, as its determinant is $8c_{\tau f}d^2$. This means that for a given F and M, we can find the corresponding thrust F_i for each rotor.

The equations of motion for the quadrotor is thus:

$$\dot{x} = v \tag{1.51a}$$

$$m\dot{v} = mg\vec{e}_{31} - FR\vec{e}_{31} \tag{1.51b}$$

$$\dot{R} = R\hat{\Omega} \tag{1.51c}$$

$$J\dot{\Omega} + \Omega \times J\Omega = M \tag{1.51d}$$

where the *hat map* $\hat{\cdot} : \mathbb{R}^3 \to SO(3)$ is defined as a cross-product operation such that $\hat{x}y = x \times y$ for all $x, y \in \mathbb{R}^3$

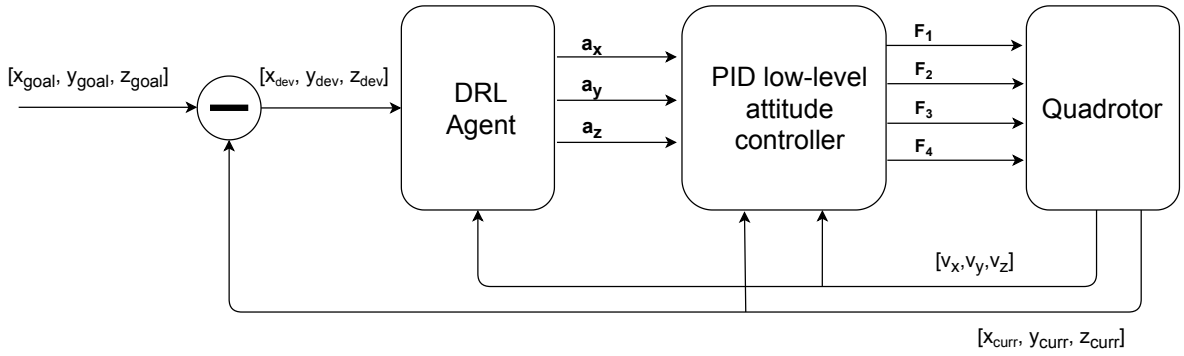


Figure 1.6: The DRL agent in the control loop

1.6 Problem Formulation

The goal of the RL agent is navigating a quadrotor to a goal. The quadrotor itself is already stabilized using a PID-controller. As stabilizing using reinforcement learning already has been solved in papers like [12], we save time by skipping this complex problem. This thesis focuses on how well the RL agent is able to navigate a quadrotor from a start position to a goal position, given information about the deviation in its position and its speed. The DRL agent fits into the control diagram as illustrated in 1.6, where it receives its current positional deviation and its speed, and outputs the desired acceleration vector. The PID low-level controller turns that desired acceleration vector into the control signals for each of the four quadrotor motors.

1.7 Related work

Our problem of flying from an initial position to a goal position is a path following problem, and differs from a trajectory tracking problem in that it does not have a time dependence. In our case since we only have one goal point, this can be seen as a path following problem with only one *waypoint*. There exists several solutions to this problem. We will present the most relevant methods.

Backstepping is a path following technique based on Lyapunov theory, and its control objective is to make sure a set of predefined errors converge to zero. This is done by defining a Lyapunov function for each error such that their time derivatives are negative definite, thus assuring stability. The control actions are then those actions that make these time derivatives negative definite for all the Lyapunov functions. Using *backstepping* as a path following problem, it is possible to obtain global convergence while keeping the control capability for quadrotors [4].

Lyapunov-based path following algorithms are similar to *backstepping* in that they are also based on Lyapunov theory. They assure the Lyapunov stability condition which in turn assures the convergence of the controller. Examples of control laws generated using this approach can be seen in [5]. In [6] the control is formulated using the rotation matrix and based on Lyapunov theory, and the controller generates angular rates and thrust reference commands. Another similar and interesting method for path following is using a Lyapunov-based controller together with a velocity observer and a constant disturbance

estimator [19].

Feedback Linearisation is a very popular technique for quadrotor control. It works by linearising the system in a given region in the state space by identifying the non-linearities and effectively cancelling them out. This makes it possible to directly apply linear control methods. The advantages of choosing such a method is the simplicity in the resulting controller and that we can do stability analysis to guarantee stability under a given set of conditions. Examples like [26] demonstrates how to solve a 3D path following problem by using input-output *Feedback Linearisation*, and how it converges to its path and stays there. Another interesting implementation of this method is in [2] where the authors show how to do path following in the case when one of the rotors malfunctions.

Geometric methods are the methods that use simple geometric strategies to converge to a path. In this categories we have algorithms like *Carrot-chasing* algorithm, *Non-linear guidance law*,*Pure pursuit*, *Line-of-sight* and *Trajectory shaping* guidance law. They all have in common that they use geometric principles like finding the tangent of the path, drawing circumferences, finding the closest point on the path, minimizing the vehicle heading angle and the path heading etc. Their simplicity and intuitiveness make them very appealing.

Model Predictive Control (MPC) is a popular technique that solves the control problem as an optimization problem. At each time step, an optimization of the systems path is done by solving a finite horizon optimal control problem, and the first step in said path is then executed. In the next time step, exactly the same process repeats. This process requires more computational resources, depending on the complexity of the optimization problem. It is also memory intensive as the necessary memory grows fast as the time horizon is increased. The advantage of using such a method is that it can handle constraints in state space and in action space, non-linear dynamics and non-linear reference paths. [22] is an example of an implementation of a Nonlinear Model Predictive Control (NMPC) used for path following, which is then integrated into a cascaded control architecture. A very interesting implementation is using the MPC as an expert in an Imitation learning setting, where one trains a policy network using the control outputs from an MPC [32]. This reinforcement learning approach is more computationally efficient than using MPC, as it computes control commands directly from sensor inputs.

Using the concepts in *Optimal Control*, we can also solve the problem of path following. The goal of *optimal control* is to actuate a dynamic system with minimum cost, which translates to following a path with minimum error and control effort. The most common control techniques are the *Linear Quadratic Regulator* (LQR) and the *Linear Quadratic Gaussian* (LQG). A general solution to the path following problem using optimal control is presented in [35], and depends on a geometric formulation based on the concept of differential flatness. Another implementation of path following using *optimal control* can be found in [15], where they use an adaptive LQR which is optimized using a genetic algorithm.

A solution to this problem using deep reinforcement learning is provided by the paper [27]. Here, the problem of altitude, attitude and velocity control is solved by an external controller, and the RL agent provides commands for altitude, yaw, and velocity commands in the x and y direction. This is similar to the setup that we will be using in this thesis.

1.8 Experimental setup

To do the training of the RL agent we ran the quadrotor using simulation. This frees us from having to interact with a physical drone to gather data, which is crucial when trying to train a RL agent as one can speed up the simulation and converge faster to a solution. It is also necessary from a safety perspective, as the quadrotor inevitably will crash during training.

1.8.1 High-level RL agent

The DDPG algorithm used to train the agent is adapted from OpenAI baselines [8]. This agent receives observations from the environment in the form of global position coordinates and velocity, and outputs an acceleration vector which can span all three coordinate axes (x,y,z). It is written using Python 3.8, and the backend for the machine learning processes is provided by Tensorflow [1].

1.8.2 Simulation environment

Gazebo is a simulation software which is designed to do 3D dynamic simulations in an easily adaptable fashion [14], and is the simulation backend we will be using to simulate the quadrotor. The model and dynamics of the quadrotor itself, is provided by RotorS [9], which is a Micro Aerial Vehicle simulation framework that is made to make solving high-level problems like collision avoidance, path planning, and vision based problems, like Simultaneous Localization and Mapping (SLAM), possible in simulation. On top of it all, we use Rviz [13] for the visualization of the quadrotor Pose and goal position to be able to turn off rendering in gazebo to save computation.

1.8.3 Software system overview

All the components of the system were tied together with the Robot Operating System (ROS) [24]. The software architecture is visualized in the figure 1.7. Here the node $rotors\_wrapper$ is a wrapper that wraps the ROS and Gazebo framework into an environment in which the high-level RL agent is able to interact with. $\backslash delta$ is the quadrotor itself, with the *roll-pitch-yawrate-thurst controller*. It has inputs from the high-level RL agent and outputs its commands into the Gazebo simulator. Finally, the state of the quadrotor and the goal is sent to *rviz* for visual illustration.

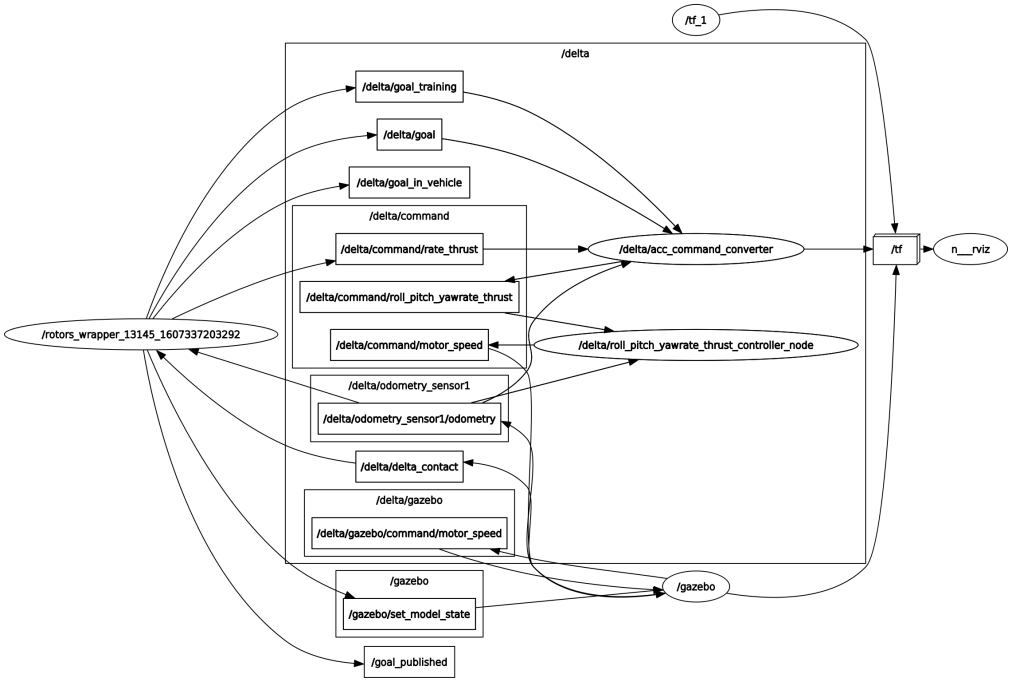


Figure 1.7: Software architecture of the experimental setup

1.9 Reward function

The performance of a RL agent is very dependent on the type of reward function used. This will guide the agent to make optimal decisions, as the rewards will tell it if a given action was good or not. There are mainly two types of reward functions: *dense* and *sparse*.

In *dense* reward functions, the agent gets a reward very often or at every time-step, and will thus be guided carefully towards optimal behaviour. The opposite is the case for *sparse* reward functions: these will reward the agent at the goal or any number of subgoals, and the agent will thus receive rewards infrequently.

The reward function for this thesis will be a mix of the two. When the quadrotor is far from its target, it uses a quadratic reward function as seen in equation 1.54. The control signal u and the positional deviation and speed x are squared using the weighting in R and Q, and the reward is the negative sum of these two. The squared penalty for deviation in position is illustrated as the blue surface in figure 1.8. To increase the reward the agent will try to do gradient ascent on this blue surface, and we see that when the quadrotor is far away from its target, this negative quadratic reward will give a strong indication as to which direction to go. This means that far away from the goal we have a *dense* reward function.

However when we approach the goal, we can see that the indications as to where to point our gradient in figure 1.8 decrease quadratically. This means that the cost benefit of moving one unit of distance closer to the goal is much greater far away from the goal compared to closer to the goal. This is where the *goal reward* is introduced: whenever the quadrotor gets inside a given goal radius δ_{pos}, it receives a goal reward of 1000, while the quadratic cost of the control signal remains the same, as seen in equation 1.54.

In this sense, when we are looking at the close vicinity of the goal, the reward function acts more *sparsely*, as the magnitude of the goal reward is much greater than the positional deviation cost from the quadratic term.

This reward function is designed to give a big negative reward whenever the quadrotor is far away from its target and give a big positive reward when inside a goal radius. The square cost of the control signal is included to ensure that the quadrotor flies smoother, i.e. gets to the goal while using the minimal amount of control signal. It also ensures that when inside the goal radius, the quadrotor is incentivised to stay still.

$$R = \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \tag{1.52}$$

$$Q = \begin{bmatrix} 0.6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.15 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.15 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.25 \end{bmatrix} \tag{1.53}$$

$$r_t = \begin{cases} 1000 - uRu^T & \text{if } ||pos_{goal} - pos_{quad}||_2 < \delta_{pos} \text{ and } ||vel_{quad}||_2 < \delta_{vel} \\ -uRu^T - xQx^T & \text{otherwise} \end{cases} \tag{1.54}$$

where $u = [a_x, a_y, a_z]$ denotes the acceleration vector and
$x = [x_{goal} - x_{quad}, y_{goal} - y_{quad}, z_{goal} - z_{quad}, v_{x,quad}, v_{y,quad}, v_{z,quad}]$
denotes the deviation in position from the goal and the speed of the quadrotor.

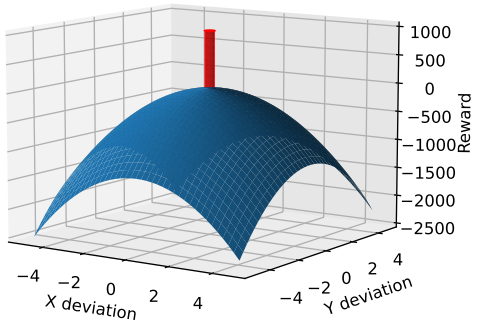


Figure 1.8: Illustration of the reward function in two positional dimensions

Chapter 2

Preliminary results and plots

As previously mentioned, the goal of the RL agent is to quickly navigate from an initial position to a goal state in a rectilinear fashion. The ideal path is a straight line between these two positions, and to measure the robustness of an agent we will measure the Root Mean Squared deviation from this line throughout the quadrotors trajectory.

4 different topologies for the Actor and Critic Networks will be explored and tested to see if there are any differences in the resulting behaviour in the quadrotor. These have all been trained in the same manner to make direct comparisons possible. From now on we will use the term Agent to denote the policy network.

To get a quantitative measure of the agents performance across several runs, we will set 10 goal positions which will be equal for all the agents, and measure the Average Root Mean Squared deviation over all runs. This will enable us to look at a score for each network topology.

2.1 Deep Deterministic Policy Gradient (DDPG)

2.1.1 Network topologies

To test the differences in behaviour when modifying the topology of the neural networks, 4 different topologies are proposed and analyzed. All the configurations have two hidden layers with the activation function ReLu, and their widths are the same on both the Actor and the Critic network. The difference is their input layers and output layers.

The *Critic* network takes as input both the observation (i.e. deviations in position and the quadrotor velocity) and the action (acceleration vector) given from the *Actor* network, and outputs a single scalar value to determine the value of a given state-action combination. There is no activation function in the last output layer, it is just a linear combination of the activations in the last hidden layer. The critic network is illustrated in figure 2.1

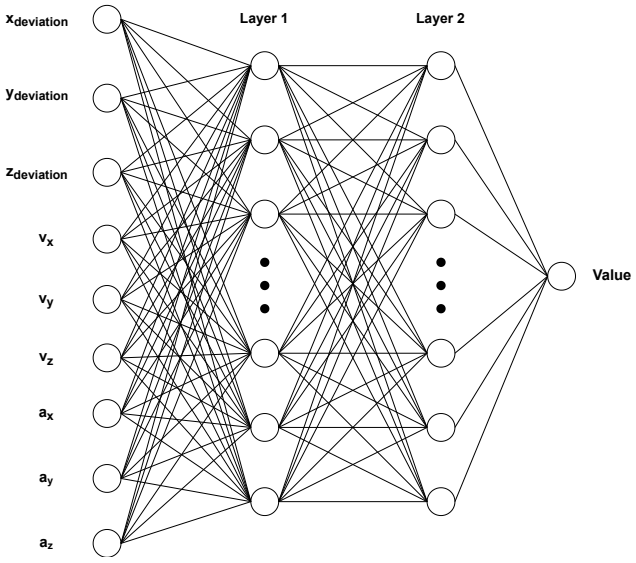


Figure 2.1: Illustration of the Critic neural network

The *Actor* takes as input the observation, which is the deviations in position and the quadrotor velocity, and gives an action, which is the acceleration vector, as shown in figure 2.2. The last layer uses the $tanh$ activation function. This ensures that the outputs of the neural network are in the range $[-1,1]$, and the resulting acceleration vector will then also be constrained. This ensures stability in training as the acceleration vector has a realistic magnitude.

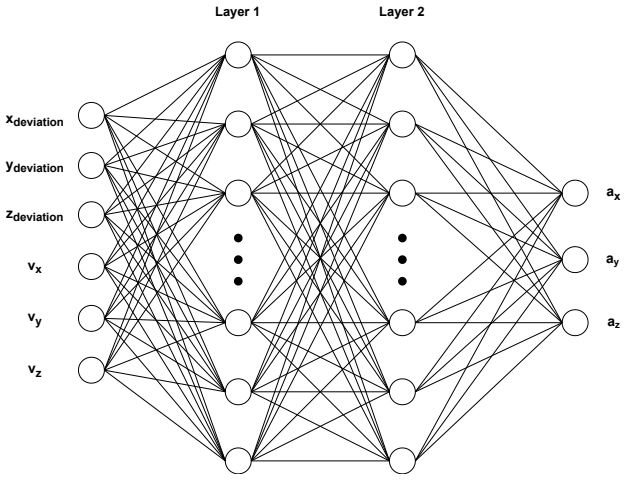


Figure 2.2: Illustration of the Actor neural network

2.1.2 Training

The training consists in the quadrotor starting in the same position every time, and having a goal drawn randomly. The goals are drawn using a spherical coordinate system, where the angles ϕ, θ are drawn with a uniform distribution between 0 and 2π. The radius r is drawn from a uniform distribution between 0 and 1, and transformed using a cubic root before it is multiplied with the goal generation radius of 3m:

	layer 1 width	layer 2 width
Network configuration 1	64	32
Network configuration 2	64	64
Network configuration 3	128	64
Integral network configuration	128	64

Table 2.1: Network configurations in DDPG

$$
\begin{aligned}
&x = r * sin(\phi) * cos(\theta) \\
&y = r * sin(\phi) * sin(\theta) \\
&z = r * cos(\phi) \\
&\text{where} \\
&r = goal\_generation\_radius * \sqrt[3]{d} \\
&d \sim unif(0,1) \\
&\phi \sim unif(0,2\pi) \\
&\theta \sim unif(0,2\pi)
\end{aligned}
\tag{2.1}
$$

The cubic root ensures that the probability of generating a radius closer to 3 m is more probable than a radius closer to 0 meters. **The random sampling of goal points is very important to ensure that the training samples are uncorrelated and that the neural networks are not *overfitting* on the training data, as explained in section 1.2.3**.

To make the results as consistent as possible, the training procedure is standardized to be equal over all network topologies. The initial hyperparameters is set as in Table 2.2, with the reward function as described in Section 1.9. The training progress of each network topology is illustrated in Section B.2.

Parameter	Value
Learning rate actor	10^{-4}
Learning rate critic	10^{-4}
discount factor γ	0.99
target network update rate τ	0.001
goal radius δ_{pos}	0.4
maximum velocity at goal δ_{vel}	0.3

Table 2.2: Initial training hyperparameters

The progress of the different training sessions can be seen in appendix B. There is an important distinction between the training progress in the standard supervised learning setting, and in the reinforcement learning setting. In standard supervised learning, one is looking for convergence by observing the training loss, i.e. the rate at which the neural network learns in each epoch. When this training loss is converging, one knows that the weights in the neural network are also converging on some optima. However, when we talk about convergence in reinforcement learning, we are talking strictly about the average

rollout return converging. Since the input distribution is continually changing and dependent on the current actor policy, the fact that the networks loss is high and thus learns, does not necessarily mean that the network is not converging. If the networks are receiving bad training data, they will just as well show a high training loss even though they may already be in some optima. Thus, when looking at the Actor and Critic loss in each epoch, the only information we can extract from them is by how much the parameter weights are changed after each epoch.

After the networks had converged in the first training session, the networks were trained another round until convergence but with goal radius decremented to 0.2 and the input cost being increased from 0.001 to 0.0015. To ensure optimal training progress, the network weights resulting in the highest rewards were the ones transferred to the second round of training. When testing these networks, the network weights with the highest score from the second training session were selected. My experience when training these networks is that it is better to start width a wider goal radius such that the quadrotor learns faster that it should go towards the goal. If the goal radius is too small to begin with, the quadrotor seldom found this goal and the goal reward was thus not experienced. Because the magnitude of the quadratic cost function diminishes quadratically when one approaches the goal, as explained in section 1.9, the agent needs to experience the goal reward to properly learn that it should go to the goal. The only positive term in the reward function 1.54 is the goal reward.

Looking at the average rollout return per epoch in the first training sessions (Training session 1 for all network configurations) in section B.2, we can see when the agent transitions from mostly learning from the negative quadratic elements, and to learn more and more from the goal reward. When the agent crosses the x-axis, the positive reward balances out the negative penalties, and the average rollout return increases faster per epoch than when the agent mostly learns from the quadratic terms. Ideally the rollout return should look like the figure B.3, where it flattens out and converges to an optima. However, continuing to train the agent after it has reached is optimum may push it off its optimum, as we can see in figure B.1. This is because DDPG is not a trust-region algorithm, and has a constant gradient step length. We can therefore see why an algorithm like TRPO or PPO would do better in this regard.

Another interesting thing to note about the training process, is how information is propagated. Since the Actor network uses the Critic network as baseline, it is depending on this network to give good value estimates to be able to improve its own policy. This means that the information is propagated through the Critic network first, before the information is passed on to the Actor. The Critic learns a good value function using a supervised learning approach on the ever-changing distribution of incoming rewards, and it is first when the Critic has learnt what constitutes a good action or not that the Actors policy actually starts to converge.

It is interesting to note that each batch of 2000 epochs took about 24 hours to run on my computer (the specifications of which are noted for reference in section B), making the computational time for the networks presented in this project about 192 hours.

2.1.3 Adding an integral effect

The idea for including integral effect in the observation space and cost function comes from this paper [36]. They note that reinforcement learning algorithms in continuous state space have a tendency for having a small steady state error as the penalties are very small close to the goal point. In classical control theory, steady-state errors are solved using integral effect. The idea is similar in this case, but is formalized through the change in reward function. This means that when the quadrotor is close to the goal, the integrators will keep integrating the error and generating a bigger penalty, and thus forcing the agent to keep minimizing the distance between itself and the goal. Another way to see it, is that for a changing input in the neural network, the output should not be stationary and thus force the agent to action.

The idea is that we expand the observation space to include an integral of the deviations in the three axes:

$$x = \left[x_{dev}, y_{dev}, z_{dev}, v_{x,quad}, v_{y,quad}, v_{z,quad}, \int_{t=0}^{T} x_{dev}\,dt, \int_{t=0}^{T} y_{dev}\,dt, \int_{t=0}^{T} z_{dev}\,dt\right] \tag{2.2}$$

and we update the Q-matrix denoting the weighting of the elements in the observation vector x for penalizing accordingly:

$$Q = \begin{bmatrix} 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.15 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.001 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 \end{bmatrix} \tag{2.3}$$

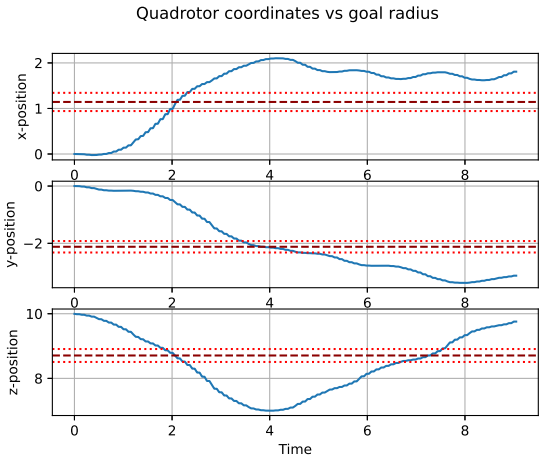
As this is a very different architecture than the three other networks, it is important to note that any direct comparisons may not be well founded. However, adding integral effect may produce some interesting findings.

2.1.4 Results

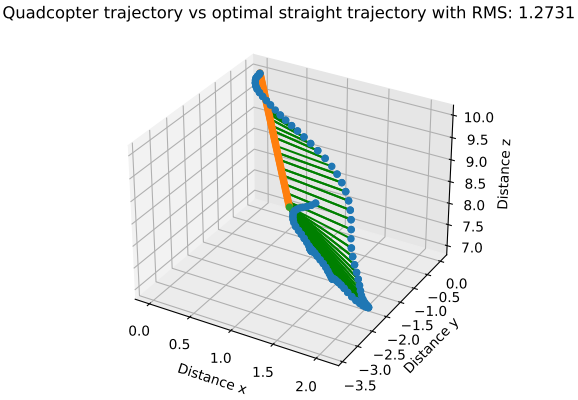
All of the examples below are generated using the same end goal position.

	Average RMS	Success rate
Network config. 1	2.588	0/10
Network config. 2	0.389	10/10
Network config. 3	1.141	7/10
Integral network config.	1.536	5/10

Table 2.3: Average Root Mean Squared distances to optimal trajectory for different network topologies

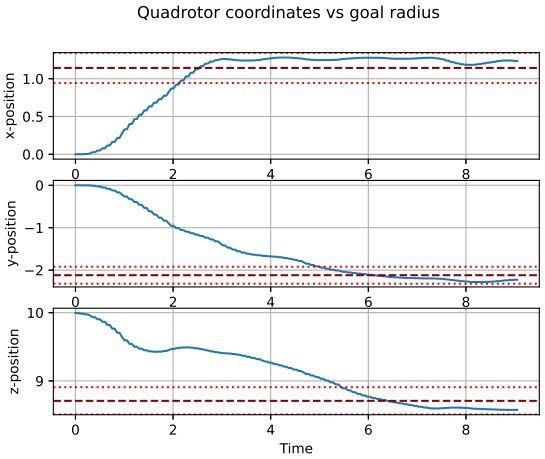


(a) Quadrotor coordinaters vs goal radius

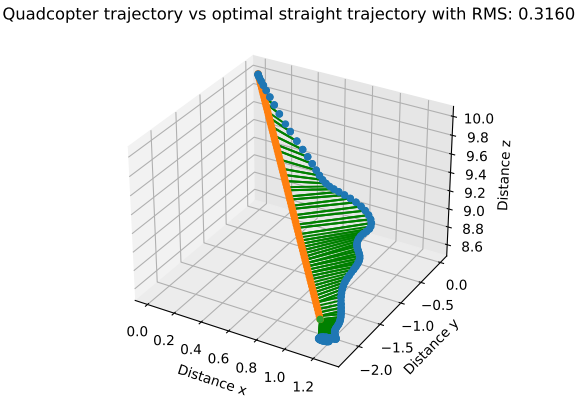


(b) Quadrotor trajectory

Figure 2.3: Example of run from network configuration 1

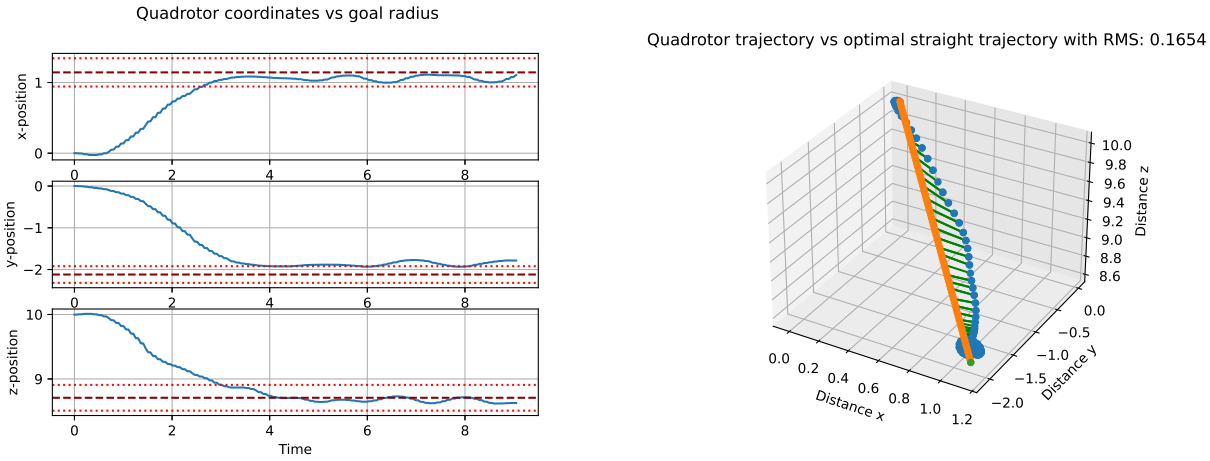


(a) Quadrotor coordinaters vs goal radius



(b) Quadrotor trajectory

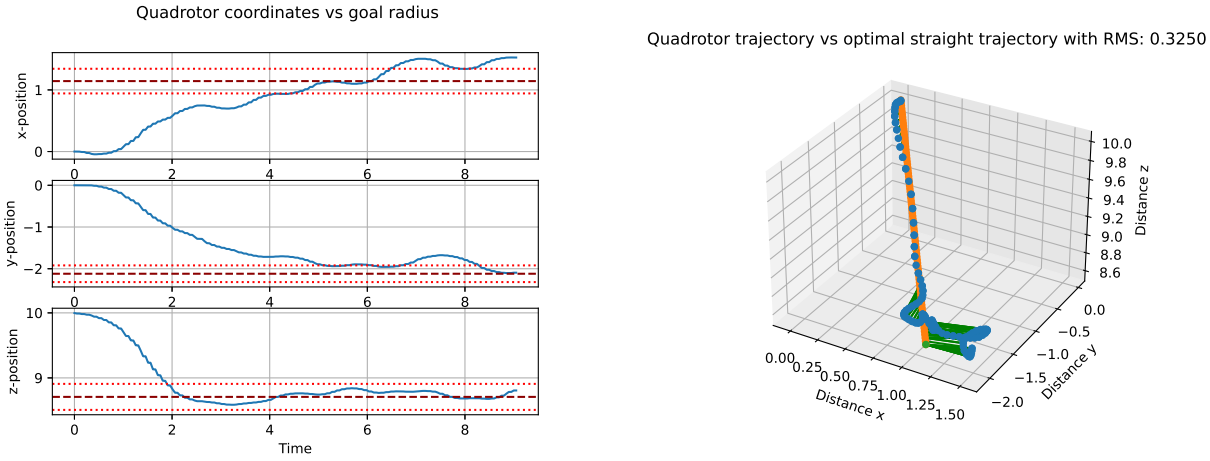
Figure 2.4: Example of run from network configuration 2



(a) Quadrotor coordinaters vs goal radius

(b) Quadrotor trajectory

Figure 2.5: Example of run from network configuration 3



(a) Quadrotor coordinaters vs goal radius

(b) Quadrotor trajectory

Figure 2.6: Example of run from integral network configuration

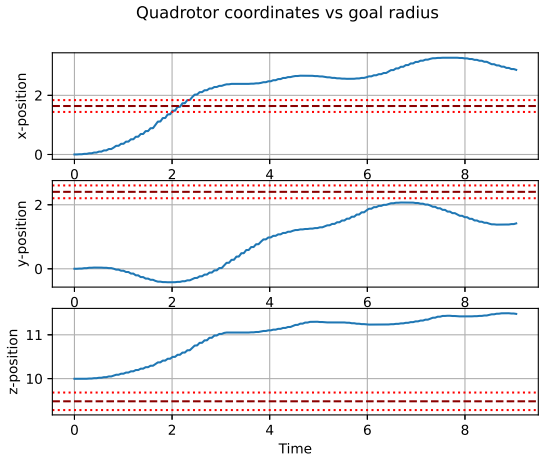
2.2 Testing for robustness

To test how robust the network topologies act when there are slight variations in the model parameters, let us vary the mass by ±10%. All of the examples below are generated using the same end goal position.

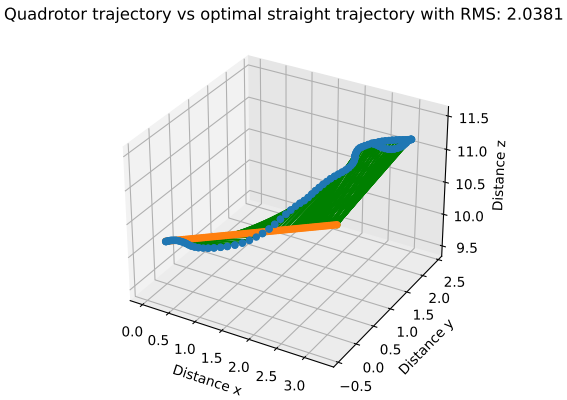
2.2.1 **Results** +10%

	Average RMS	Success rate
Network configuration 1	1.876	0/10
Network configuration 2	2.918	0/10
Network configuration 3	2.337	0/10
Integral configuration	2.748	0/10

Table 2.4: Average RMS values for +10% modified quadrotor weight

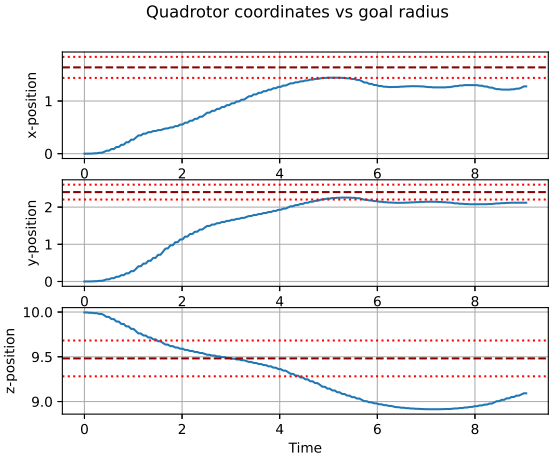


(a) Quadrotor coordinaters vs goal radius

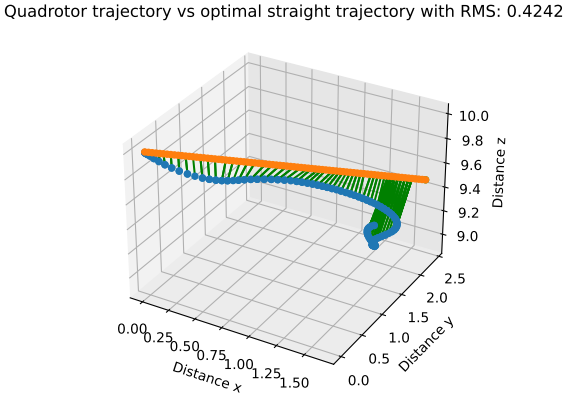


(b) Quadrotor trajectory

Figure 2.7: Example of run from network configuration 1 with +10% modified mass

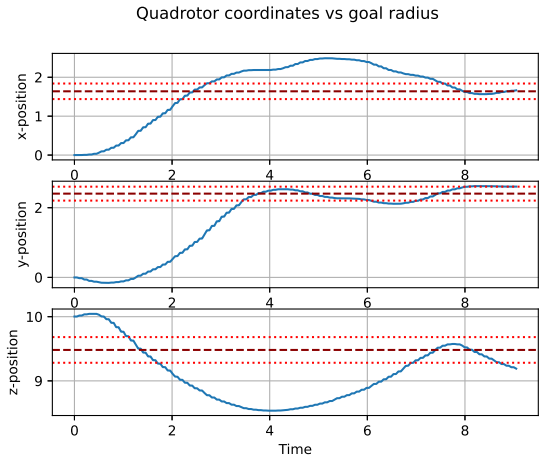


(a) Quadrotor coordinaters vs goal radius

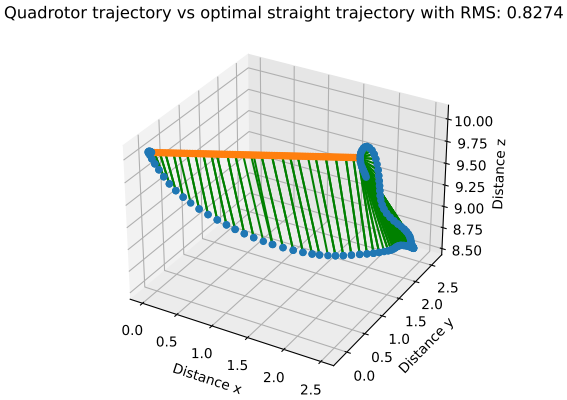


(b) Quadrotor trajectory

Figure 2.8: Example of run from network configuration 2 with +10% modified mass

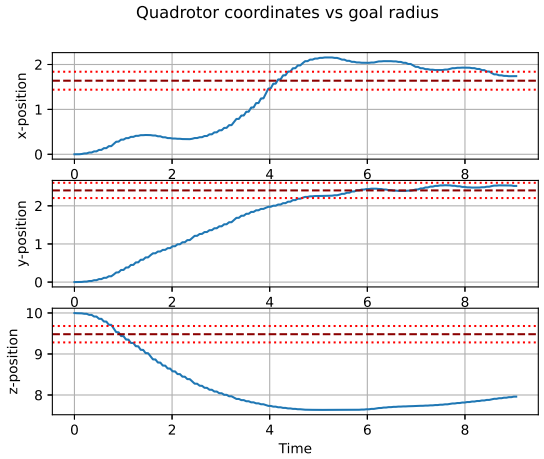


(a) Quadrotor coordinaters vs goal radius

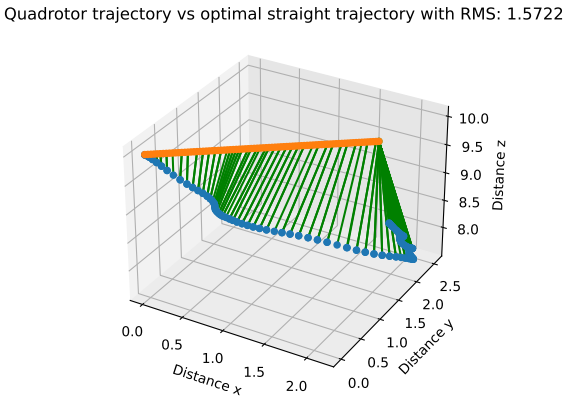


(b) Quadrotor trajectory

Figure 2.9: Example of run from network configuration 3 with +10% modified mass



(a) Quadrotor coordinaters vs goal radius



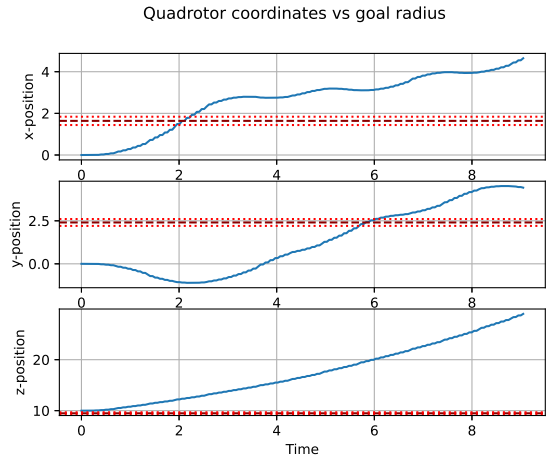
(b) Quadrotor trajectory

Figure 2.10: Example of run from integral network configuration with +10% modified mass

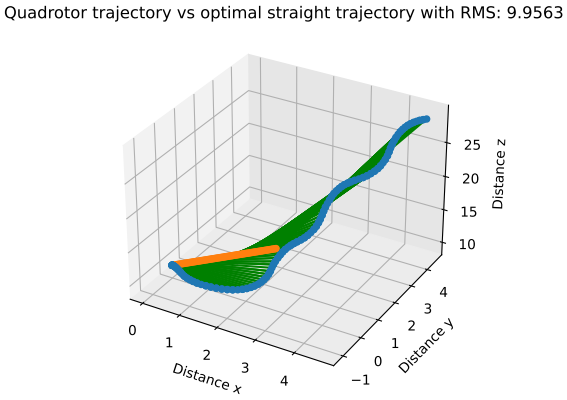
2.2.2 **Results** −10%

	Average RMS	Success rate
Network configuration 1	7.392	0/10
Network configuration 2	3.665	0/10
Network configuration 3	5.659	0/10
Integral configuration	5.531	0/10

Table 2.5: Average RMS values for −10% modified quadrotor weight

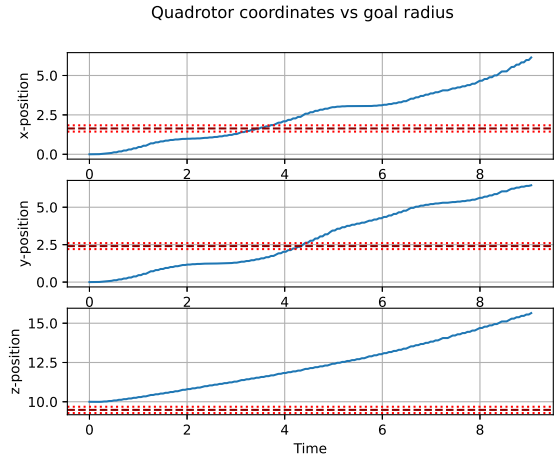


(a) Quadrotor coordinaters vs goal radius

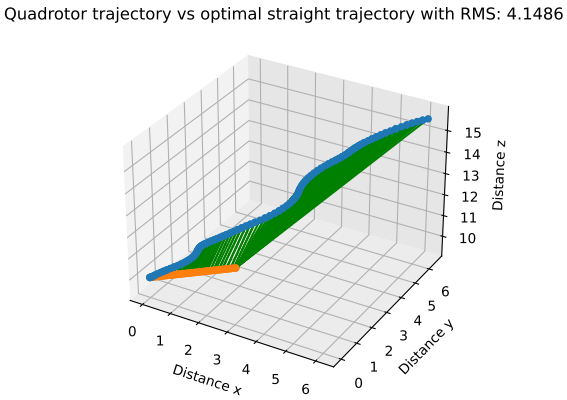


(b) Quadrotor trajectory

Figure 2.11: Example of run from network configuration 1 with −10% modified mass

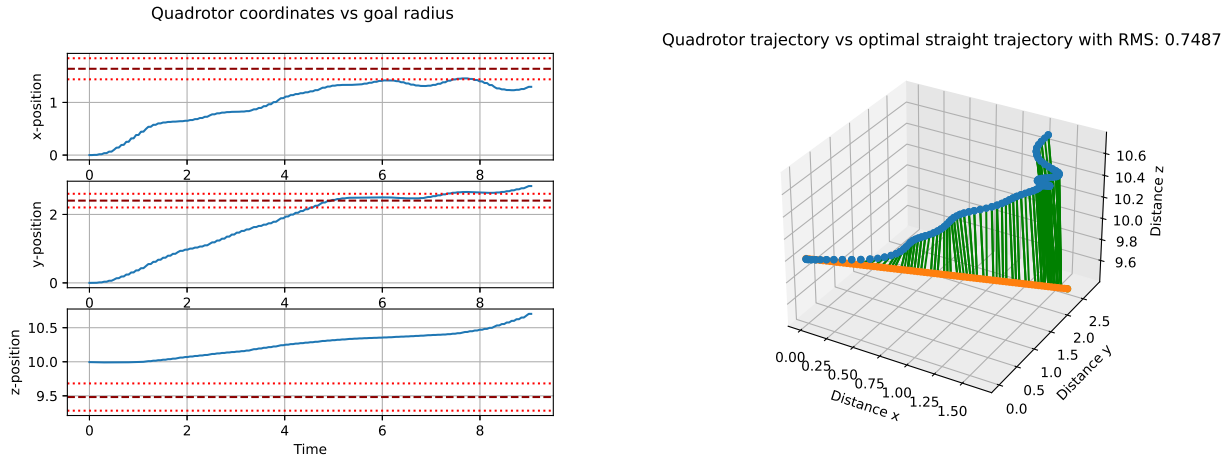


(a) Quadrotor coordinaters vs goal radius



(b) Quadrotor trajectory

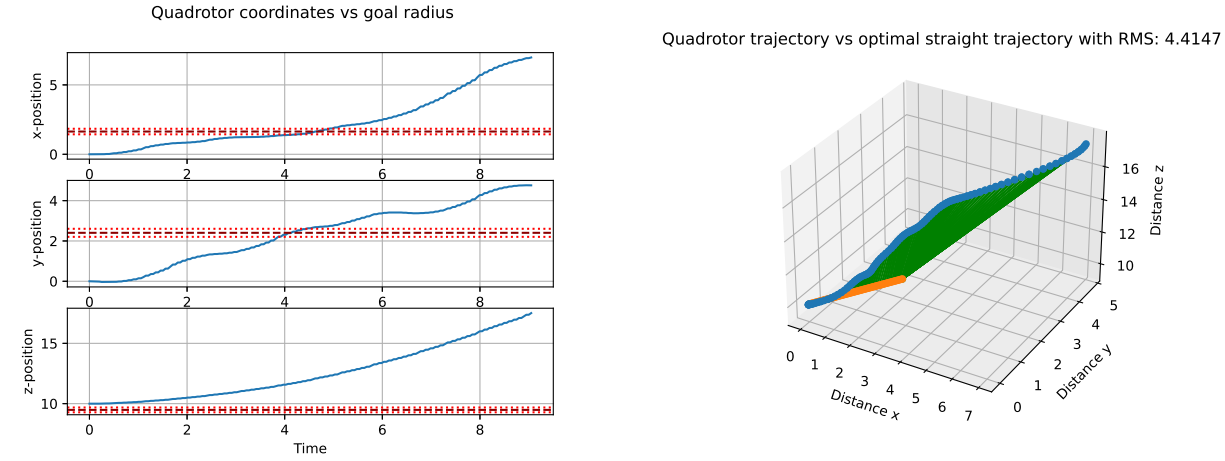
Figure 2.12: Example of run from network configuration 2 with −10% modified mass



(a) Quadrotor coordinaters vs goal radius

(b) Quadrotor trajectory

Figure 2.13: Example of run from network configuration 3 with −10% modified mass



(a) Quadrotor coordinaters vs goal radius

(b) Quadrotor trajectory

Figure 2.14: Example of run from integral network configuration with −10% modified mass

2.3 Discussion

2.3.1 Unmodified mass

Since ANNs are considered black-box algorithms, it is not possible to make very bold claims as to what is happening inside them. Therefore, this report is careful about making any conclusions of the results generated from the ANNs. However, we will discuss the tendencies that we can see from the results, and make modest explanations for why they are that way. They key goal when experimenting with the different network topologies, was to ensure that the networks feature extraction complexity matched the actual policy feature complexity in the training data.

We begin with the results from the different networks when running without the modification in mass.

As we can see from the table 2.3, the network configuration 2 works the best, with an RMS of 0.389 and a 100% success rate on the 10 goal points. If we look at one of the trajectories generated from this network in figure 2.4, we see that the quadrotor quite confidently navigates to the goal, or rather to inside the goal radius. It also stays within the goal radius once it has been reached. However, we do not see a critically damped second order response in the plots which would be the ideal step response for our system. We have to remember however, that this behaviour has been self-taught by the DDPG algorithm with only our reward function as guidance. The low average RMS of 0.389 tells us that the network is able to follow the optimal straight path pretty well, considering that the goal points are situated about 3 meters away most of the time. It is also worth mentioning that the axes in the 3D plot have different step lengths, and one has to keep this in mind when judging the trajectory. When looking at the training progress for this network in figures B.3 and B.4, we see that it converges after about 1750 epochs during the first training, and only needs about 150 epochs more in training session 2 to learn the smaller goal size.

We continue by looking at the results from the first network configuration. With an Average RMS of 2.588, it performs much poorer. This is a high value considering that the goal points are situated about three meters away most of the time. But it does learn some general trends though, as we can see in the plot for the trajectory in figure 2.3. The network understands in which general direction it is supposed to go, but constantly overshoots and is not very good at maintaining a stable hover inside the goal radius. This is consistent with what one may expect from a shallower neural network, as explained more in-depth in section 1.2.2. The deeper the network, the more fine-grained its feature extraction and behavioural learning is, whereas shallower networks only can encode information for the more general trends in the training data. This suggests that this network topology may be too shallow to solve our problem, as its ability to encode complex features lower than the actual policy feature complexity in the training data. Looking at the training progress in figures B.1 and B.2, we can see that the average rollout returns per training epoch converges on much lower values than using network configuration 2, as shown in figures B.3 and B.4. It is also interesting to note that the network converges quicker than its deeper counterpart.

Looking at the network configuration 3, we again see a poorer performance than network configuration 2, as shown in table 2.3. It does have a pretty good success rate at 70%, but with a significantly higher average RMS. Looking at the training progress in figures B.5 and B.6, we see that convergence takes many more steps than the previous network topologies. This is consistent with what we experienced in the other two networks; the deeper the longer before convergence. A pitfall in having a neural network too large, is that we can risk having an *overparametrized function approximator*. This essentially means that we can end up having the network learning to solve the problem of navigation split up into several different subproblems, i.e that it does not generalize properly to a global solution. This means that the network is able to, and tries to, encode more complex features than the policy features in the actual data set, and that the network easily *overfits* on its training data. The symptom of this is that the quadrotor sometimes completely fails and flies off randomly, because it is spawned into a region in state space where the neural network does not have a solution to the given subproblem. With a significantly higher average RMS and a lower success rate than network configuration 2, it seems that this is the case. This also means that for the subregions of state space where the network does have an optimal solution, it may be a very fine-tuned and optimized one as we can observe in the example trajectory 2.5, with a very small RMS error and quick

converge to withing the goal radius.

The integral network configuration being so different than the others, it is difficult to do direct comparisons to the other three networks. The state space is larger, and the reward function is therefore quite different. Therefore the neural network has a harder job of learning how to extract important information from the input. The result is a network with average RMS of 1.536 and a success rate of 50% which is not particularly impressive in comparison to the network configuration 2. The learning progress as seen in figure B.8 shows that the network converges to an optimum quite quickly in the second training session. It may however mean that it is simply more difficult to learn on the basis of the new reward term. If this network would have been trained better, it could be that it offers better results against parameter (e.g., mass) uncertainty.

An important observation is that choosing the correct topology for the networks in RL is more important than in the standard *supervised learning*. In standard *supervised learning*, an overly complex neural network can be taught to generalize by feeding it more independent and identically distributed training samples. However when working with RL, we remember that the training data is generated by the agent itself, and any step in the wrong direction will generate bad training data. This means that divergence during training is a big problem, and we cannot assume that the overly complex neural networks will continue receiving good training data for the longer period of time that they require.

2.3.2 Modified mass +10%

Looking at table 2.4, we can see the performance of the network configurations when the mass of the quadrotor is increased på 10%. None of the network configurations manages to reach the goal properly, but looking at the average RMS we can still get an indication of their performances.

The first thing to notice is that the network configuration 1 now does better than network configuration 2. This is very unexpected. Why would the crude and shallow network outcompete the balanced and accurate deeper network configuration 2? One property of the network configuration 1 that we noted in the previous result section, is that it understood pretty well in which direction the quadrotor should be moving. However, the magnitude of the desired acceleration vector was not changing properly according to the distance away from the target. This resulted in an overshooting behaviour. Now that we have changed the mass of the quadrotor, we have essentially done two things: increased the gravitational pull and decreased the magnitude of the acceleration vector. Surprisingly this results in a lower average RMS than in the case where we had not modified the mass. It seems that decreasing the magnitude of the acceleration vector has resulted in a more stable behaviour. Another thing to note in the figure 2.7, is that the increased gravitational pull does not seem to result in the quadrotor falling below the height of the goal point.

The second thing to notice, is that the integral network seems to be handling this change in mass quite well. The average RMS has only increased from 1.536 to 2.748. An example of this can be seen in 2.10. In this plot we see that the quadrotor is able to converge on the correct position in the x and y-axes, however it struggles with the increased gravitational pull. In comparison, the network configuration 2 increases almost an order of magnitude in its average RMS from 0.389 to 2.918. It seems that having the

integral effect does indeed make the network more robust to uncertainties in the model, as previously suggested.

Looking at the network configuration 2 as exemplified in figure 2.8, we see that it is able to converge somewhat to the goal position in the x and y-axes, but struggles with the increased gravitational pull just like the network configuration 3 and the integral network configuration.

Lastly, the network configuration 3 has an increase in average RMS from 1.141 to 2.337, which is a large value considering that the goals mostly are about 3 meters from where it spawns. The trajectory of this network is exemplified in figure 2.9, where it struggles with converging on the goal in all three axes.

A possible explanation as to why the deeper networks fare worse when changing the mass, is that the generality in the solutions within the networks is decreasing when increasing its depth. It learns more fine-grained solutions, which are then more sensitive to changes in the initial parameters of the quadrotor model. This does not necessarilly mean that using a deeper network cannot properly work for our problem. It may simply mean that it is a matter of further and more robust training. Nonetheless, the key point taken from this work is that we need to identify the network structure that allows us to learn a good policy, while also maintaining as much simplicity as possible.

2.3.3 Modified mass -10%

When decreasing the mass by 10%, we do exactly the opposite of the previous section. The magnitude of the acceleration vector has increased, and the gravitational pull has decreased. We notice straight away that all network configurations diverge in the z-axis, as seen in figures 2.11,2.12,2.13,2.14. The quadrotor suddenly becoming lighter, their altitudes increase rapidly without recovery. It seems that in the z-axis, the networks are much less robust to negative changes in mass, than positive changes as we saw in the previous section 2.3.2. Changing the magnitude of the acceleration vector also has a profound impact of the dynamics of the system. In the previous section 2.3.2 we discussed how decreasing the mass made the dynamics slower, as the acceleration vector got scaled back. Now we have the opposite challenge: the magnitude of the acceleration vector is larger than what the networks intend them to be. We observe the dynamics to be more unstable and consequently the average RMSes to be worse as seen in table 2.5.

Beginning with the network configuration 1, we see a significant degradation in performance as seen in table 2.5. As discussed in the previous section, the network configuration 1 is pretty good at the general direction in which the acceleration vector should point, but not as good at scaling its magnitude properly. In the case of having unmodified mass, the network overshot and it seemed like the magnitude of the acceleration vector was too great, and that increasing the mass made the behaviour of this network behave more stable. It is therefore to be expected that its performance should drop as one decreases the mass. The magnitude of the acceleration vector which is already too great, is increased. As we see in figure 2.11, the quadrotor tries to converge on the x and y-axes, but overshoots when reaching the goal coordinates without recovering.

When looking at the results from network configuration 2 in table 2.5, these results are combarable to the case where the mass is increased as seen in table 2.4, with an average RMS of 3.665. In general, this network suffers from the same problem as the network configuration 1. Making the magnitude of the

acceleration vector greater, results in overshooting in the x and y-axes without recovery, an example of which is seen in figure 2.12. The acceleration vector is pointing in the right direction in the x and y-axes, but is not scaled back as it approaches the goal coordinates making it diverge.

The network configuration 3 also experience a significant drop in performance, as seen in table 2.5. With an average value of 5.659, is considerably worse than the case in which the mass is increased, where it had an average RMS of 2.337. The overshooting behaviour was also present in this network configuration, although it at times did comparably well as seen in figure 2.13. Most of the time however, it did not converge in the x and y-axes.

The integral configuration is only marginally better than the network configuration 3, as seen in table 2.5, and most certainly a lot worse than its performance when increasing the mass as seen in table 2.4. This suggests that having an integral effect when the system dynamics become more unstable as a result of increasing the magnitude of the acceleration is not as helpful as when the opposite is the case. A slowly converging system will certainly gain from having an integral effect, where the integral effect will force a faster convergence. However, when the system is already overshooting, it seems that adding integral effect into the state-space is not as helpful.

2.3.4 Advantages and disadvantages of having a learning-based controller

The biggest challenge when implementing a learning-based controller is that there exists no convergence proofs for neural networks. This is because the neural networks are black boxes; we cannot open them up and understand what happens inside, and the neural networks may as well have reached a local minimum. At least not precisely yet. Similarly, we cannot do any formal consistency analysis either. We can only train and test the network in a broad range of settings and hope that every point in state-space has an optimal policy. This is in contrast to standard controllers, where consistency and stability can be proven mathematically. When training a network, it is fitted to a set of training data which is completely dependent on the parameters of the system with which that data is generated. If a value needs to be changed in the model of the system, the neural networks have to be retrained. And again, there is no proof that all its knowledge about the old model is correctly modified to fit the new one. When using mathematical models, one can change parameters and test for stability straight away. The design choices when one constructs a neural network is largely experience based, and one has to extensively experiment with what depth, width and activation functions works the best for the given problem. And because a neural network may converge to different weights and biases each time it is trained with exactly the same initial conditions, this process is not an exact science. All one has, are guiding principles and past experience. And lastly, for the training of the neural network to really produce good results, one needs many samples. Each network in this project thesis required about 8 million data samples each. One thus depends on a fast and accurate simulator for getting data samples that are good, and realistically represents what the controller will experience in real-life.

All those challenges aside, there truly are some benefits to choosing a learning-based controller. The neural network may learn complex non-linear functions and tendencies that we humans never would have managed to model mathematically, and discover features in the data set no one even knew existed.

These methods can also be implemented and trained directly, without having to do extensive mathematical modelling and to formulate a fitting controller. As long as you have the necessary data, the neural network may learn a mathematical model all by itself. The last reason to consider a learning-based controller, is exactly the reason we are working on one for this project thesis, namely that it can be integrated into an existing deep learning framework where one includes for example computer vision. When writing the master thesis in the next semester, we will integrate a similar quadrotor controller into a problem that involves sensing and environment awareness using deep learning methods. The core idea is to use exteroceptive modalities (e.g., LiDAR or vision) and solve end-to-end the problem of collision-free navigation without necessarily making an explicit online 3D reconstruction of the environment. This has the potential to lead to fast and safe navigation at a fraction of the computational cost currently required to ensure collision-free flight.

In general, when working with low-order dynamical systems, we can derive an accurate mathematical model from which we can make an optimal controller. Therefore, this *path following* problem can be solved explicitly using methods as discussed in 1.7. If it were not for the fact that we will be integrating this into an end-to-end solution using deep learning, it would have been better to use such methods.

Chapter 3

Conclusions and Further Work

3.1 Summary

In this project report we have developed the necessary theoretical background within the fields of machine learning and reinforcement learning to understand the building blocks of modern state-of-the-art deep reinforcement learning methods. Then, a mathematical derivation took us from the basic policy gradient formulation through all the necessary RL concepts until we arrived at the algorithm with which this project report was developed, namely the Deep Deterministic Policy Gradient algorithm. The difference between DDPG and two trust-region methods, TRPO and PPO, were explained for context.

The dynamics for the quadrotor were then presented, together with a thorough explanation of the way the DRL agent is integrated into the control loop. For reference, related work with similar problem formulations were introduced. We then looked at both the experimental setup and the software architecture used to train the neural networks and generate the results. The reward function was introduced and its effect on the training progress explained.

Lastly, the hyperparameters and the method of training the neural networks were explained. To test the networks for robustness, a change of $\pm 10\%$ in mass was introduced. The results from the experiments were presented and discussed.

3.2 Conclusions

The most important conclusions of this project thesis are that the size and shape of the neural network used for learning a good navigation policy **do** have an important impact on the behaviour of the quadrotor. There is a trade-off between generalization and overparametrization, where shallow networks tend to generalize more, and deep networks tend to overparametrize. Striking a balance between these two effects produced satisfactory results. A policy network with two layers, having 64 as width in both, was able to accurately and consistently reach the goal state with a comparatively low average RMS error from the optimal linear path. This means it has the best topology out of all the configurations, matching its ability to learn complex features with the actual policy feature complexity in the data set pretty well. With this, we have provided an example of consistent behaviour using deep reinforcement learning for quadrotor

navigation, which in itself is an interesting result for future work.

When testing for robustness, we began increasing the mass by 10%. This worsened the performance of the network configurations 2 and 3, as the magnitude of the acceleration vector suddenly changed unexpectedly. These acceleration vectors were finely tuned, and thus the resulting performance when increasing the mass worsened. Surprisingly, the network configuration 1 improved. The agent had understood in which general direction it was supposed to go, but its acceleration vector was too aggressive. Increasing the mass and thus decreasing the magnitude of the acceleration vector, made the behaviour of this network configuration more stable. It was also interesting to observe that the deterioration in performance of the integral network when increasing the mass, was not as significant. This suggests that there may be some advantage to having integral effect when the system dynamics are made slower.

Interestingly, it seems that all network configurations were much less robust to decreasing the mass by 10% compared to the case where it was increased. Since the gravitational pull is decreased, all network configurations experienced rapid divergence in the z-axis, namely the height. The second effect of decreasing the mass, is a corresponding increase in magnitude of the acceleration vector. This made the behaviour of all networks deteriorate considerably. Their behaviour showed that they commanded the quadrotor in the right direction in the x and y-axes, but overshot without recovery within the time limit.

3.3 Further Work

Ideally, the goal should have had a much smaller radius. The radius of acceptance was in this project kept at a distance of 0.2m, as decreasing this radius proved to be a very computationally expensive endevour as one had to run for a long time before the networks converged within the new radius. If one were to perfect the results from the networks presented in this report, this radius should have been incrementally decreased even further.

At the same time, we used a constant learning rate when training the networks. Ideally, this should be adjusted to ensure monotonic decreasing steps, which is exactly what trust-region methods like TRPO and PPO do. If one was to continue improving the DDPG agents, one would choose the network weights from the highest scoring epoch, and training this further with a smaller learning rate.

This project report only deals with having one goal, but in a larger context it would have been interesting exploring having these networks track a trajectory with several waypoints.

Appendix A

Acronyms

UAV Unmanned Aerial Vehicle

RL Reinforcement Learning

DRL Deep Reinforcement Learning

DDPG Deep Deterministic Policy Gradient

TRPO Trust-region policy optimization

PPO Proximal policy optimization

SLAM Simultaneous Localization and Mapping

PID Proportional-Integral-Derivative controller

ANN Artificial Neural Network

SGD Stochastic Gradient Descent

MDP Markov decision process

Appendix B

Algorithms, training progress and computer specifications

B.1 Algorithms

Algorithm 1 One-step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parametrization $\pi(a|s,\theta)$
Input: a differentiable state-value function parametrization $\hat{v}(s,w)$
Parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to 0)
loop forever (for each episode):
    Initialize S (first state of episode)
    $I \leftarrow 1$
    loop while S is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\theta)$
        Take action A, observe S',R
        $\delta \leftarrow R + \gamma \hat{v}(S',w) - \hat{v}(S,w)$ (if S' is terminal, then $\hat{v}(S',w) \doteq 0$)
        $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S,w)$
        $\theta \leftarrow \alpha^\theta I \delta \nabla ln\big(\pi(A|S,\theta)\big)$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$

Algorithm 2 Deep Deterministic Policy Gradient (DDPG)

Randomly initialize critic network $Q_c(s,a|\theta^Q)$ and actor network $\mu(s|\theta^\mu)$ with weights θ^{Qc} and θ^μ
Initialize target network Qc' and μ' with weights $\theta^{Qc'} \leftarrow \theta^{Qc}$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
Initialize a random process $\mathcal{N}$ for action exploration
Receive initial observation state s_1
for *n = 1, Number of epochs* **do**
    for *C = 1, Number of cycles* **do**
        for *p = 1, Number of Rollouts* **do**
            Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise.
            Execute action a_t and observe reward r_t and observe new state s_{t+1}
            Store transition (s_t, a_t, r_t, s_{t+1}) in R
            if Episode ends **then**
                Restart a new episode
                Reinitialize noise $\mathcal{N}$ for action exploration
        for *K = 1, Number of trainsteps* **do**
            Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) from R
            Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
            Update critic by minimizing the loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^{Qc}))^2$
            Update the actor policy using the sampled policy gradient:
            $\Delta_{\theta^\mu} J \approx \frac{1}{N}\sum_i \Delta_a Q_c(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \Delta_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
            Update the target networks:
            $\theta^{Qc'} = \tau\theta^{Qc} + (1-\tau)\theta^{Qc'}$
            $\theta^{\mu'} = \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

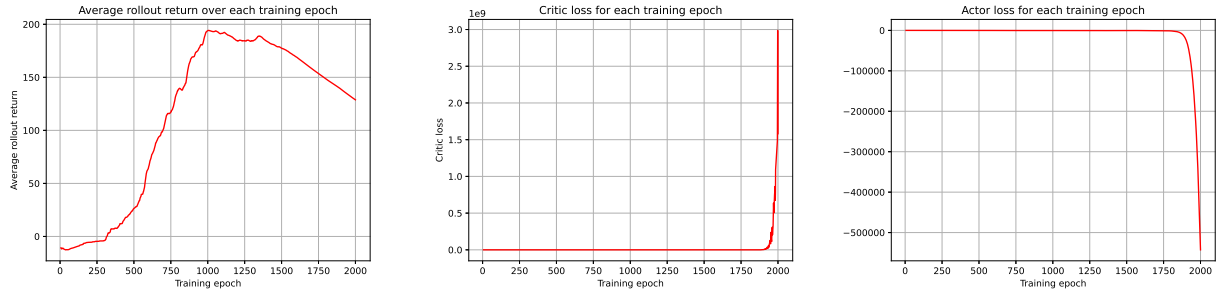
Algorithm 3 Trust-region Policy Optimization

Input: initial policy parameters θ_0 and initial value function parameters ϕ_0
Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K
for k = 0,1,2,... **do**
    Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment
    Compute rewads-to-go $\hat{R}_t$
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{π_k}
    Estimate policy gradient as: $\hat{g}_k = \frac{1}{|D_k|}\sum_{\tau\in D_k}\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)|_{\theta_k}\hat{A}_t$
    Use the conjugate gradient algorithm to compute $\hat{x}_k \approx \hat{H}_k^{-1}\hat{g}_k$, where $\hat{H}_k$ is the Hessian of the sample average KL-divergence
    Update the policy by backtracking line search with: $\theta_{k+1} = \theta_k + \alpha^j\sqrt{\frac{2\delta}{\hat{x}_k^T\hat{H}_k\hat{x}_k}}\hat{x}_k$, where $j \in \{0,1,2,...,K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint
    Fit value function by regression on mean-squared error: $\phi_{k+1} = \arg\min_\phi \frac{1}{|D_k|T}\sum_{\tau\in D_k}\sum_{t=0}^{T}\left(V_\phi(s_t) - \hat{R}_t\right)^2$, typically via some gradient descent algorithm

Algorithm 4 Proximal Policy Optimization, Actor-Critic style

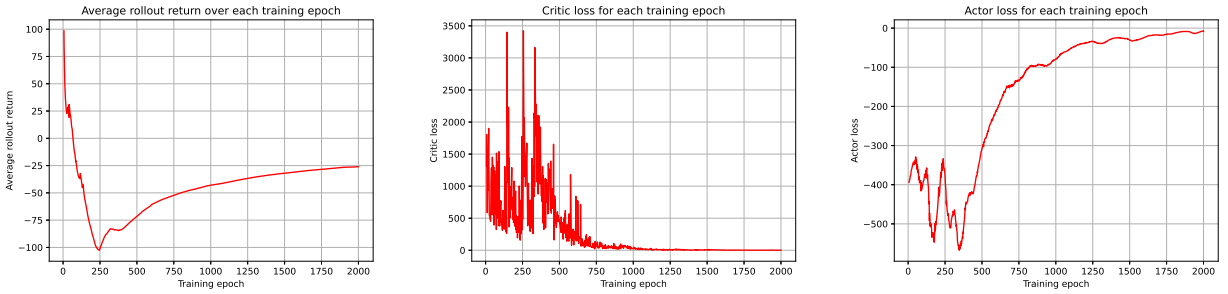
for iteration = 1,2,... **do**
  for iteration = 1,2,...,N **do**
    Run policy π_{old} in environment for T timesteps
    Compute advantage estimates $\hat{A}_t, ..., \hat{A}_T$
  Optimize surrogate L w.r.t θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$

B.2 Training progress



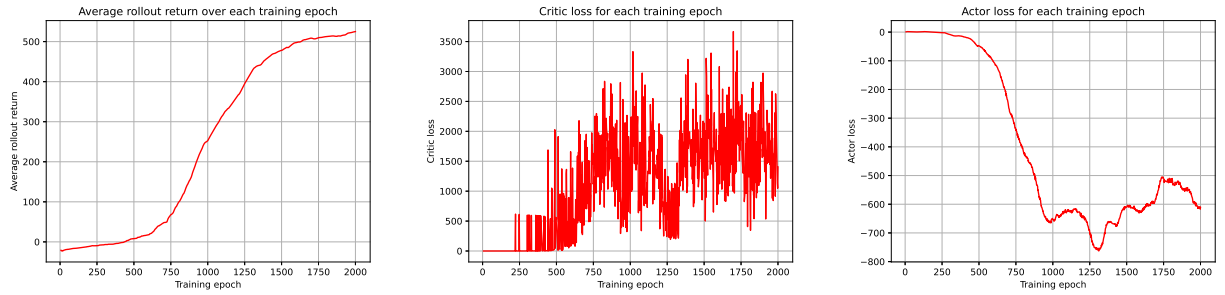
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.1: Training session 1 for network configuration 1



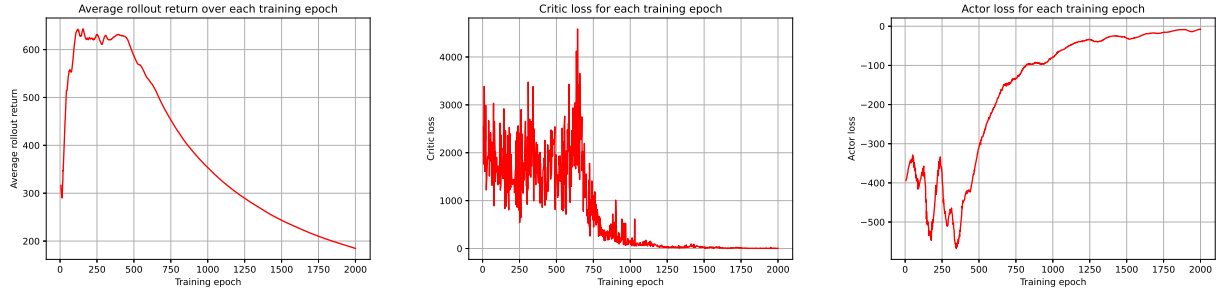
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.2: Training session 2 for network configuration 1



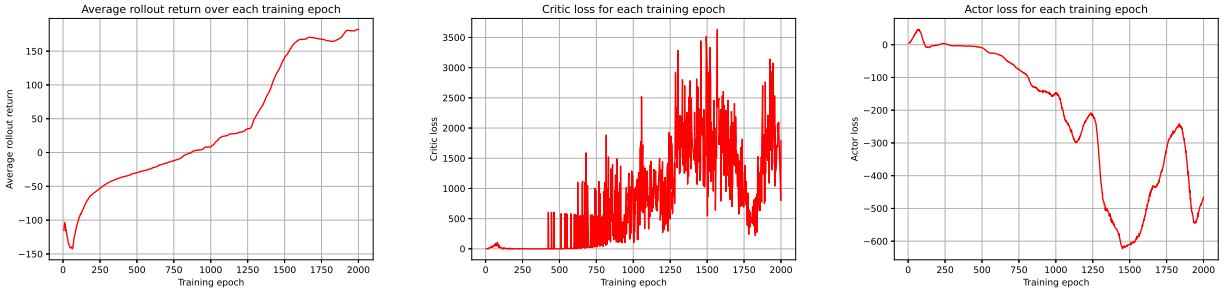
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.3: Training session 1 for network configuration 2



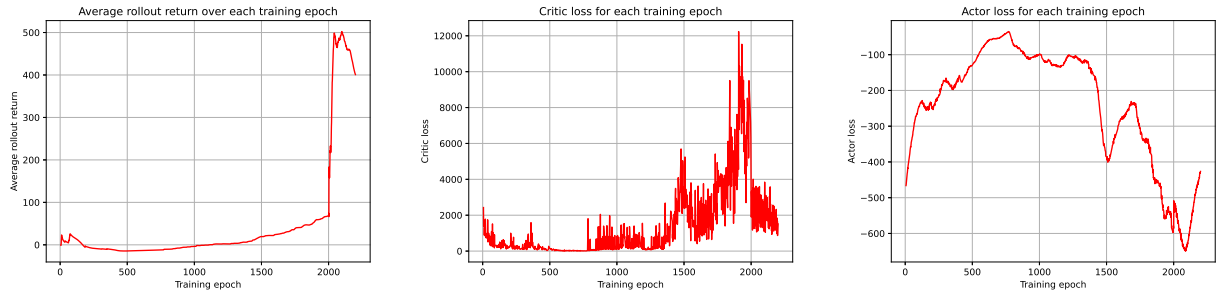
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.4: Training session 2 for network configuration 2



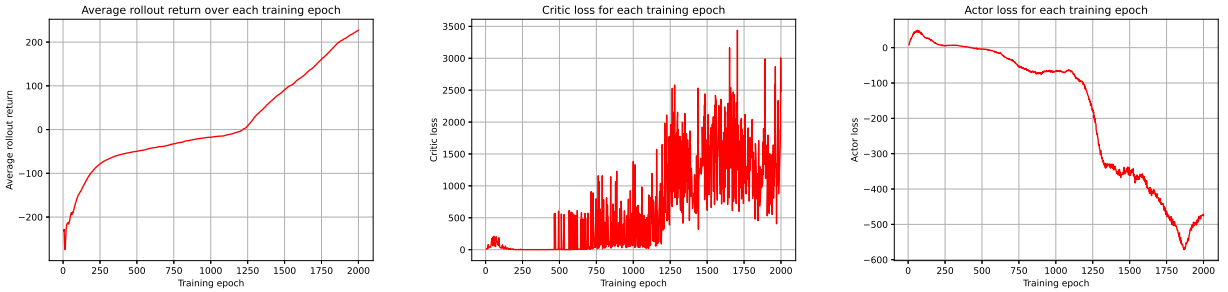
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.5: Training session 1 for network configuration 3



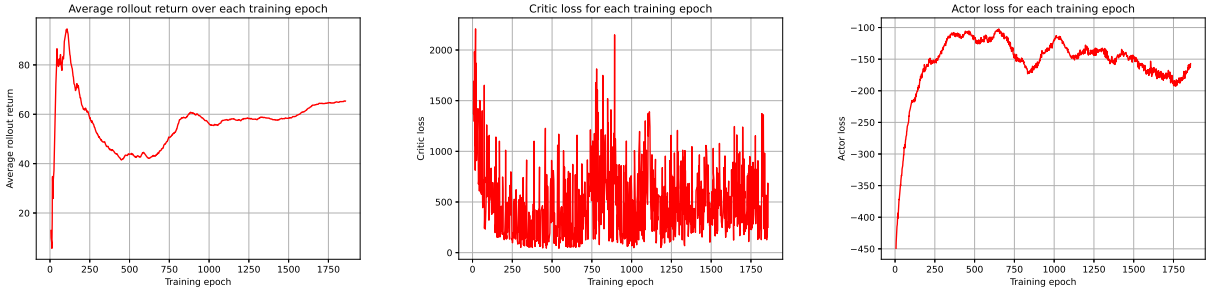
(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.6: Training session 2 for network configuration 3



(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.7: Training session 1 for integral network configuration



(a) Average rollout return per epoch (b) Average Critic loss per epoch (c) Average Actor loss per epoch

Figure B.8: Training session 2 for integral network configuration

B.3 Computer specifications

The results for this project was computed on a Dell 9570 XPS 15 laptop with the following components:

- 8th Generation Intel Core i7-8750H Processor
- NVIDIA(R) GeForce(R) GTX 1050Ti with 4GB GDDR5
- 16GB, 2x8GB, DDR4, 2666MHz

Bibliography

[1] Martın Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Adeel Akhtar, Steven L Waslander, and Christopher Nielsen. “Fault tolerant path following for a quadrotor”. In: *52nd IEEE Conference on Decision and Control*. IEEE. 2013, pp. 847–852.

[3] *Autonomous robots lab*. URL: https://www.autonomousrobotslab.com/.

[4] D. Cabecinhas, R. Cunha, and C. Silvestre. “A Globally Stabilizing Path Following Controller for Rotorcraft With Wind Disturbance Rejection”. In: *IEEE Transactions on Control Systems Technology* 23.2 (2015), pp. 708–714. DOI: 10.1109/TCST.2014.2326820.

[5] Yang Chen et al. “Planar smooth path guidance law for a small unmanned aerial vehicle with parameter tuned by fuzzy logic”. In: *Journal of Control Science and Engineering* 2017 (2017).

[6] Venanzio Cichella et al. “A 3D Path-Following Approach for a Multirotor UAV on SO(3)”. In: *IFAC Proceedings Volumes* 46.30 (2013). 2nd IFAC Workshop on Research, Education and Development of Unmanned Aerial Systems, pp. 13–18. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20131120-3-FR-4045.00039. URL: http://www.sciencedirect.com/science/article/pii/S1474667015402654.

[7] *DARPA Subterranean challenge*. URL: https://www.subtchallenge.com/.

[8] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[9] Fadri Furrer et al. “RotorS—A modular gazebo MAV simulator framework”. In: *Robot Operating System (ROS)*. Springer, 2016, pp. 595–625.

[10] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[12] J. Hwangbo et al. “Control of a Quadrotor With Reinforcement Learning”. In: *IEEE Robotics and Automation Letters* 2.4 (2017), pp. 2096–2103. DOI: 10.1109/LRA.2017.2720851.

[13] Hyeong Ryeol Kam et al. “Rviz: a toolkit for real domain data visualization”. In: *Telecommunication Systems* 60.2 (2015), pp. 337–345.

[14] Nathan Koenig and Andrew Howard. “Design and use paradigms for gazebo, an open-source multi-robot simulator”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. IEEE. 2004, pp. 2149–2154.

[15] S. Kukreti, M. Kumar, and K. Cohen. “Genetically tuned LQR based path following for UAVs under wind disturbance”. In: *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2016, pp. 267–274. DOI: 10.1109/ICUAS.2016.7502620.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[17] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[18] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: *Master’s Thesis (in Finnish), Univ. Helsinki* (1970), pp. 6–7.

[19] José Luis Mendoza-Soto, José J Corona-Sánchez, and Hugo Rodrıguez-Cortés. “Quadcopter path following control. a maneuvering approach”. In: *Journal of Intelligent & Robotic Systems* 93.1-2 (2019), pp. 73–84.

[20] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[21] Jorge J Moré. “The Levenberg-Marquardt algorithm: implementation and theory”. In: *Numerical analysis*. Springer, 1978, pp. 105–116.

[22] P. Niermeyer et al. “Geometric path following control for multirotor vehicles using nonlinear model predictive control and 3D spline paths”. In: *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2016, pp. 126–134. DOI: 10.1109/ICUAS.2016.7502541.

[23] *Proximal Policy Optimization*. URL: https://openai.com/blog/openai-baselines-ppo/.

[24] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[25] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[26] Ashton Roza and Manfredi Maggiore. “Path following controller for a quadrotor helicopter”. In: *2012 American Control Conference (ACC)*. IEEE. 2012, pp. 4655–4660.

[27] B. Rubí, B. Morcego, and R. Pérez. “A Deep Reinforcement Learning Approach for Path Following on a Quadrotor”. In: *2020 European Control Conference (ECC)*. 2020, pp. 1092–1098. DOI: 10.23919/ECC51009.2020.9143591.

[28] Sanjay P Sane. “Neurobiology and biomechanics of flight in miniature insects”. In: *Current opinion in neurobiology* 41 (2016), pp. 158–166.

[29] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[30] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[31] Sagar Sharma. "Activation functions in neural networks". In: *International Journal of Engineering Applied Sciences and Technology* 4 (2017).

[32] S. Stevšić et al. "Sample Efficient Learning of Path Following and Obstacle Avoidance Behavior for Quadrotors". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3852–3859. DOI: 10.1109/LRA.2018.2856922.

[33] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[34] C Van Der Malsburg. "Frank Rosenblatt: principles of neurodynamics: perceptrons and the theory of brain mechanisms". In: *Brain theory*. Springer, 1986, pp. 245–248.

[35] W. Van Loock et al. "Optimal Path Following for Differentially Flat Robotic Systems Through a Geometric Problem Formulation". In: *IEEE Transactions on Robotics* 30.4 (2014), pp. 980–985. DOI: 10.1109/TRO.2014.2305493.

[36] Y. Wang et al. "Deterministic Policy Gradient With Integral Compensator for Robust Quadrotor Control". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.10 (2020), pp. 3713–3725. DOI: 10.1109/TSMC.2018.2884725.

[37] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.