Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Lasse Hansen Henriksen

# Hovering Control of a Quadrotor Using Monocular Images and Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2019

NTNU
Norwegian University of
Science and Technology

Lasse Hansen Henriksen

# Hovering Control of a Quadrotor Using Monocular Images and Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

# Preface

This master thesis was inspired by the impressive progress that reinforcement learning (RL) and then especially deep reinforcement learning (DRL) have seen in the last few years. The successful applications of DRL in the field of robotics and control have mainly been focusing on how these algorithms are able to solve complex problems that are typically hard to formulate via the classical control theory approach. One drawback with the current state of the art DRL however, is that these algorithms typically require an awful lot of data and computational power to find good solutions. Not much research has been conducted regarding the capabilities of DRL to act optimally under imperfect information either seeing as most displays of DRL has been done in simulations where ground truth state variables are available.

One area of robotics where DRL is viewed as an interesting alternative to control is UAVs, and more specifically quadrotors. Motivated by both the ups and downs of DRL, I started to explore the possibility of forming a solution to the 3-D hovering problem for quadrotors using only camera images. To dispute the common — but fair — presumption that was DRL requires too much data to be applicable to robots in the real world I sought a solution that unites the best of both worlds — the DRL and classical control theory approach. This idea is what transpired the work presented in this thesis.

Even though the quadrotor problem is the topic of study in this thesis the contributions are of general value and show how DRL can be used together with a classical control theory approach for acting under imperfect information while being feasible to apply in real-world applications.

This thesis was supervised by Anastasios Lekkas from the Department of Engineering Cybernetics at Norwegian University of Science and Technology (NTNU) and would not have been possible without his valuable input. This masters' thesis is not a direct continuation of the project thesis which similarly studied a quadrotor control problem. However, this thesis utilizes prior

knowledge and experience gained with DRL when working with the project thesis, and the RL theory presented here is based on the project thesis. The DRL algorithm used was also implemented (by myself) during the project thesis work. The project thesis has therefore been included as an attachment for the evaluators of this work.

Lastly, I want to acknowledge the other contributions that made this project possible:

- the open source software that was used, and which without would have made this project impossible. This includes, but is not limited to, Tensorflow, Numpy, Matplotlib, OpenCV, Gazebo and ROS.
- the provision of a quadrotor, extra batteries, and access to a laboratory to conduct tests, by the institute.

*June 24, 2019*
*Lasse Hansen Henriksen*

# Abstract

In recent years the field of reinforcement learning has experienced a renaissance due to breakthroughs in the field itself, and because of other enabling technologies such as efficient and powerful software frameworks, computational power, and especially research on artificial neural networks. Using reinforcement learning in conjunction with neural networks has evolved to be the rule rather than the exception, and multiple examples exist that showcases its impressive problem-solving capabilities as a general framework for solving complex problems. Because of how general the framework is it can be applied to a large variety of problems including the field of robotics and control.

One such problem of interest is control of quadrotors, which similarly has seen a resurgence of interest because of its applicability to a multitude of tasks including surveillance, transport, and commercial delivery services. Even though the control of quadrotors is a well-studied problem in the literature it remains a hard one due to its underactuated nature, the difficulty of modelling forces related to aerodynamics, and navigating in GPS denied environments.

In this thesis, the 3-D hovering problem for quadrotors is studied and a controller design that attempts to unite approaches from reinforcement learning, computer vision, and traditional control theory is presented. By applying the proposed controller to a simulated environment we show that the controller is able to solve the problem by using ground truth state variables and by using indicative state variables based on a simple computer vision algorithm. Furthermore, the simulated-tested controller is transferred to a real-world quadrotor system where the controller is tested on the same problem. We then demonstrate how the controller generalizes with respect to what it has learned in simulation, and how training the controller to adapt to the real world can be achieved through a feasible and practical method. Lastly, the controllers performance post real-world training is demonstrated.

# Sammendrag

I de siste årene har fagfeltet forsterkendelæring opplevd en renessanse på grunn av gjennombrudd i feltet selv, og på grunn av andre muliggjørende teknologier slik som effektive software verktøy, beregningskraft og særlig forskning på kunstige nevrale nettverk. Bruk av nevrale nettverk sammen med forstærkende læring har utviklet seg til å være regelen i stedet for unntaket, og det finnes flere eksempler som viser denne teknologiens imponerende problemløsningsevner som et generelt rammeverk for å løse komplekse problemer. På grunn av hvor generelt dette rammeverket er kan det brukes på et stort utvalg av problemer, inkludert robotteknikk og kontroll.

Et problem innenfor robotteknikk av interesse er kontroll av quadcoptere, som på samme måte har sett en gjenoppblomstring av interesse på grunn av dets anvendelighet til en rekke oppgaver inkludert overvåking, transport og kommersielle leveringstjenester. Selv om kontroll av quadcopters er et godt studert problem i litteraturen, er det fortsatt vanskelig på grunn av dets underaktuerte natur, vanskeligheten ved å modellere krefter relatert til aerodynamikk, og navigering i områder uten GPS dekning.

I denne oppgaven studeres oppgaven med å holde et quadcopter svevende i et gitt 3-dimensjonalt referanse punkt, og ett kontroller design som forsøker å forene tilnærminger fra forsterkende læring, datasyn og tradisjonell kontrollteori presenteres. Ved å bruke denne kontrolleren i et simulert miljø viser vi at styreenheten er i stand til å løse problemet både ved bruk av de sanne posisjoneringsvariablene og ved hjelp av indikative tilstandsvariabler basert på en enkel datasyn algoritme. Videre overføres kontrolleren som er blitt testet i simulering til et virkelig quadcopter-system hvor kontrolleren testes på samme problem. Vi demonstrerer så hvordan kontrolleren generaliserer med hensyn til hva den har lært i simulering, og hvordan trening av kontrolleren for å tilpasse seg den virkelige verden kan oppnås gjennom en praktisk metode. Til sist vises kontrollerens evne til å kontrollere quadcopteret etter at den har fått trent i den

virkelige verden.

# Contents

# List of Figures

# Acronyms

| | | |
|---|---|---|
| MDP | = | Markov decision process |
| POMDP | = | Partially observable markov decision process |
| MC | = | Monte-carlo |
| TD | = | Temporal difference |
| RL | = | Reinforcement learning |
| DRL | = | Deep reinforcement learning |
| MBRL | = | Model based reinforcement learning |
| GPS | = | Global positioning system |
| UAV | = | Unmanned aerial vehicle |
| FOV | = | Field of view |
| DDPG | = | Deep deterministic policy gradient |
| DQN | = | Deep Q-Learning |
| ANN | = | Artificial neural network |
| ODE | = | Open Dynamics Engine |
| URDF | = | Universal Robot Description Format |
| ROS | = | Robot Operating System |
| PID | = | Proportional-integral-derivative |
| PPO | = | Proximal policy gradient |
| TD3 | = | Twin Delayed Deep Deterministic policy gradient |
| MPC | = | Model predictive control |

Chapter 1

# Introduction

## 1.1 Background and motivation

While reinforcement learning (RL) first saw light in the 1950s, it has recently caught new attention as deep reinforcement learning (DRL) has emerged promising a broad range of application areas. In addition to breakthroughs on the field itself, its rise has been heavily enabled by other technologies. This includes the increasing computational abilities of silicon-based computers and their availability, research on artificial neural networks (ANNs) that has been especially driven by the ImageNet challenge [2], and software frameworks facilitating efficient computation of – among other things – realistic simulations and the aforementioned ANNs.

Using ANNs as function approximators in conjunction with reinforcement learning has been such a good fit that it has even been been deserving of its own name – deep reinforcement learning (DRL). Neural networks have empowered reinforcement learning to work with tasks that inherit a continuous state space with either a discrete or a continuous action space, and even so, has shown promising results on tasks that are typically hard to formulate using traditional methods requiring specifically tailored, ad-hoc solutions.

Especially two novel algorithms is owed much credit for the recent advances of applying reinforcement learning, namely Deep Q-Networks (DQN) [3] and Deep Deterministic Policy Gradient (DDPG) [4]. DQN was the first

algorithm to solve complex tasks that have continuous state space and introduced key advancements like target-networks and the replay buffer. DDPG was inspired from the former and notes the first successful attempt at integrating neural networks to solve continuous action space problems, which is fundamental to tackling most problems in robotics.

Some general examples of successful applications includes abstract tasks like robotic manipulator grasping and stacking of Lego using monocular images [5], navigation in complex environments [6], and playing various games like Doom [7], Chess and Shogi [8], Go [9], all the Atari games [10] and Starcraft 2 [11].

Many of these examples show the importance of simulators as they depend on them, or are in a sense simulators themselves, in order to gather enough training data to be able to converge to a satisfactory results. The combined computational power that was required for simulation and training in these and other successful applications are not typically available to the average user. In fact, many of these examples required tremendous computational effort to achieve their results and would require years of training in the real world to achieve the same results. It is therefore often infeasible to apply them in practice. Training RL agents in the real world is also a question of safety because optimal behaviour is only guaranteed in the long-run, and such trial and error may cause potential damage to the environment the agent operates in. This, other problems, and the potential side effects concerning safety in artificial intelligence (AI) systems are reviewed in [12].

A significant part of the problem stems from the fact that the ANNs used are difficult optimization problems that are inefficient to solve, but reinforcement learning has also shown to be inept when it comes to sample efficiency. Therefore large amounts of research goes into improving the sample efficiency when training the networks. This includes research on optimizers like Adam [13], picking the most important transitions experienced by the agents for training [14], learning to learn from sparse rewards [15], and learning to learn via meta-learning [16][17] just to name a few relevant research topics.

This problem is further amplified considering that many of the aforementioned success stories learns directly from pixels which leads to a large amount

of parameters to optimize for in a dauntingly vast search space. Even if using pixel values directly as inputs is not always necessary or the best solution depending on the task and the state feedback available, it is often the most convenient way to tackle a problem. The reason for this is diverse, for example camera images provide large amounts of information, does not require extra steps for state estimation because camera sensors do not drift, and camera images are flexible with respect to feature extraction. One approach for reducing the large state space introduced by images is to use auto-encoders to form a lower dimensional state representation. This has been applied with success [18][19], but adds another ANN to the loop that has to be trained.

One interesting problem in robotics where DRL can be applied is quadrotor control. This is a well studied problem, and control is typically achieved by modelling the equations of motion for the aircraft [20]. Yet, this problem still remains a challenging one due to the complexity of aerodynamics modelling, and problems that arise with respect to estimating position when attempting to navigate in indoor environments that lacks coverage of the global positioning system (GPS). For indoor navigation position control is popularly achieved by cameras and pose estimation techniques from computer vision theory by either having a camera attached to the drone itself [21][22], or viewing the drone from a stationary position [23][24] in the environment. Methods using simultaneous localization and mapping (SLAM) are also used [25][26][27], and are arguably more flexible because they can operate without relying on artificial landmarks, but are in return more computationally expensive.

## 1.2 Thesis objective

The problem studied in this thesis is hovering of a quadrotor, otherwise known as dynamic positioning. Motivated by the adversity of training deep ANNs that learns directly from pixels it is therefore interesting to explore the possibility of using low-level features extracted from pictures, in this case image positions, by using traditional computer vision methods. In addition, because control of quadrotors is a well-studied topic and motion controllers already exists, this thesis seeks to construct a reinforcement learning controller that builds upon

an existing low-level velocity controller as opposed to learning to control a quadrotor from scratch.

Being able to utilize purely low-level extraction methods from computer vision to do control is a desirable solution. It would represent a method that rather of moving in the direction of more advanced computer vision methods, for example SLAM as opposed to camera pose estimation, instead seeks to move in the direction of a more sophisticated control algorithm able to act under limited information. This could possibly yield a solution that is more light weight, as opposed to SLAM which can be rather computationally expensive. Since the reinforcement learning framework is able to act optimal even in the case of imperfect– and uncertain information, it is reasonable to think that this may be a viable solution.

Learning raw motor control inputs would be significantly harder considering that quadrotors are heavily underactuated, and experience tells us that it would therefore be unrealistic to train such an agent to be able to perform well within a practical time frame in the real world. In addition, most real systems do not allow you to do the low-level control and features a velocity controller instead. It is also reasonable to assume that the dynamics of a velocity controller for a simulated drone to be more accurate than the actual dynamics of a simulated drone compared to a real world replica. Therefore we expect that the agent trained in a simulation to be able to transfer what it has learn to control the real drone better and reduce real-world training time, which can be tedious because of safety concerns and time constraints. In this regard using a velocity controller is also advantageous as we are guaranteed that the drone will not flip over and fly with maximum velocity towards the ground and break immediately after it is released in the real world for the first time. Lastly, it is interesting to explore DRL as an alternative to quadrotor control as it is a model-free approach and quadrotor dynamics are notoriously hard to model because of aerodynamics that can vary significantly during operation.

To summarize, the goal of this thesis is to develop a position controller for quadrotors based on DRL by using an existing low-level velocity controller and monocular images to reason about position. In the end, this thesis aims to make the following contributions:

1. Explore the possibilities, and design a positioning controller to solve quadrotor hovering control problem via the reinforcement learning framework and simple monocular images.

2. Analyze the accuracy and potential of the resulting controller with respect to operating under external disturbances.

3. Examine if the position controller is feasible to apply to a real-world system within a reasonable time frame such that it is both a practical– and a viable option to existing solutions that does not have to depend on training in simulations for considerable amounts of time.

## 1.3 Related work

Because of the novelty of DRL algorithms it does not exist a plethora of research that attempts to embed them into quadrotor– control problems and applications. And to the authors knowledge there does not exist any published work based on DRL that attempts to tackle the quadrotor hovering problem by using only image position features as proposed in this thesis. While none of the research papers mentioned in this sections deal specifically with the hovering problem, some of them solve it indirectly because they are in essence general position controllers. The examples given below represents some of the applications of DRL to quadrotor control and showcases a few issues related to the use of DRL as well.

A model-based reinforcement learning (MBRL) approach was used in [28] to control the pitch and roll angles of a drone for hovering and shows that while starting from scratch the agent is able to make quick improvements. This work however does not display a fully working controller solution, only the current capabilities of MBRL.

In [29] and [30] DRL agents were trained for quadrotor control to follow trajectories, and showcases controllers that can outperform the traditional model predictive control (MPC) approach in accuracy while being two orders of magnitude less expensive with respect to computational cost.

There are two notable works that tackle the quadrotor landing problem.

The first approach [31] applies a divide and conquer strategy using two DQN networks to learn from low resolution images to first detect a marker and then do vertical descent. The other approach [32] uses DDPG to learn the reference velocities to a velocity controller using the full orientation of the drone as state representation consisting of the position and velocity relative to a moving platform as well as a pressure sensor to register successful landing attempts. None of this methods deal with state estimation, and requires hundred of thousands of training steps to converge to satisfactory solutions.

In [33] a velocity controller for a quadrotor based on DRL that outperformed a well-tuned proportional-integral-derivative controller (PID). The controller policy was trained within roughly 30,000 training steps, which is surprisingly fast. This approached used the full 6-DOF velocity state from simulation and did not deal with state estimation.

The work presented in [34] showcases a position controller based on Q-learning and PID control that manages to stabilize the position of a drone horizontally. The accuracy however is limited because the state space is divided into a discrete grid-space. Even though this work is not DRL, it showcases how RL can be used for hovering for a quadrotor. This approach does not deal with state estimation either.

In fact, none of these research examples deals with state estimation and uncertainty. This is often because the controllers are only displayed in simulation where one in practice are omniscient with respect to state variables. It seems that the overwhelming work that has been conducted on DRL for quadrotor applications so far has been focusing on showcasing the impressive reasoning power of the algorithms — given perfect information. Another reoccurring theme is that the controllers often requires training steps in the range of $10^5 - 10^6$ in order to converge to good control policies. In this thesis we therefore hope to shed some light on both the issue of acting under uncertain information and the issue of training tempo.

## 1.4 Outline of report

In Chapter 2 the theory and background that was used in the design of the controller is presented along with justification for why the particular methods were chosen. Chapter 3 assembles the theory from Chapter 2 to design a positioning controller and the specifics of the design and implementation are discussed. The proposed controller is then tested in a simulated environment in Chapter 4 and results displaying the performance of the quadrotor is presented and discussed. In Chapter 5 the controller that was trained in simulation is applied to the real world by adapting the method used in simulation to train further and to allow it to adapt to the real world, and then the performance of the controller post training is presented and discussed. Chapter 6 discusses different paths that is of interest to explore in the future, and finally, Chapter 7 concludes the thesis.

Chapter 2

# Background and theory

## 2.1 Reinforcement learning

This chapter introduces the basic principles and theory that reinforcement learning is built on, and then explores the necessary prerequisites to understand the more complex state-of-art reinforcement learning algorithms while the choice of algorithm for the particular problem in this project is justified. The content discussed in these sections is heavily based on the book "Reinforcement Learning: An Introduction" by Sutton and Barto [35].

### 2.1.1 The fundamental agent

In reinforcement learning (RL) the term *agent* is used to describe an entity which act towards achieving a goal. For the agent there are three fundamental signals at each timestep of its existence: the observation $o_t$ which is used to construct the agent internal representation of the world – its current state $s_t$ – in the environment, a reward $r_t$ based on the agents current state and the actions performed up until then, and the action $a_t$ that the agent applies to the environment. These signals represents the interactions between the general RL agent and its environment. Figure 2.1 encapsulates these relationships.

In the literature it is common to find that the observation signal $o_t$ is omitted entirely, in which case we have the direct relationship $s_t = o_t$. It is how-

ever useful in some cases to distinguish between the two signals in applications where the raw sensory output is not used directly but instead passed through a processing unit that builds the agents representation of the world from the observation. An example of this is when the state of the system can only be inferred by extrapolation of past observations, or when it is desirable to reduce the dimensionality of the state vector itself. This is an emphasis on the fact that agents in the real physical world has to deal with noisy sensors and imperfect information which inevitably leads to uncertainty about the state of the system. Processing of observations is therefore an important part of a RL agent even thought it is not a part of the learning algorithm itself, and it has close connections to the underlying Markov property (Section 2.1.2) and should therefore be designed with accord along with the state representation itself.

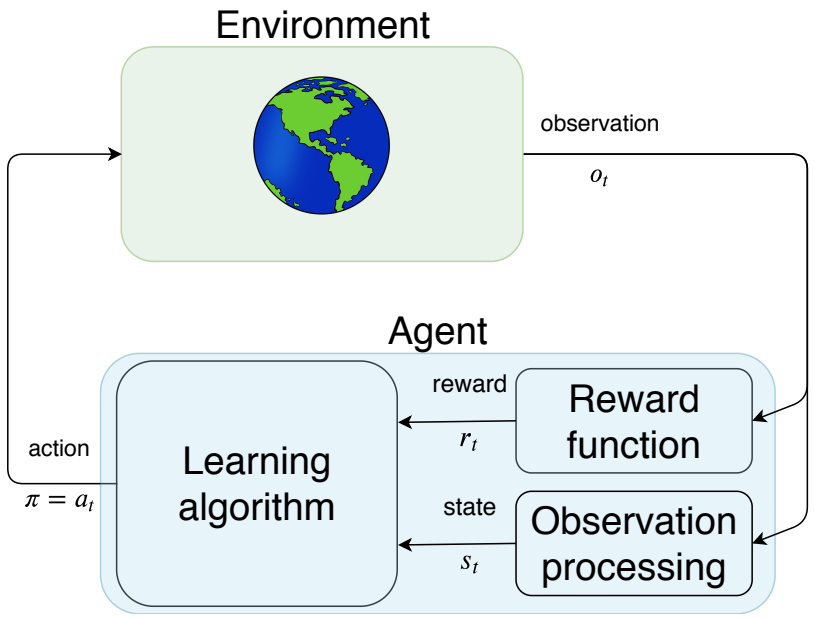


**Figure 2.1:** Illustration of the interactions between the general reinforcement learning agent and the environment in which it operates.

The distinction between the environment and the agent is not always clear. Let us say the objective is to control a robotic arm and move the end-effector to a desired location. It is then tempting to think of the robotic arm as the agent, however it would be more correct to think of the agent as the brain of

the robotic arm. The robotic arm can be viewed as a part of the environment as well, and what the agent has to learn is the controlled arms dynamics, but also its interaction with other parts of the environment. This is the gist of what is known as model-free reinforcement learning. In the case of model-based learning, the agent learns a representation of the underlying dynamics as well, but that is out of scope for this project.

### 2.1.2 Markov Decision Processes

The reinforcement learning problem can be categorized as a sequential decision making process where the agents goal is to learn how to take actions based on states that gives the maximum amount of future rewards. An important underlying assumption is that the transitions of the system is completely characterized by the immediate preceding state and action only, $S_{t-1}$ and $A_{t-1}$ respectively. This means that the dynamics of the system do not depend on earlier states and actions at all. If this holds for any state of the system, the system is said to have the Markov property, and a sequential decision problem which has this property is called a Markov decision process (MDP) [36].

In the original formulation a MDP is denoted by the tuple ($\mathcal{X}$, $\mathcal{U}$, $R$, $T$), where $\mathcal{X}$ is the set of all states, $\mathcal{U}$(x) is the set of all actions associated with a state x $\in \mathcal{X}$, $R(x, u)$ is the reward function which depends on the state x and action u and $T$ is the transition model which gives the transition probabilities $P(x^{'}|x, u)$. If the system is deterministic, then the transition function is simply given by the current state $x$ and action $u$: $x^{'} = f(x, u)$. Note that this notation is not used further, and for the rest of this report the notation introduced in Section 2.1.1 is used.

For this property to hold the present state of the system is required to contain all information that influence the future states of the system. This property is important because it reduces the amount of parameters required to construct a the transition model of an arbitrary system. To see this, consider a transition model where the Markov property holds, i.e.

$$P(s_{t+1|s_t} = P(s_{t+1}|s_t, s_{t-1}, ..s_1). \tag{2.1}$$

This means that we can discard all previous states $s_{t-1}, ..s_1$ as we try to predict the future.

However, all the necessary information to infer about the future is not always available to us. Usually we operate in partially observable universes where the true state of the system is not always given, but instead we observe evidence of it. A MDP where the state is only partially observable is called a Partially Observable Markov Decision Process (POMDP). A POMDP can be converted into a MDP by introducing a belief state that can be derived from the observed evidence, which is the observation $o_t$ in Section 2.1.1. This is useful in many situations and can be used to act optimal based on the agents beliefs of the world.

### 2.1.3 Rewards

The reward the agent receives at each timestep is designed by the programmer and depends on the current state of the agent in the world and its goal. The reward function can for example be related to how close the agent is to a setpoint or path, or it can be a negative reward for each timestep that is not a terminal state, an absolute goal, and a positive reward for reaching the terminal state. The reward is essentially a scalar measure of how good the agent is currently doing, and is a function of the state variable of the agent such that the agent is able to infer about the value of its state. The performance of the agent depends on the quality of the reward function and its design is therefore important.

The agent's goal is in general to maximize its total expected reward over a time horizon known as an episode. The total expected reward is called the expected return $G_t$, and in its simplest form it is just the sum of the rewards over an episode:

$$G_t \triangleq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{2.2}$$

However, the simple sum of rewards are seldom used. Instead the discounted sum of future rewards are used. The agent chooses $A_t$ which maximizes the

expected discounted return

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{N-1} R_{t+N} = \sum_{n=0}^{N} \gamma^k R_{t+n+1}. \quad (2.3)$$

The discount factor $\gamma$ determines the present value of future rewards such that the agent will tend to prefer immediate rewards over future rewards. When $\gamma \to 1$ the agent becomes more farsighted, and when $\gamma \to 0$ the agent becomes more intent on immediate rewards.

It is important to note that returns and successive timesteps are related to each other in a recursive relationship as

$$\begin{aligned} G_t &\triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4}) + ... \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \quad (2.4)$$

which also simplifies notation.

### 2.1.4 Value functions and policies

So far we have talked about the signals that the agent interacts with: the observation $O_t$ the agent receives that it uses to represent its current state $S_t$, the input (action) the agent applies to the environment $A_t$, and the reward the agent receives $R_t$. Most reinforcement learning algorithms uses these quantities to define a "goodness" of states or taking certain actions in a state. The "goodness" defines how much future reward is expected from a certain state and onward into the future. This value is what has to be estimated, or learned, by the agent.

Closely associated to the notion of valuing a state and the actions that can be performed in them we have what is known as a policy which describes the way the agent acts. The policy of an agent is a mapping from the current state to either a probability distribution of actions, or a mapping directly from state to action known as a deterministic policy. To present the basic principles in this chapter the probabilistic policy $\pi(a|s)$ is used, where the probability of selecting action $A_t = a$ is given by the state $S_t = s$ at timestep $t$.

By definition the value function $v_\pi(s)$ expresses the value of a state $S = s$ when acting under the policy $\pi$ from that particular state and onwards. In other words, it is the expected return from starting in $s$ and then following $\pi$ to the end of time. It is called the state-value function for policy $\pi$. For MDPs this can be expressed formally as

$$v_\pi(s) \triangleq \mathbb{E}\left[G_t | S_t = s\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+n+1} | S_t = s\right]. \tag{2.5}$$

In a similar fashion we can define the value of action $a$ in state $s$ under policy $\pi$ as $q_\pi(s, a)$, which is called the action-value function. This is expressed as

$$\begin{aligned} q_\pi(s, a) &\triangleq \mathbb{E}\left[G_t | S_t = s, A_t = a\right] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+n+1} | S_t = s, A_t = a\right]. \end{aligned} \tag{2.6}$$

The value– and action-value functions are learned from experience, and a fundamental property of these functions is that they satisfy a recursive formula similar to that of the recursive formula for returns given by equation (2.4). This formula expresses a consistency condition between a state and its successors. Hence, states that are spatially related to each other are related with respect to their expected future value. For $v_\pi(s)$ this can be expressed as:

$$\begin{aligned} v_\pi(s) &\triangleq \mathop{\mathbb{E}}_{\pi}\left[G_t | S_t = s\right] \\ &= \mathop{\mathbb{E}}_{\pi}\left[R_{t_+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathop{\mathbb{E}}_{\pi}\left[G_{t+1} | S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}. \end{aligned} \tag{2.7}$$

This is known as the Bellman equation for $v_\pi$. It expresses a relationship between a state $s$ and successor states $s'$. The Bellman equation averages all the rewards from successor states $s'$ weighted by their probabilities to form an esti-

mate for the state $s$ that is equal to the discounted sum of the expected rewards from the successor states $s'$ plus the immediate reward that is associated with transitioning from $s$ to $s'$. This can be viewed as a one-step look-ahead into all possible future states and performing a backup operation that conveys information about the future states back to the current state. Deriving the Bellman equation for the action-value function

This is actually a fundamental property which makes reinforcement learning methods able to not only learn from complete episodes of returns, but also learn from any transition between states that the agent experiences.

### 2.1.5 Bellman optimality equations

Broadly speaking, the reinforcement learning problems objective is to find a policy that generates the most reward over time. A policy $\pi$ is considered better or equal to another policy $\pi'$ if its expected return is greater or equal to that of $\pi'$. In terms of the value function this can equally be expressed as having $v_\pi(s) \geq v_{\pi'}(s)$ for all s $\in \mathcal{S}$. The policies that satisfy this inequality have the same state-value function called the optimal state-value function $v_*$, and they are denoted as the optimal policy $\pi_*$. The optimal state-value function can be expressed as

$$v_*(s) \triangleq \max_\pi v_\pi(s) \tag{2.8}$$

for all possible state s $\in \mathcal{S}$. The optimal policies also have the same optimal action-value function

$$q_*(s,a) \triangleq \max_\pi q_\pi(s,a) \tag{2.9}$$

for all possible states s $\in \mathcal{S}$ and all possible actions a in state S, a $\in \mathcal{A}(s)$. The optimal action-value function $q_*$ can also be expressed as a function of $v_*$ as

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{1+t}) | S_t = s, A_t = a\right]. \tag{2.10}$$

These functions also satisfy the consistency condition given by the Bellman equation in (2.7). The consistency condition for $v_*$ can be expressed without a reference to a specific policy because it is the optimal value function. This means that the value of any state while following the optimal policy is equal to the expected return of the best action $a$ in state $s$. This is known as the Bellman optimality equation, and for $v_*$ it is expressed as

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma v_*(S_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r | s, a)\left[r + \gamma v_*(s')\right]. \end{aligned} \tag{2.11}$$

By using (2.4). For $q_*$ the Bellman optimality equation is given by

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r | s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]. \end{aligned} \tag{2.12}$$

The importance of the Bellman optimality equations can also be intuitively understood by Richard Bellman's Principle of Optimality:

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision* [37].

This principle suggests that a sequential decision problem can be divided into

smaller sub-problems and the optimal course of action in any of the sub-problems also constitutes the optimal action in the full problem. Problems that inherits this property is said to have optimal substructure and is fundamental to dynamic programming, which why the Bellman equation is also known as the dynamic programming equation. These equations are the cornerstone for solving the problem of maximizing cumulative reward in a sequential decision problem, otherwise known as reinforcement learning.

### 2.1.6 Temporal-difference learning

The Bellman equation (2.7) indirectly assumes that we are able to attempt all actions in any given state and observe the outcome to predict the value of any state at any time. This would be equivalent to performing an exhaustive search over the state– and action space. The ability to observe all possible outcomes given any state and all possible actions is seldom the case, and is really only possible if the environment is a simulation. It would for example not be possible if the agent operates in the real physical world and intractable if the action or state variables are continuous variables. The latter is the case for the project in this report, and more importantly: the transition probabilities which describes the dynamic of the system is unknown. The Bellman equation also assumes that there is an end state, which for this project is neither the case because an episode may be of arbitrary length. It therefore seems clear that a different approach must be taken.

A fitting approach that utilizes the Principle of Optimality (Section 2.1.5) is a class of model-free reinforcement learning methods called temporal-difference learning (TD learning). These methods performs an update to the current state estimates by using the difference of the current states value estimate and the discounted value of the proceeding state plus the reward associated with that transition to give a prediction of the value estimate for the current state. Because learning only requires one transition between states these methods can be applied to problems with episodes of arbitrary lengths. The learning process is as an iterative process and is known as bootstrapping in statistics. To illustrate this is the TD update rule which can be used to predict $v_\pi$, which gives the value of any state while following the policy $\pi$:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]. \tag{2.13}$$

Here $\gamma$ is the discount parameter, $V(S_t)$ is the value of the current state $S_t$, $V(S_{t+1})$ is the value of the proceeding state $S_{t+1}$, $R_{t+1}$ is the reward associated with the transition $S_t \rightarrow S_{t+1}$, and $0 < \alpha < 1$ is the learning rate used to move the estimate of $V(S_t)$ towards its predicted value. Note that the capital "V" for the value function expresses that this is an estimate of the value function, and not the true value function which we dealt with in Section 2.1.4-2.1.5. The expression in brackets in equation 2.13 is known as the TD-error,

$$\delta_t \triangleq R_{t+1} + \gamma V(S_{t+1}) - V(S_t), \tag{2.14}$$

and is measure of the estimate error at the time. Furthermore, the first two terms in the TD-error is known as the target,

$$y_t \triangleq R_{t+1} + \gamma V(S_{t+1}), \tag{2.15}$$

and expresses the current estimate of the true value function $v_\pi$ under the current policy.

The full algorithm is shown in Algorithm 1 and is known as the TD(0) algorithm. It has shown to converge to an optimal value function $v_*$ under the following conditions [38]:

1. The estimated values are stored in memory, such that they are not forgotten.
2. The learning rate, at any time, satisfies the condition $0 < \alpha < 1$.
3. The variance of the reward is bounded, hence Var($R_t$) $< \infty$.

**Algorithm 1** TD(0) prediction

Initialize learning rate $\alpha \in (0, 1]$
Initialize $V(s)$ for all states s $\in \mathcal{S}$.
**for** episode in $1 : M$ **do**
  Set $S$ = $S_0$
  **for** timestep in episode $1 : n$ **do**
    $A \leftarrow$ action given by $\pi(S)$
    Perform action A, observe resulting reward R and new state $S^{'}$
    $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S^{'}) - V(S)\right]$
    $S \leftarrow S^{'}$
  **end for**
**end for**

The TD(0) algorithm alleviates the problem of needing to know the outcome of all possible actions at any point in time, the problem of unknown system dynamics, and allows episodes to be of any length, even infinite. There is however one problem with this algorithm: it does not tell us how to improve our policy, and in addition this method is less data efficient than it could be because it only learns about the policy it is currently following. Fortunately we can do better if we use the action-value function as opposed to the value function, which is known as Q-learning. This is an off-policy TD learning algorithm, which means that the learned action-value function Q directly approximates the optimal action-value function $q_*$ regardless of what policy is being followed. We have

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right], \tag{2.16}$$

which is very similar to Equation 2.13 for the update rule for TD(0), however transitioning from value– to action-value functions and using the max operator for choosing the action in the proceeding state $S^{'}$. This can be intuitively understood by realizing that the value of the current state and the chosen action can only be as good as the best option (or action) the agent can choose in the proceeding state.

The full algorithm is shown in Algorithm 2. For the algorithm to converge

to the true action-value function it is also required that all states are visited an infinite numbers of times, meaning that the probability of visiting a state never becomes identical to zero. A typical strategy for assuring this is to use an $\epsilon$-greedy policy that chooses a random action with a non-zero probability $\epsilon$.

**Algorithm 2** Q-learning

Set hyperparameters like total amount of episodes to train $M$ step size $\alpha$, discount factor $\gamma$, and small $\epsilon > 0$ for epsilon greedy exploration.
Initialize storage for action-values $Q(s, a)$ and set initial values for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
**for** episode in $1 : M$ **do**
  Initialize $S$
  **while** S is not a terminal state **do**
    Choose action $A$ from $S$ using policy derived from Q (e.g $\epsilon$-greedy)
    Take action $A$, observe reward $R$ and next state $S'$
    Do:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right] \quad (2.17)$$

    $S \leftarrow S'$
  **end while**
**end for**

The methods considered in this section requires memory to store all combinations of state value– and state-action value pairs and they are therefore often referred to as tabular methods because they require bookkeeping for all the estimates. The downfall of these methods is that they are not able to deal with continuous state– and action spaces at all because it would require an infinite amount of memory.

In some cases a solution can still be derived by discretizing the environment, but for complex problems – like control in robotics – this would inevitably lead to a sub-optimal solution because of loss of information. It is also worth noting that even though discretization may enable use to form a solution the amount of memory needed is still growing exponentially with the number of state variables, and therefore the solution would still be intractable to compute in many cases. The next section therefore discusses a learning

method that is applicable to continuous environments.

### 2.1.7 Policy gradient methods

While the previously discussed learning methods were dependent on keeping all value- and action-value estimates in memory, policy gradient methods can learn a parameterized policy that selects an action directly given a state without approximating the value or action-value function of the environment. Policy gradient methods are generally divided into three different categories: actor-only, critic-only and actor-critic methods. This section is dedicated to discussing actor-only methods, otherwise known as vanilla policy gradient methods. The next section discusses actor-critic methods, while critic-only methods are omitted because they are best fit to deal with discrete action spaces which is out of scope for this project.

The notation for the parameterized policy is denoted as $\pi(a|s, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the policys parameters. The policy outputs a probability for choosing an action $a_t$ given a state $s_t$ and parameters $\boldsymbol{\theta}_t$ at a time $t$: $\pi(a|s, \boldsymbol{\theta}) = P(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$. The policys parameters also have the t-subscript indicating they are also time-varying which is because they are subject to optimization.

Policy gradient methods seek to maximize some scalar performance measure $J(\boldsymbol{\theta})$. With respect to the policy parameters $\boldsymbol{\theta}$ the objective can be optimized by doing gradient ascent using the gradient of $J$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t) \tag{2.18}$$

where $\nabla J(\boldsymbol{\theta}_t)$ is a stochastic estimate whose expectation approximates the real gradient of the performance measure $J$. The objective which we seek to maximize is the discounted expected return

$$J(\boldsymbol{\theta_t}) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t\right]. \tag{2.19}$$

We should expect the above expectation to be equal to $v_{\pi_{\boldsymbol{\theta}}}$ where $\pi_{\boldsymbol{\theta}}$ is the parameterized policy. And therefore we need a way of calculating

$$\nabla J(\boldsymbol{\theta}) = \nabla v_{\pi_{\boldsymbol{\theta}}}(s_0). \tag{2.20}$$

This however is not trivial, because the effect of the policy on the state distribution is a function of the environment dynamics which is unknown to us. This is where the policy gradient theorem [39] comes in, which for the episodic case can be stated as

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}(s_0)} \\ &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \end{aligned} \tag{2.21}$$

where $\mu(s)$ is the weighted state distribution probabilities under the policy. Note that the right side of (2.21) can be calculated if the parameterized policy $\pi_{\boldsymbol{\theta}}$ is differentiable with respect to its parameters.

Parameterization can be realized by devising a parameterized numerical preference function $h(s,a,\boldsymbol{\theta})$ that assigns each action a probability of being chosen. An example of this is the exponential soft-max function which gives the probability distribution and policy

$$\pi(a|s,\boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{2.22}$$

where $e \approx 2.711828$ is the base of the natural logarithm. An advantage of using the soft-max function is that exploration is assured by the stochastic policy, and it will eventually converge to a deterministic policy as the agent learns the best actions. This is clearly advantageous to the $\epsilon$-greedy strategy often used for exploration in tabular methods as we saw in Section 2.1.6. While this is one example of a parameterization function the choices are many and the choice is dependent on the specific problem. What is important is that the parameterized function is able to approximate the policy function to a satisfactory degree. By using a function approximator the requirement of visiting all states to compute their value is deposed off, because the function also approximates the neighbouring action-value pairs, not only the pairs it has observed. Other examples of function approximators includes linear combinations of features (states) and weights $\boldsymbol{\theta}$ as $h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathrm{x}(s,a)$, or deep artificial neural networks (Sec-

tion 2.2).

The policy gradient theorem provides us with an expression that is proportional to the gradient. All we need now is a way to sample a quantity which is equal to approximates this expression and eliminating the dependence on the action-value function $q_\pi$ such that a policy can be learned without consulting the any value function. This can be done by noticing that the right-hand side of Equation 2.21 is a sum over states weighted by how often states are visited under the policy $\pi$, which if followed are encountered in these proportions, hence:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}_t) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathop{\mathbb{E}}_{\pi} \left[ \sum_a q_\pi(S_t,a) \Delta \pi(a|S_t,\boldsymbol{\theta}) \right] \end{aligned} \tag{2.23}$$

Next, the last sum and $q_\pi$ can be reduced to the return $G_t$ by multiplying and dividing the inner sum by $\pi(a|S_t,\boldsymbol{\theta})$. This is because the expectation under the weighting $\pi(a|S_t,\boldsymbol{\theta})$, which is the correct sum over actions, equals $q_\pi(S_t,A_t)$, therefore:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}_t) &= \mathop{\mathbb{E}}_{\pi} \left[ \sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\Delta \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \\ &= \mathop{\mathbb{E}}_{\pi} \left[ q_\pi(S_t,A_t) \frac{\Delta \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \\ &= \mathop{\mathbb{E}}_{\pi} \left[ G_t \frac{\Delta \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \end{aligned} \tag{2.24}$$

This sample can now be plugged into the generic gradient descent algorithm (Equation 2.18) which yields the following update rule better known as the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\Delta \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \tag{2.25}$$

The full REINFORCE algorithm is shown in Algorithm 3. It is important to note that REINFORCE uses the complete return which includes all rewards

the agent has experienced up to time t. Learning approaches that uses returns like this are known as Monte-Carlo algorithms which are not well defined for problems that have episodes of infinite length. These methods are not discussed further in this report as they are not used and we consider them to be a sub-optimal approach for the problem studied in this project.

**Algorithm 3** REINFORCE

Initialize the differentiable policy parameterization $\pi(a|s, \theta)$
Initialize hyper parameters: learning rate $\alpha$, discount rate $\gamma$
**for** episode in $1 : M$ **do**
  Generate an episode $S_0, A_0, R, 1...S_{T-1}, A_{T-1}, R_T$ following the current policy
  **for** each step in the episode $t = 0, 1, ...T - 1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \ln \pi(A_t|, S_t, \theta)$
  **end for**
**end for**

### 2.1.8 Actor-critic methods

The policy gradient method described in the previous section relies on the episodic returns, which is the Monte-Carlo approach to learning. Monte-Carlo learning has higher variance than temporal difference learning because temporal differences bootstraps and thus the action and immediate reward from one timestep to another is not affected by previous actions in that episode. Since all policies must be stochastic to some degree in order to learn, Monte-Carlo is more prone to variance because at each timestep there is potential variance that is injected into the return for that episode. In addition, Monte-Carlo methods can be hard to implement in practice for continuing problems.

To alleviate some of this problem, we now introduce parameterized value function, known as the critic, in addition to the parameterized policy which we will now reference to as the actor. By modelling the critic as a bootstrapping method we can thus reduce the variance associated with the pure policy gradient approach. The value function is similarly to the parameterized policy denoted $v(s, \mathbf{w})$ where **w** is the parameter vector.

A popular approach to actor-critic methods is therefore to use a one-step

return like $TD(0)$, however any policy evaluation technique may be used in practice. The parameter update rule from REINFORCE in Equation 2.25 can be written in terms of the $TD(0)$ update to form an online algorithm by replacing the return $G_t$ with $G_{t:t+1} - \hat{v}(S_t, \mathbf{w})$ whose expectations are the same:

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha(G_{t:t+1} - \hat{v}(S_t, \mathbf{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha\delta_t \ln\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t).
\end{aligned} \tag{2.26}$$

Where $\delta_t$ is the TD-error introduced Section 2.1.6. For the critic, the squared $TD(0)$ is used as the loss function because it is the estimated error between the approximated value and the observed value (the reward) we would like to minimize, hence

$$J(\mathbf{w}) = \frac{1}{2}\delta_l^2, \tag{2.27}$$

where $\delta_l = (R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}))$. And because we want the TD error to converge to 0 as the value function approaches the true value function, we get the parameter update law directly by using $\nabla J(\mathbf{w}) = \delta_l\nabla v(S_t, \mathbf{w}_t)$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\nabla v(S_t, \mathbf{w}_t). \tag{2.28}$$

We have now arrived at a model-free reinforcement learning method which are able to deal with both continuous state– and action spaces while also being applicable to problems that have episode undefined episode lengths. In the next section one such method is introduced which is later used as a basis for a high-level controller.

### 2.1.9 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) algorithm [4] is an actor-critic algorithm that builds on the work of Silver et al [40] who found that the deterministic policy gradient exists such that deterministic policies (not to be confused by the weighed state distribution probabilities in Section 2.1.7), denoted $a = \mu(s|\boldsymbol{\theta}^{\mu})$ can be constructed. The deterministic policy gradient is

$$\nabla_{\boldsymbol{\theta}^{\mu}} J \approx \nabla_a Q(s, a|\boldsymbol{\theta}^{Q}) \nabla_{\boldsymbol{\theta}^{\mu}} \mu(s|\boldsymbol{\theta}^{\mu}) \tag{2.29}$$

Where $\boldsymbol{\theta}^{\mu}$ are the parameters of the policy function and $\boldsymbol{\theta}^{Q}$ is the parameters of the action-value function, and hence $Q(s, a|\boldsymbol{\theta}^{Q})$ is the parameterized action-value function and $\mu(s|\boldsymbol{\theta}^{\mu})$ the parameterized policy function. The deterministic policy $\mu(s|\boldsymbol{\theta}^{\mu})$ deterministically maps a state $s$ to an action $a$ given the policy parameters $\boldsymbol{\theta}^{\mu}$.

The approach was also inspired by Mnih et al's [41] contribution to reinforcement learning through an algorithm called Deep Q-Networks (DQN). DQN is based on Q-learning and uses neural networks as function approximators and are able to take continuous state spaces as input to produce discrete actions and output. DDPG specifically utilizes two inventions that allowed DQN to learn in a stable and robust way:

1. *Replay buffer* that stores past experienced transitions. This allows for off-policy training of mini-batches of past experienced sampled uniformly from the replay buffer, which increases sample– and computational efficiency which stabilizes training.

2. Separate *target networks* with separate parameters of the parameterized policy– and action-value function to calculate the targets. In [4] these are updated according to the "soft" update law:

$$\begin{aligned} \boldsymbol{\theta}^{\mu'} &= (1 - \tau)\boldsymbol{\theta}^{\mu'} + \tau\boldsymbol{\theta}^{\mu} \\ \boldsymbol{\theta}^{Q'} &= (1 - \tau)\boldsymbol{\theta}^{Q'} + \tau\boldsymbol{\theta}^{Q} \end{aligned} \tag{2.30}$$

   Where $\boldsymbol{\theta}_{\mu'}$ and $\boldsymbol{\theta}_{Q'}$ are the target policy– and action-value- function parameters for their respective target policy function $\mu'$ and target critic

function $Q'$. This constrains training of the targets to happen at a more slow pace which improves stability.

The critic is learned using Q-learning and allows the agent to learn off-policy, i.e the behaviour policy simulates trajectories while evaluating and improving Q-values regardless of what policy is being followed. This is similar to the loss function for the critic in Section 2.1.8, but using the action-value function instead. By using the target networks to compute the returns, we get

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{s+1}|\boldsymbol{\theta}^{\mu'})|\boldsymbol{\theta}^{Q'}) \tag{2.31}$$

and the critics loss function

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\boldsymbol{\theta}^Q))^2. \tag{2.32}$$

This is the sum of all the losses from $i$ to $N$ where $N$ is the mini-batch size used for training.

An advantage of being off-policy is that the algorithm can incorporate exploration into the process without interfering with the learning algorithm itself. In DDPG an Ornstein-Uhlenbeck process is used to generate temporally correlated noise which is added directly to the actor policy output:

$$\mu'(s_t) = \mu(s_t|\boldsymbol{\theta}_t^{\mu}) + \mathcal{N} \tag{2.33}$$

Where the noise process is denoted $\mathcal{N}$. The Ornstein-Uhlenbeck process behaves like a Wiener process, and has shown to be efficient in physical control problems with inertia, but other noise models may be used as well depending on what best suits the particular system.

The full algorithm is shown in Algorithm 4. Note that in practice the learning steps do not start before the replay buffer contains at least a mini-batch size amount of transitions.

**Algorithm 4** DDPG

Randomly initialize critic network $Q(s, a|\boldsymbol{\theta}^Q)$ and actor $\mu(s|\boldsymbol{\theta}^\mu)$ with weights $\boldsymbol{\theta}^Q$ and $\boldsymbol{\theta}^\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$, $\boldsymbol{\theta}^{\mu'} \leftarrow \boldsymbol{\theta}^\mu$
Initialize replay buffer R.
**for** episode $1 : M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration.
  Receie an initial observation $s_1$
  **for** each step in the episode $t = 1 : T$ **do**
    Select action $a_t = \mu(s_t|\boldsymbol{\theta}^\mu) + \mathcal{N}$ according to the current policy and exploration noise.
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{s+1}$.
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{s+1}|\boldsymbol{\theta}^{\mu'})|\boldsymbol{\theta}^{Q'})$
    Update critic by minimizing the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\boldsymbol{\theta}^Q))^2 \tag{2.34}$$

    Update the actor policy using the sampled policy gradient:

$$\nabla_{\boldsymbol{\theta}^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\boldsymbol{\theta}^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\boldsymbol{\theta}^\mu} \mu(s|\boldsymbol{\theta}^\mu)|_{s_i} \tag{2.35}$$

    Update the target networks:

$$\begin{aligned} \boldsymbol{\theta}^{\mu'} &\leftarrow (1-\tau)\boldsymbol{\theta}^{\mu'} + \tau\boldsymbol{\theta}^\mu \\ \boldsymbol{\theta}^{Q'} &\leftarrow (1-\tau)\boldsymbol{\theta}^{Q'} + \tau\boldsymbol{\theta}^Q \end{aligned} \tag{2.36}$$

  **end for**
**end for**

## 2.2 Artificial neural networks

As we have already established in Chapter 1, the control of quadrotors is a challenging task because of the nonlinear system dynamics, and therefore we need a parameterized functions that is capable of approximating the dynamics of the system, the action-value and policy function space. For this we will use

artificial neural networks (ANNs), which when used as function approximators in reinforcement learning applications is often called deep reinforcement learning (DRL). This chapter presents the basic operations of ANNs, justifies the choice of using them in this project, and presents some of the most relevant technologies that are used in state-of-the-art neural network architectures.

### 2.2.1 Basic operations

Artificial neural networks can be thought of as a mathematical function that maps an input $\boldsymbol{x}$ to a final output $\boldsymbol{y}$ where the dimension of both the input and output can be of arbitrary size. Figure 2.2 shows a generic ANN structure which can be characterized as a directed graph where the arrows are the incoming signals from the previous layer, each blue circle represents a neuron, and the grey circles represents the input layer that performs data augmentation on the input. The neurons consists of a set of weights $\boldsymbol{w}$ and a bias $b$ and an activation function $g(\cdot)$ that is parameterized by the weights. For each neuron the incoming signal $\mathbf{x}$ is processed such that the output from the neuron is $g(\boldsymbol{w}\boldsymbol{x}^T + b)$, and therefore the mapping from the input to the output of the network can be viewed as several steps of matrix multiplications. The final output y is then a nested function of activations, $y = f(g(g(..)..)$. The operation of producing a output from the input is sometimes referred to as the *feed-forward operation*.

The other fundamental operation of neural networks is the *backpropagation* algorithm. By using the chain rule known from basic calculus and a loss function $J$ the gradient of each weight can be found by propagating the error from the end of the network backwards into the layers to calculate the contributing error affiliated with each parameter in the network. The loss function is parameterized by the networks parameters $\theta$, where $\theta$ is the set of all weights and biases in the network, and expresses the error in the networks prediction $y$ relative to a target. The target is what we want the network to produce, hence its name, and we therefore want this error to tend to zero. The parameters of the network is adjusted by an optimization algorithm, where the most popular method used is the gradient descent algorithm. Given that the loss function is a differentiable function its parameters can be adjusted by gradient descent as

follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\theta} J(\boldsymbol{\theta}) \tag{2.37}$$

where $\alpha$ is the learning rate.

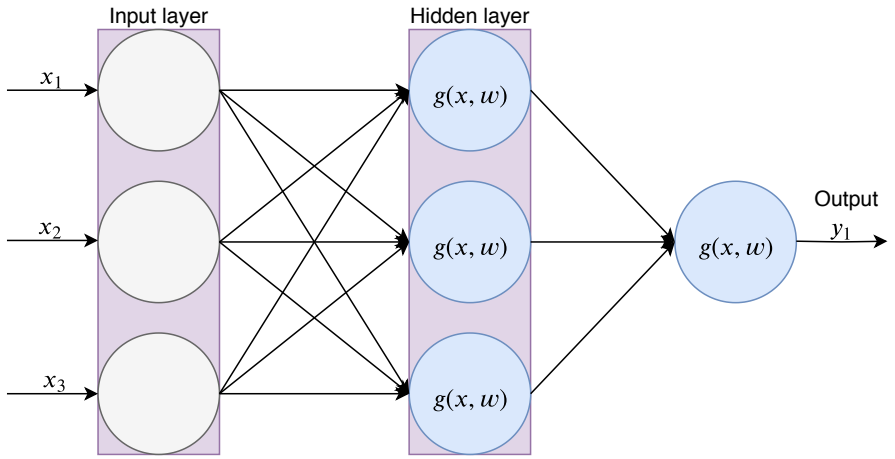


**Figure 2.2:** A generic feed-forward ANN with three input units, one hidden layer with three neurons, and a single output neuron.

### 2.2.2 Universal function approximator

The activation functions of the network plays in important role in networks ability to approximate the function of the sampled inputs it is given. By letting the activation functions be nonlinear functions, one can prove [42] that a network with only one hidden layer and a finite number of neurons is in fact an universal function approximator that can approximate any continuous function on a compact region of the network's input space to any degree of accuracy. However, most networks used in practice displays at least two hidden layers and often many more. This is because of an important insight [43] that shows that deeper and bigger networks are required to extract abstract features from the input space while at the same time being able to generalize well with respect to new inputs.

Networks with more than hidden layers, multi-layered, are often called deep neural networks (DNNs). Training these networks can be difficult with respect to convergence and stability because of the amount of parameters in the optimization problem. Choosing just the right architecture with respect to number of neurons in hidden layer, the number of hidden layers and the activation functions in each of them can also be a challenging with respect to the problem of over– and underfitting. These problems arises when the network approximates the input to an accurate degree but fails to predict the output of input it has not seen before, and when the training of the network is stopped prematurely, respectively.

Because of the ANNs versatility as a function approximation it is therefore an appealing choice for reinforcement learning. Despite the shortcoming of stability proofs, the computational expense of training the networks, and the difficulty of explaining the networks actions, they have been applied to reinforcement learning with great success. They have been showcased in the majority of the most successful applications during the past years, including problems [44][30][29] that are similar to the one studied in this project. Indeed, it would seem that using ANNs in conjuncture with reinforcement learning is the rule rather than the exception and monumental to the advances in the field.

### 2.2.3 Improvements

Neural networks have been studied intensively and many extensions to the activation functions, network architecture and optimizers have been added during the last few years. Much of the research has been driven by the success of applying convolutional neural networks to image classification and has led to efficient frameworks for computing neural networks being developed.

Examples of extensions to the classical ANN architecture includes batch normalization [45] which normalizes the input till each hidden layer by reducing the covariance shift, the rectifier linear unit (ReLU) [46] which is an activation function with many appealing features such as being less computationally expensive than previous activation functions and mitigating the vanishing gradient problem [47], Xavier initialization [48] which is method for initializing each weight in each layer that improves convergence by reducing the effect of

the vanishing– and exploding gradients, and the Adam optimizer [13] which is an advanced form of gradient descent that utilizes the moments of the gradients and keeps a per parameter learning rate. There are many other extensions that are used in state-of-the-art applications, but the aforementioned examples are the most relevant as they are used directly in this project, and therefore deserves a mention on their own.

## 2.3 Computer vision

In this project a single camera is used for providing the reinforcement learning agent with observations. Instead of using the raw images, every pixel, as input, they are processed such that the dimensionality of the input space is reduced. This enables us to filter away superfluous information allowing the neural networks to train faster and more efficiently, and allowing networks to be trained within a feasible time frame governed by the period and scope of this project and the hardware available.

This section introduces a color model used for image processing, a simple method for tracking a distinctive colored object in an image, and how the method can be used to infer about the position for a quadrotor by considering its dynamics and the camera configuration. The object tracking method discussed in Section 2.3.3 is used in the controller design in Section 3.3, but is not an advanced method at all, and entails that information is lost under transformation. The choice of simplicity is intentional in order to explore the reinforcement learning methods ability to act based on incomplete and imperfect information for control. The chosen method does not represent an approach that can be expected to work robustly in general, but rather used for convenience in this project. The theory in this section is based on the book "Computer vision: Algorithms and Applications" by Richard Szeliski [49].

### 2.3.1 Color model

Computer displays produces colors by mixing the three so-called additive primary colors red, green and blue (RGB). However, for the human intuition it is hard to conclude what is the correct mixing of the three colors to produce

any another other specific color, especially with respect to the lightness and intensity of the color. This is what motivated the creation of the hue, saturation and value (HSV) color model, which is based on how colors are understood by the human perception system. A visualization of the HSV model is shown in Figure 2.3. The figure depicts a cone where the the six most distinctive colors consisting of the aforementioned additive colors red, green and blue, and the subtractive colors yellow, magenta and cyan are featured around the circular disc. Along the radius of the cone is the saturation, which defines the "colorfulness" of the color relative to its brightness (or value), and along the height of the cone is the value which defines the brightness of the color. This model is useful for a human designer to define the color he/she is looking for in an image with some leeway regarding the luminance in the room where the picture is taken, and therefore provides some robustness to the experimental setup featured later in this report.

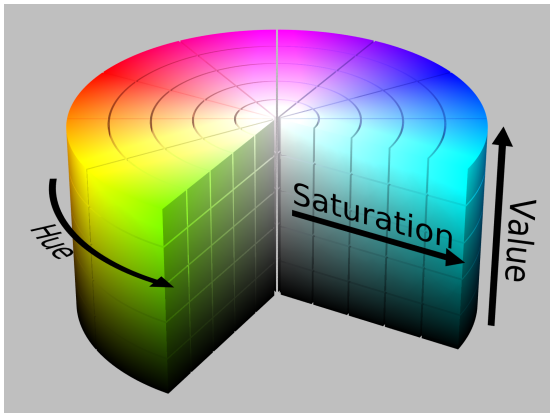


**Figure 2.3:** Visualization of the HSV color model (Wikipedia).

### 2.3.2 Pose estimation

The most relevant problem in computer vision in relation to this project is what is known as pose estimation, which is the problem of determining the pose – or transformation – of an object in a 2D image to the correct corresponding points in the 3D world. In this section we will introduce the concept by using camera projection as a starting point.

Let us introduce two coordinate systems: the image plane, and the world coordinates in which both the object and quadrotor reside in. The image plane, or just the image, is a mapping from 3D points in the world coordinate frame to a 2D plane conducted by the camera. This is visualized in Figure 2.4. It is possible to calculate the 2D to 3D projection, which is the transformation from pixel coordinates (u,v) in the image plane to world coordinates (X, Y, Z) if the camera matrix is known:

$$\tilde{\boldsymbol{x_s}} = \boldsymbol{P}\boldsymbol{p_w} \tag{2.38}$$

where $\tilde{\boldsymbol{x_s}} = [u, v]$ is the pixel coordinate, $\boldsymbol{p_w} = [X, Y, Z]$ is the world coordinate point, and the camera matrix $\boldsymbol{P}$ is given by

$$\boldsymbol{P} = \boldsymbol{K}\begin{bmatrix}\boldsymbol{R} & | & \boldsymbol{t}\end{bmatrix}. \tag{2.39}$$

$\boldsymbol{K}$ is known as the calibration matrix that describes the camera intrinsics mapping from the camera coordinate frame to the image plane, $\boldsymbol{R}$ is the rotation matrix relating the camera– and world coordinate frames via rotations around the three major axis (X, Y, Z), and $\boldsymbol{t}$ is the 3D translation from the world– to the camera coordinate frame. A commonly used parameterization for the calibration matrix $\boldsymbol{K}$ is:

$$\boldsymbol{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.40}$$

where $(c_x, c_y)$ is the optical center of the camera and $f$ the cameras focal length. Usually an invertible $4 \times 4$ matrix is used to calculate the camera matrix $\boldsymbol{P}$, such that

$$\tilde{\boldsymbol{P}} = \begin{bmatrix} \boldsymbol{K} & \boldsymbol{0} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} = \tilde{\boldsymbol{K}}\boldsymbol{E} \tag{2.41}$$

where $\boldsymbol{E}$ is a homogeneous, rigid-body 3D transformation. The intrinsic parameters of the camera can be estimated a priori via measurements, i.e via matching known, corresponding points in the image with points in the world

and performing an iterative least-squares search for the intrinsic parameters.

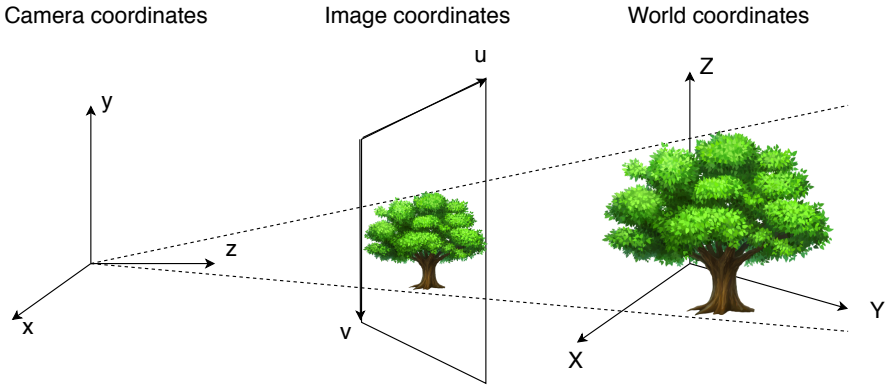


**Figure 2.4:** There are three different coordinate systems in an imaging system: the camera–, image– and world coordinate frames.

In an online setting it is possible to calculate the corresponding world points via corresponding points in the image assuming the calibration matrix is known. This is known as the *perspective-n-point problem* and there exists several algorithms for solving it. Furthermore, it is possible to estimate the pose of an object in the image if the objects geometry is known and assuming it is possible to obtain feature descriptors that lies on the object in the image. A general example of this that uses an assortment of known methods can be found on the Open Source Computer Vision Library (OpenCV) projects website [50].

However, to limit the scope of this project, and because it is of interest for this project to explore the capability of the reinforcement learning agent framework introduced in Section 2.1 to operate under limited– and imperfect information, a simpler method described in the next section is used in this project. This method has the advantage that it is computationally faster because it can be implemented via simple matrix multiplications and requires no iterative schemes to obtain a solution. On the other hand, it only provides the position of the object in the image, and hence carries less information about the true position of the quadrotor relative to the object in the real world as compared to the outlined method in this section.

### 2.3.3 Image coordinates of a colored object

To track the position of an object in an image it is possible to use the colors of the image, specifically the distinction between the colors of the object and the rest of the image. This can be achieved in two steps: masking, and then using the moments of the resulting image.

A mask in image processing is a operator that works on an a per individual pixel level by a spatial– or logical law, or both. For this task we will use a mask which returns 1 if the pixel is within a defined lower– and upper bound, and 0 otherwise. Using the HSV color model to define the bounds we can write this mathematically as

$$I_{x_i,y_j} = \begin{cases} 1, & \text{if } HSV_{upper} \geq I_{x_i,y_j} \geq HSV_{lower} \\ 0, & \text{otherwise} \end{cases} \tag{2.42}$$

where $i, j$ is the spatial index in the 2-dimensional image matrix, and $HSV_{upper}$, $HSV_{lower}$ is the lower– and upper bound by design, respectively. By applying this method we now have a binary image where each pixel equal to one is a member of the set of pixels that constitutes the object in the image.

Next, we define the image moments of the 2-dimensional structure which is our image as the following:

$$m_{p,q} = \sum_{i=0}^{i=W} \sum_{j=0}^{j=H} I_{x_i,y_j} x_i^p y_j^q \tag{2.43}$$

where $p$ is the order along the x-axis, $q$ is the order along the y-axis, $W$ the width of the image, $H$ the height of the image, and $I_{x_i,y_j}$ is the value of the at position $x_i, y_j$ for the binary image. To determine the position of the object in the image we need three different moments, the first-order moment along the x- and y axis, and the zero moment, otherwise known as the area of the object in the image:

$$m_{1,0} = \sum_{i=0}^{i=W} \sum_{j=0}^{j=H} I_{x_i,y_j} x_i^1 y_j^0 \tag{2.44}$$

$$m_{0,1} = \sum_{i=0}^{i=W} \sum_{j=0}^{j=H} I_{x_i,y_j} x_i^0 y_j^1 \tag{2.45}$$

$$m_{0,0} = \sum_{i=0}^{i=W} \sum_{j=0}^{j=H} I_{x_i,y_j} x_i^0 y_j^0 \tag{2.46}$$

Finally, the position of the object in the image can be calculated as:

$$x_c = \frac{m_{1,0}}{m_{0,0} + \epsilon} \tag{2.47}$$

$$y_c = \frac{m_{0,1}}{m_{0,0} + \epsilon} \tag{2.48}$$

where $x_c, y_c$ is the position of the centroid of the object, the moments $m_{1,0}, m_{1,0}, m_{1,0}$ is as defined in (2.44) – (2.46), and $\epsilon$ is a small constant included in practical applications in order to avoid division by zero in case the area itself is zero.

### 2.3.4 Reasoning about horizontal position

The method presented in the previous section can track the position in the image under the assumption that the color of the object is distinct from the rest of the picture. However, in a real world application we are interested in the position of a quadrotor relative to the object, not the actual position of it in the image. By choosing a convenient configuration of the camera attached to the quadrotor, and considering how a quadrotor moves through space, the position of the object in the image can be used to control the relative position of the quadrotor.

First, let us consider how a quadrotor moves in the xy-plane assuming a constant altitude. The quadrotor can be characterized has having two modes of flight: it "leans" over in order produce a net force that has components in both the vertical direction as well as in the direction of desired motion, and when it is hovering at a standstill the rotors are parallel to the gravitational plane. This is depicted in Figure 2.5.

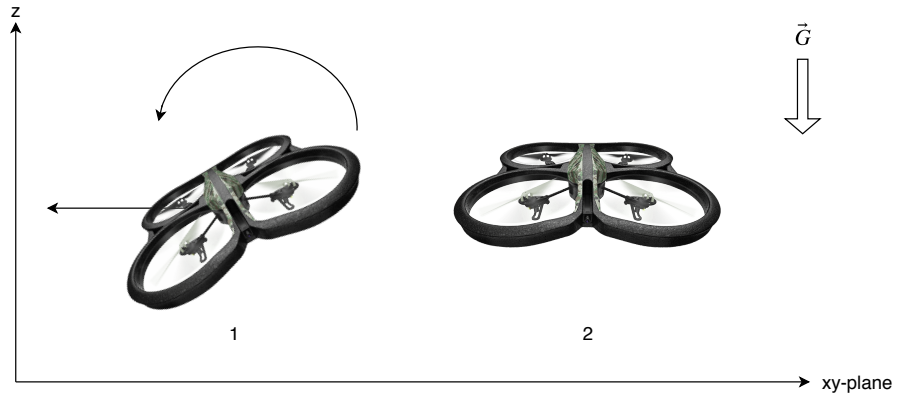


**Figure 2.5:** A simplified representation of how a quadrotor moves through space: 1) translation by rotation, and 2) standstill by aligning rotors parallel to the gravitational plane.

Given the nature of the quadrotors movement and the assumption of constant altitude, one possibility is to attach a camera underneath the chassis looking downwards, and place a visible object beneath the drone. By this configuration the case where the centroid of the object is in the origin of the image only has one stable equilibrium. In other words, given that the desired position of the object is the center of the image, $\mathbf{p}_{c,d} = [0, 0]$, then the only way for the drone to maintain $\mathbf{p}_{c,d}$ is when the velocity of the drone is zero, e.g. when the drone is at standstill. This is illustrated in Figure 2.6 which shows the two possible scenarios where the centroid position is the origin, one being the aforementioned stable equilibrium, the other being an unstable equilibrium. Note that while the figure indicates that there is only one unstable equilibrium there is actually an infinite amount of them because the camera can be centered at the object from anywhere in the xy-plane given the right tilt of the drone. However, this is usually not realistic because a drone can not normally fly sideways, and the object would disappear from the view of the camera in any case because real cameras has finite resolution.

Since the the stable equilibrium is the only equilibrium that is possible to maintain over time for the drone, this seems to be an adequate problem for the reinforcement learning framework to solve. This framework coincides soundly with this problem because its goal is to maximize its cumulative reward, and by

defining the origin of the image to be the desired position of the tracked object, the maximum cumulative reward is achieved by obtaining and maintaining the state in which the drone hovers directly above the object — the stable equilibrium. This is also the behaviour we desire from a positioning stabilizing controller.

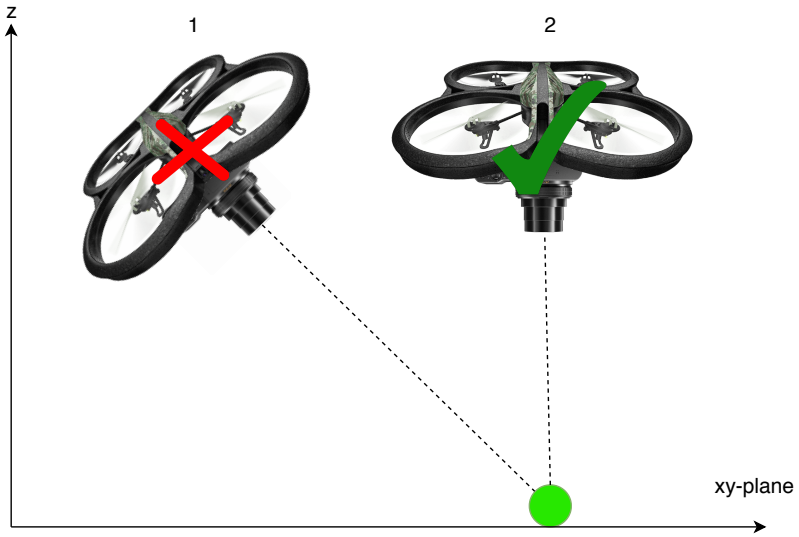


**Figure 2.6:** The camera while attached to the drone can view the object to be at the center of its view by either 1) tilting the camera such that the camera points in the direction of the object, or 2) by hovering directly above it.

Chapter 3

# System design and implementation

## 3.1 Software frameworks

In this chapter the third-party software frameworks the were used in the project are introduced. To allow the focus of the project to be within the scope of controller design several efficient frameworks was used to perform the underlying tasks. Otherwise the task would be too large to take on within the time-scope of this project. Among other things, these underlying tasks included simulating the quadrotor system, training neural networks, processing images from the camera, and inter-process communication. Note that all the software frameworks presented here are free-to-use and open-source.

### 3.1.1 Gazebo simulator

The robotics simulator Gazebo [51] was used to simulate a quadrotor. Gazebo is able to accurately simulate complex physical dynamics based on the objects mass, friction, inertia and nearly all other physical properties that is of importance to robotic vehicles in the real-world. The open-source library Open Dynamics Engine [52] (ODE) is used by default for Gazebo to calculate the dynamics and kinematics for all rigid bodies inside the simulation. A robot

consisting of its joints and links, and their physical properties is specified by its URDF (Universal Robot Description Format) files which is an XML file format. A robots URDF files describes all elements of the robot, and also has support for providing realistic 3D meshes to the robot.

The simulator provides a fine grained level of control of everything that happens inside the simulation. Gazebo allows for programs that alter and control the simulation to be run within the simulation itself through entities called "plugins" which is C++ programs that have full access to all objects resides inside the simulation through Gazebos API. This allows the user to provide the simulation with additional features. This can for example be battery-to-motor torque– or steering dynamics for a car. The simulator also ships with several plugins that is ready to be attached to the robot, which is done through the robots URDF files. Most notable is the availability of plugins that provides realistic sensor simulation for commonly used sensors in robotic applications, such as LIDAR, sonar, mono– and depth cameras, IMU, GPS and more.

### 3.1.2 Robot Operating System

Robot Operating System [53] (ROS) is an open-source robotics middleware which provides message passing between processes, implementation of commonly used functionality such as coordinate transformation and interrupt functionality for reading new sensor data and other convenient features such as logging and data visualization. ROS is multilingual and supports C++, Python and Lisp and facilitates scalable programs and a modular design of software for robotic applications. ROS also has a big online community and implementations of commonly used algorithms for robotics is easy to find. The modularity and availability of implementations makes it a particularly good fit for use in research applications, such as this project. Conveniently, ROS is also embedded into Gazebo and makes communicating with the simulation easy. Gazebo also provides services that ROS can use to reset the simulation, alter– and get the state of objects, pausing and un-pausing it. These are practical features in a reinforcement learning setting.

### 3.1.3 OpenCV

OpenCV [54] (Open Source Computer Vision Library) is an open source computer vision and machine learning library developed for accelerating computer vision applications in commercial products and providing a common infrastructure for both companies and research groups. It provides optimized implementations that are written natively in C++ of over 2500 commonly used algorithms, which is important because processing images is computationally expensive since they are essentially very large matrices. Because of its wide spread use and being open source it also has a large online community and therefore features readily available support as well.

### 3.1.4 Tensorflow

Tensorflow [55] is an open source library for numerical computation created by Google that enables large-scale machine learning. In Tensorflow the developer defines what is known as the computational graph, which is a high-level abstraction structure that defines how data moves from the input to the output. While the high-level abstractions are provided through a python API, the underlying functionality are implemented as high-performance C++ binaries. Tensorflow offers implementations of all the necessary technologies that are required to build state-of-the-art neural network architectures and promotes quick prototyping and is therefore a good fit for this project.

## 3.2 Quadrotor platform

This section introduces the experimental quadrotor platform that was chosen for this project — the AR.Drone 2.0. from Parrot Inc. This particular platform was an attractive choice for several reasons. It is a commercially available drone and is therefore easy to acquire, and it only costs about 1300 NOK ($\approx$ 150 USD) which is very cheap compared to some of the more high-end drones on the market. Because of its availability the spare parts and extra batteries are also easy to find. This is important due to the fact that testing with an experimental controller may be both time– and battery consuming — with

respect to training neural networks — and could easily lead to damage of the drone itself.

The most decisive reason for choosing this quadrotor was however that the drone features an internal velocity controller, a downwards looking camera, a simulation freely available in Gazebo, and compatibility with ROS for both the real– and simulated platform. The camera configuration was important for the feasibility of applying the method described in Section 2.3.3, and the velocity controller because we do not wish to "reinvent the wheel" as emphasized in Chapter 1, but instead base the low-control on existing and established methods. The access to a simulation was critical in order to train an reinforcement learning agent because those methods typically require a lot of data to work, and the ROS communication middleware makes development easier because the same written code can be used both on the real– and simulated drone with very few adjustments.

The next subsections introduces the most important aspects of the drone in regards to this project. This includes a short introduction to the dynamic model of quadrotors, the coordinate frames of the drone, its sensors and hardware, velocity controller, and the simulation. The interested reader may look to [56] for additional details about the physical platform.

### 3.2.1 Equations of motion

For the sake of completeness, a general dynamic model of a quadrotor is included here because it represents how the simulator – Gazebo, or rather ODE – simulates the kinematics of a quadrotor. It is common for controller design approaches to model and study the kinematics of the robot to derive a controller law. For the design of the controller in the next section the dynamics are of no concern because we use the velocity controller of the quadrotor which already controls the underlying dynamics of the quadrotor and algorithm used, DDPG, is a model-free optimization method. It is nonetheless important to have some understanding of them in respect to how accurately the simulation reflects the real world, which is of interest if we expect the controller to behave accordingly in the real world after it has been developed in a simulation.

The equations for expressing the motion of a quadrotor in 6-DOF is promptly

available in the literature, for example by [23]:

$$\begin{aligned} m\ddot{x} &= (\sin\psi\sin\phi + \cos\psi\cos\phi\sin\theta)u_1 \\ m\ddot{y} &= (-\cos\psi\sin\phi + \sin\theta\sin\psi\cos\phi)u_1 \\ m(\ddot{z}+g) &= \cos\theta\cos\phi u_1 \\ I_{xx}\ddot{\phi} + (I_{zz}-I_{yy})\dot{\theta}\dot{\psi} &= u_2 \\ I_{yy}\ddot{\theta} + (I_{xx}-I_{zz})\dot{\phi}\dot{\psi} &= u_3 \\ I_{zz}\ddot{\psi} &= u_4 \end{aligned} \tag{3.1}$$

The equations describing the linear motion in $x, y, z$ are described in a earth-fixed reference frame (otherwise known as the world frame), and the equations describing the angular motion $\phi, \theta, \psi$ are given in a body-fixed reference for convenience. Here $g$ acceleration due to gravity, $I_{xx}, I_{yy}$ and $I_{zz}$ is the moments of inertia, and $u_1, u_2, u_3$ and $u_4$ are input signals defined via the forces the propellers exert on the aircraft (see [23]). The coordinate frames and the axis of rotation of the drone is summarized in Figure 3.1 which also shows the image coordinate frame of the bottom camera discussed later.

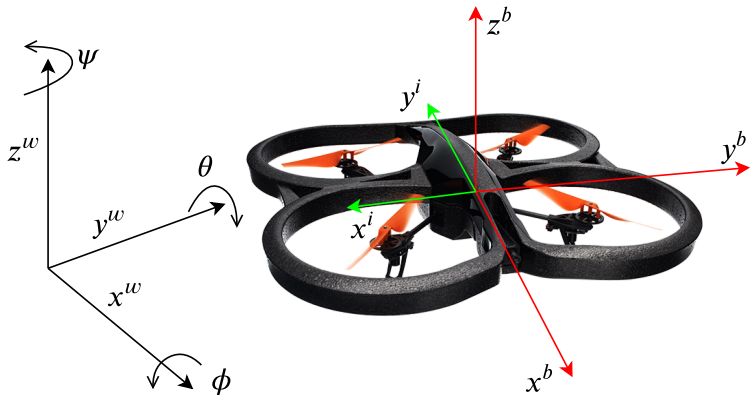


**Figure 3.1:** The Parrot AR.Drone 2.0 and its coordinate frames: the world frame {w}, body frame {b}, and image frame {i}. Both the body– and image frame are fixed to the body of the quadrotor. The angles $\phi$, $\theta$ and $\psi$ are commonly referred to as the roll, pitch and yaw angles, respectively.

It is worth noting that the particular simulation makes no effort in intro-

ducing more advanced aerodynamics, this is a major shortcoming with respect to accuracy of modeling. For example, it does not take into account complex aerodynamics such as drag forces, blade flapping and winds that occur when the aircraft hovers close to the ground. However, because we intend to move at slow speeds and at a sufficient height above the ground it should suffice for this particular application. Since we are using the velocity controller of the quadrotor it is the accuracy of this controller that is the most important aspect with respect to the accuracy of the simulation. Furthermore, having knowledge about the motion dynamics of the underlying controller can be instrumental to designing an appropriate reward function later. For a more thorough review of nonlinear aerodynamic phenomena for quadrotors see [20].

### 3.2.2 Sensors and hardware

The physical AR.Drone features several sensors intended to help stabilize the drone. This includes an inertial measurement unit (IMU) with gyro, accelerometer and magnetometer, pressure- and ultrasound sensor for altitude measurements, and two camera: one mounted underneath the drone to measure optical flow to aid stabilizing the horizontal velocity dynamics, and one mounted on the front of the drone for general purposes. This assembly of sensors is particularly selected to be able to estimate the orientation of the drone and its altitude which is imperative for the velocity controller to function. The simulated drone features all the same sensors as the real one.

In addition to the altitude and orientation estimation that the quadrotor provides, we are interested in the downward looking camera which will be used to provide us information about the drones horizontal position. This camera provides a 60 FPS video stream of 320x240 resolution images up-scaled to 640x360. It is not clear from the documentation [56], but the field of view (FOV) of the camera appears to be 45°x 25.3°. This is a relatively small field of view for a camera, and puts restrictions on the feasible range for the controller proposed in the next section.

The physical drone is also equipped with an ad-hoc WiFi network. In this project we connect to the drone via this network such that we can receive sensor data, i.e. camera images, process them off-board and send velocity

commands back to the drone. This introduces roughly 100 milliseconds of delay to the controller loop, but has the advantage that the heavy computation – like training of neural networks – can be off-loaded to more capable hardware like a laptop or desktop computer. The extra latency can potentially be imitated to improve the accuracy of the simulated drone to the the real system, but this was not done.

### 3.2.3 Velocity controller

For the physical drone platform there are no documentation for how to the velocity controller is implemented. However, for the simulated drone the velocity controller is implemented as described in [1] which revolves around a quadrotor simulation available in the ROS package "hector_quadrotor". While this paper revolves around a different simulated quadrotor, it was made by the same creators that made the AR.Drone simulation used here, which can be found in the ROS package "tum_simulator". By visiting the documentation for the two ROS packages on the official ROS wiki website http://wiki.ros.org and their respective git repositories it can be confirmed that the same velocity controller is implemented in both of them.

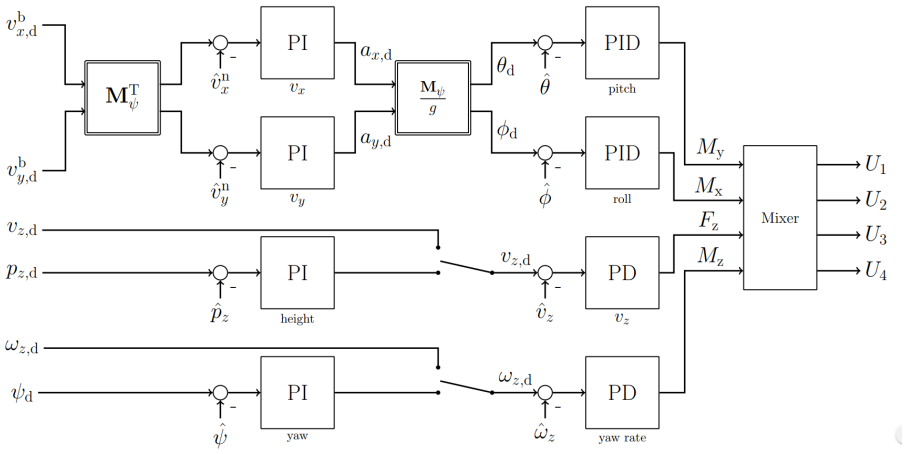


**Figure 3.2:** The cascaded PID controller structure used to implement the velocity controller in [1].

The implemented velocity controller follows a common designer concept

that makes the initial assumption that each axis can be controlled independently, and for motions with small deviations from the hovering state this assumption remains valid. As depicted in Figure 3.2, the inner control loop controls the attitude, yaw rate and vertical velocity, while the outer control loop controls the horizontal velocities, altitude and heading. The superscript 'b' in the diagram refers to the body frame. It can be seen from the figure that the desired horizontal velocities $\mathrm{v}^b_{x,d}$ and $\mathrm{v}^b_{y,d}$ are controlled indirectly by steering the pitch and roll angles of the drone. Figure 3.3 illustrates this by an example that shows how the drones velocity and orientation responds to a step response in $\mathrm{v}^b_{x,d}$ and $\mathrm{v}^b_{y,d}$, and also gives an indication as to how accurate the velocity controller is.

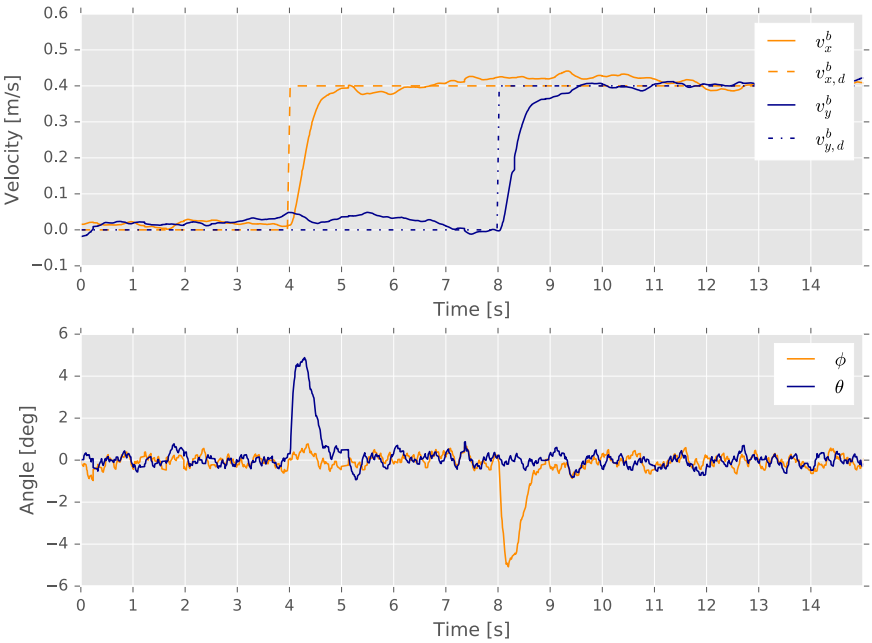


**Figure 3.3:** At $t = 4$ the drone receives a sudden step from 0 to 1 in the commanded x velocity, and at $t = 8$ commanded y velocity also jumps to from 0 to 1. To obey the new commands the drone tilt its corresponding axis – pitch or roll – to gain a horizontal force component, and to remain at the desired velocity it has to continuously "rock" the axis of rotation.

The AR.Drone simulator differ slightly from Figure 3.2 because the verti-

cal velocity and yaw rate can not be controlled by supplying the desired altitude $\mathrm{p}_{z,d}$ or desired heading $\psi_d$, only their derivatives $\mathrm{v}_{z,d}$ and $\omega_{z,d}$. To summarize, the velocity controller accepts a four-dimensional velocity command as

$$\mathbf{v}_d = \begin{bmatrix} \mathrm{v}^b_{x,d} \\ \mathrm{v}^b_{y,d} \\ \mathrm{v}_{z,d} \\ \omega_{z,d} \end{bmatrix}. \tag{3.2}$$

Even if the velocity controller of the physical– and simulated system are not the exact same it is not critical for the purpose of this project. What is significant is whether or not the motion of the quadrotor is accurate with respect to the real system. Unfortunately, there does not exist any scientific publication with regard to the accuracy of the simulator compared to the real system. However, judging by a video[1] posted by the creators of the simulation, it seems to be accurate to a certain degree.

Noise is included in the simulation and the drone therefore realistically drifts over time when hovering. The only stable states of the drone are the pitch and roll which are stabilized by the velocity controller. Figure 3.4 illustrates the drift of the simulated drone over a 1 minute time interval when hovering, i.e. when $\mathbf{v}_d = [0, 0, 0, 0]$, using the initial pose $[x, y, z, \phi, \theta, \psi] = [0, 0, 2, 0, 0, 0]$. The simulated drift was also observed to have a constant disturbance component which is randomized every time the drone is reset inside the simulation. One can imagine this as being a constant laminar airflow, and it affected the drone such that for every episode it was tougher for the drone to move in a particular direction.

The drone, which is only equipped with a velocity controller, is subject to drift because it relies only on sensors that has to be integrated in order to obtain position, and therefore error accumulates over time. As can be seen in Figure 3.4 the altitude, or the z-direction, is the direction in which the quadrotor drifts the least. This is because of the ultra-sound and air pressure sensor that enables the drone to measurement the altitude directly, which can be used to aid in the altitude stabilization control. The extra redundancy with respect

[1] "Gazebo Simulator for the Parrot AR.Drone quadrocopter" - Youtube

to measuring motion in z-direction explains why the drone is drifting less in vertical direction than in horizontal direction.

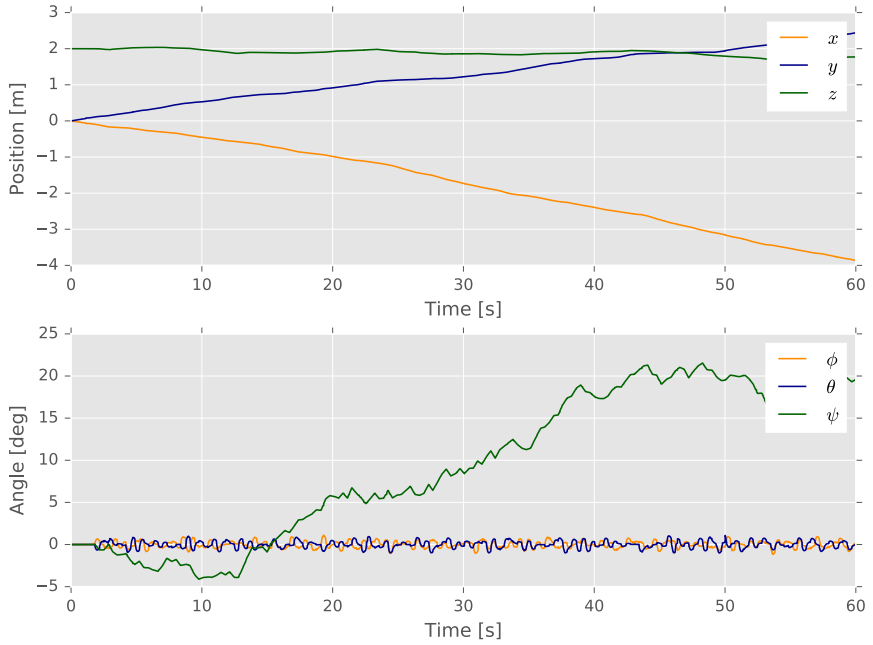


**Figure 3.4:** The drones position and heading drifts over time, while the roll and pitch angles are stabilized by the velocity controller. This data was gathered using a single episode, but represents the drift of the drone in general.

## 3.3 Dynamic positioning controller

Based on the thesis' goal outlined in Chapter 1 and the insight into reinforcement learning and computer vision provided in Chapter 2 this section proposes a dynamic positioning controller for quadrotor positioning relative to an object fixed to the ground plane. First the problem is formulated, then a high-level schematic of the controller is outlined before the output of the controller and state representation of the system is discussed. Lastly, a reward function is proposed for solving the particular problem in the reinforcement learning framework. The controller represents the main contribution of this thesis, which is to explore both the feasibility and capability of current state-of-the-art rein-

forcement learning methods to act upon uncertain state information to perform control tasks for quadrotors.

### 3.3.1 Problem formulation

Dynamic positioning is the task of maintaining the position of a vehicle relative to a setpoint. In this case the vehicle is a quadrotor, and we wish to maintain the quadrotors 3D position in space. We seek to do this by observing the position of an object through camera images, and the altitude estimated by the drones internal state estimator. The goal is for the drone to achieve the state where the object is located in the exact middle of the camera view, and the altitude error relative to some setpoint is zero. In reality we wish to control the drone such that it hovers directly above the object, but we use the aforementioned image based position of an object as a pseudo objective instead since the objective of maintaining the object in the center of the image is closely related to having the drone hover above the object in the real world. This was explained in further detail in Section 2.3.4. To define the scope of the controller a restriction on the initial condition of the problem is assumed: the object is assumed to be present in the picture when the controller is turned on.

### 3.3.2 Controller schematic

The controller presented in this section is primarily based on the DDPG algorithm presented in Section 2.1.9 to act as a high-level controller providing reference commands to the already existing velocity controller of the drone. To form a representation of the state of the system the controller uses the simple method for tracking the position of a distinctively colored object in images presented in Section 2.3.3.

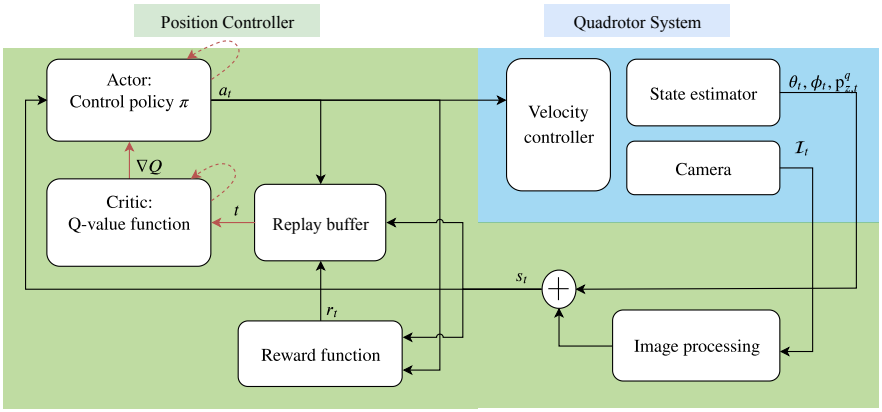


**Figure 3.5:** Block diagram visualizing the controller and feedback loop of the overall system. Marked in green is the position controller proposed by this thesis, while marked in blue is the parts of the quadrotor system which the position controller interacts with. The orange arcs represents the signals during training of the neural networks.

An overall structure of the system is provided in Figure 3.5. The black edges are the signals present in a single controller loop at time $t$. An image $\mathcal{I}_t$ is received by the controller, it is processed into a state $s_t$ which is used to compute a reward $r_t$ – the performance measure of the agent[2] – and compute a new action $a_t$ by the actor network which is used as new input to the velocity controller of the drone. The action, reward and both the current and next state is stored in the replay buffer as a transition $t = [s_t, a_t, r_t, s_{t+1}]$. Next are the orange arcs: the transitions in the replay buffer are used to train the neural networks that are used as function approximators for the Q-value function (Critic) and control policy $\pi$ (Actor). The gradient of the Q-network, $\nabla Q$, is used to train the actor network, and the dotted lined arcs represents the update to the networks weights.

### 3.3.3 Output

The action – or output – of the positioning controller is a velocity reference which is sent to the velocity controller. This controller will not attempt to con-

[2]Be aware that the term 'agent' is used interchangeably with the term 'controller' in this case.

trol the heading of the drone as it is sufficient to control the velocity commands $\mathrm{v}^b_{x,d}$ and $\mathrm{v}^b_{y,d}$ for horizontal control. The actor networks output is therefore the action

$$\mathbf{a}_t = \begin{bmatrix} \mathrm{v}^b_{x,d} \\ \mathrm{v}^b_{y,d} \\ \mathrm{v}^b_{z,d} \end{bmatrix} = \begin{bmatrix} \mathrm{v}^{cmd}_{x,t} \\ \mathrm{v}^{cmd}_{y,t} \\ \mathrm{v}^{cmd}_{z,t} \end{bmatrix}, \tag{3.3}$$

at time $t$, where the velocity command $\mathrm{v}^b_{z,d}$ is also included to control the quadrotors altitude. Here, the subscript 'cmd' is short for command. Instead of adopting the notation of [1] where the implementation of the velocity controller was outlined, we will use the notation above for convenience. This is because the commanded velocity is always given in the body frame and it is therefore implicit what frame the command exists in. And more importantly, we need a form that is able to convey information about what timestep a signal occurs.

Note that it is also possible to control the drone in the xy-plane by using the yaw rate command and one of the horizontal velocity commands, or both of them. Introducing an extra dimension to the action vector makes the problem harder to learn for the neural networks and so the latter option was dropped. There is no need to turn when you can simply move in any direction along the xy-plane using the commands in (3.3). Likewise, using only one of the horizontal velocity commands in combination with the yaw rate command was considered sub-optimal in comparison.

The drone is able to reach rather high velocities exceeding 10 m/s. For the purpose of this project however high velocities are unnecessary since the goal is ultimately to hover at still. Furthermore, higher velocity commands leads to bigger responses in the roll and pitch angles of the drone which affects the bottom cameras view of the object underneath it. This causes the objects position in the image to become unreliable as the image plane is no longer parallel to the ground plane, and it might even lose sight of it completely. Some speed is however required to achieve a satisfactory response of the controller and to counteract the drift, and the maximum commanded velocities was therefore restricted to 0.4 m/s.

### 3.3.4 State representation

To represent the state of the system the following state variables were available for use and experimented with during testing in simulation:

1. the position $\mathbf{p}_t^o = [\mathrm{p}_{x,t}^o, \mathrm{p}_{y,t}^o]^T$ of the object in the image relative to the center of the image at time t,
2. the velocity $\mathbf{v}_t^o = [\mathrm{v}_{x,t}^o, \mathrm{v}_{y,t}^o]^T$ of the object in the image relative to the center of the image at time t,
3. the altitude position $\mathrm{p}_{z,t}^q$ relative to a given setpoint at time t,
4. the velocity $\mathrm{v}_{z,t}^q$ of the quadrotor relative to the ground at time t,
5. the roll angle $\phi_t$ and pitch angle $\theta_t$ of the drone at time t, and
6. the action $a_{t-1}$ performed at the previous timestep t.

The position $\mathbf{p}_t^o$ is the centroid of a colored object in the image calculated via (2.47)–(2.48) in Section 2.3.3. In frames where the object is not visible in the image the position of the centroid is mapped onto the closest image wall from the previous image position of the object. The velocity of the object is calculated as $\mathbf{v}_t^o = \frac{\mathbf{p}_t^o - \mathbf{p}_{t-1}^o}{\Delta t}$ where $\Delta t$ is the reciprocal of the controller rate $f_c$. The altitude of the quadrotor, and its roll $\phi$ and pitch $\theta$ angles are estimated by the quadrotors internal state estimator. The velocity in the z-direction is calculated as $\mathrm{v}_{z,t}^q = \frac{\mathrm{p}_{z,t}^q - \mathrm{p}_{z,t-1}^q}{\Delta t}$. Similarly to the velocity commands always being in the body frame, the position of the object is always given in the image frame, and the altitude and its velocity always in the world frame. The idea behind this representation of the system is that the position and velocity observed in the image can be used for controlling the quadrotors x– and y position, and the altitude and velocity relative to the ground can be used to control quadrotors z position.

The state variables based on the image coordinates are visualized in Figure 3.6, where the position and velocity is given in the image frame coordinates. The image coordinates are scaled coordinates where the aspect ratio is conserved. They are normalized such that the shortest axis – the y-axis– spans

from -0.5 to 0.5, and in addition the coordinates are also moved to the center of the image by translation[3]:

$$\begin{aligned} \mathrm{p}^o_{x,t} &= \frac{u}{H} - \frac{1}{2}\frac{W}{H} \\ \mathrm{p}^o_{y,t} &= \frac{v}{H} - \frac{1}{2} \end{aligned} \tag{3.4}$$

where (u, v) is the pixel coordinate in the image, and (W, H) is the width and height of the image, respectively. With an image resolution of 640x360 the x-axis then spans from -0.888 to 0.888. The scaling was chosen such that the values of the image coordinates was in the same order of magnitude as the other state variables. This is a common pre-processing step for input data to neural networks, and is done to avoid large differences in parameter values that in return can cause instability during training.

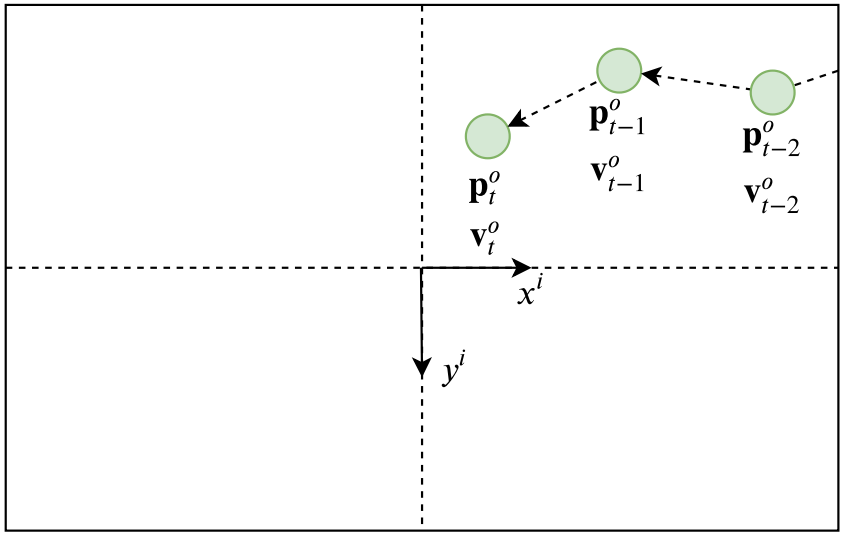


**Figure 3.6:** At each timestep the position $\mathbf{p}^o_t$ of the object – here represented by a green blob – in the image is derived, and its velocity $\mathbf{v}^o_t$ is calculated with respect to the position of the object in the previous timestep.

In the state augmentation of the DRL agent it is advantageous to include as much relevant information as possible such that it can efficiently infer about the

[3]Image coordinates usually have their origin in the top left corner, with the positive y-axis pointing downwards.

relationship between the future reward it experiences and the action it chooses in an arbitrary state. This advocates for the inclusion of the pitch– and roll angles, which are necessary for the reinforcement learning agent to infer about the position of the drone. For example, imagine a scenario where the object in the image is located at the center of the image, but the pitch– and/or roll angle is far off from zero. From this we could logically deduce that the quadrotor is in fact not hovering over the object, but rather is tilted such that the camera is pointing directly to the object. This topic was touched upon in Section 2.3.4 where we discussed how a reinforcement learning agent would be able to reason about the real position of the drone versus the position of an object in an image, which was based on a quadrotors dynamics and configuration of an attached camera.

Alternatively, the position in the image can be transformed onto the ground plane. Thus when the quadrotor tilts in order to move the position of the object will not change. This would be the equivalent of having a gyro-stabilized camera that always points in the direction of the gravitational field. However, it was decided to not do this because it injects noise in the observed position, and we do not know the quality of the roll and pitch measurements of the rather cheap quadrotor. In addition, the altitude of the real quadrotor was observed to have some drift during flight time and therefore the transformation would become exceedingly inaccurate over time.

The choice of using the previous action in the state vector to calculate new actions may seem strange at first, but it has an intuitive explanation. Since the velocity controller effectively stimulates the pitch and roll axis to achieve its the desired velocity knowing the previous action gives the agent information about how a new action will affect the roll and pitch. For example: the drone may be following the command $\mathbf{a}_t = [0, 1.0, 0]$, but this does not mean that the corresponding pitch angle is far off from zero. In fact, the same pitch angle may be observed when following the command $\mathbf{a}_t = [0, 0, 0]$. This can be seen from Figure 3.3. However, if the action is suddenly changed to $\mathbf{a}_t = [0, -0.1, 0]$, the response in pitch would be very different for the two cases, and therefore the inclusion of the state of the velocity controller, $a_{t-1}$, give valuable information to the agent.

In the state representation of the system the past velocity and action variables was used in addition to the current state variables $\mathbf{v}_t^o$ and $a_{t-1}$, such as $\mathbf{v}_{t-1}^o$ and $\mathbf{a}_{t-2}$. These variables were included to give the agent information about the constant disturbance like drift in the simulation mentioned before. This also has an intuitive explanation: by knowing how the past inputs have affected the system we observe evidence of how the current drift, or disturbance, affects the system, which in return can be used to counteract the disturbance.

### 3.3.5 Reward function

The reward function acts as a performance measurement for the agent, and effectively defines how it ought to act in its environment. The design of the reward function is therefore important, but unfortunately there is no golden rules for how to do this since the design depends entirely on the problem and the states available. Even though designing good reward functions require some experience, there is a few ground principles that can be followed. First of all we would like to avoid sparse rewards if possible as it slows convergence. Secondly, we would like to avoid too much complexity. For example, if the agent is given reward for several concurrent objectives it may make the reward signal more noisy and harder for the agent to understand, thus slowing convergence and training data efficiency. Another reason for avoid complexity is that we do not want to micromanage the exploration and behaviour of the agent too much because it may unintentionally hinder the agent from finding the optimal policy.

With this in mind we would like to choose a base reward function which is at maximum when the position of the object is in the center of the image and when the altitude error is zero. Additionally, we would like the reward to decrease uniformly in distance from the center. This ensures that the path of maximum reward is always a straight line from the current position to the goal. The Gaussian function

$$f(x) = ae^{-\frac{(x-\mu)^2}{2\sigma}} \tag{3.5}$$

was therefore used as a base reward signal for the agent, where $a$ controls

the amplitude of the function, $\mu$ is the expected value, and $\sigma$ is the standard deviation. For the reward function we want the function to be at maximum for $x = 0$, and we therefore set $\mu = 0$, choose $a = 1$, and with $x = \sqrt{{\mathrm{p}^o_{x,t}}^2 + {\mathrm{p}^o_{y,t}}^2 + {\mathrm{z}^q_{e,t}}^2}$ we get the reward function

$$r_t = e^{-\frac{{\mathrm{p}^o_{x,t}}^2 + {\mathrm{p}^o_{y,t}}^2 + {\mathrm{p}^q_{z,t}}^2}{2\sigma}}. \tag{3.6}$$

The parameter x represents the error of the position of the object from the image center and the altitude error relative to the setpoint. The idea here is that $x$ acts as pseudo euclidean distance error. This reward approaches one when all the errors goes to zero. This function is visualized in Figure 3.7 as a 2-D heat map with standard deviation $\sigma = 0.05$.

Note that there exists other functions with the same uniformly decreasing characteristic, and some of them were experimented with during training. They did however all give the same results, and none of them seemed to improve convergence or performance over the others. The Gaussian function was therefore chosen simply because it is intuitive to tune the parameters.

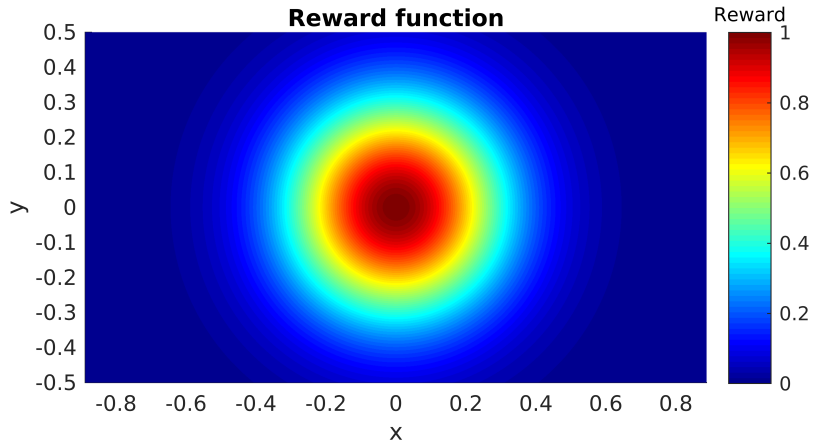


**Figure 3.7:** The reward function with $\sigma = 0.05$ visualized in the image frame as a heat map where the altitude here is set to zero as a constant. In 3-D the reward function is a sphere that gets progressively hotter – more red, higher reward – towards the center.

The choice of the amplitude of the reward function however was not arbi-

trary. If we know the maximum reward obtainable in a state $s_t$ we can compute the upper bound of the value function $V(s)$ given a discount factor $\gamma$ as

$$V(s) \leq \sum_{n=0}^{\infty} r_{max}\gamma^n = \frac{r_{max}}{1-\gamma}. \tag{3.7}$$

Hence, if $r_{max} = 100$ and $\gamma = 0.99$ then $V(s) \leq 10000$. This demonstrates that the value function and therefore the parameters of the neural networks can grow very large. We are therefore encouraged to limit the maximum reward the agent experiences in order to improve numerical stability and avoid large parameters in the neural networks. The use of L2 regularization in the DDPG algorithm was therefore omitted from the critic networks because it was difficult to tune in practice. No other hyper-parameters of the original DDPG algorithm was altered.

Chapter 4

# Simulations

In this chapter the results attained from training the proposed controller from Chapter 3 in the simulated environment will be presented along with discussion revolving around the results and the most crucial parameters and their importance for the system. The chapter starts of by introducing the training setup that was used to train agents, and then continues by presenting the results of using the proposed controller. Next, a controller based on the ground truth variables of the drone is constructed as a baseline and used for comparison and to investigate the potential of the DRL based controller design when imperfect information is taken out of the equation.

## 4.1 Training method

To train agents a simulated environment had to be constructed before the experiments could be carried out. The environment built and the training method used was constructed such that it would be feasible to reproduce the conditions the quadrotor experienced in the simulation in the real world. A ground marker for the image processing module to track had to be chosen, feasible initial conditions for the drone had to be determined, appropriate length of episodes for training had to be chosen, and a method for objectively evaluating the quality of the agents during training had to be found.

### 4.1.1 Environment

For the simulated environment it was desirable to make a simple environment that could be reproduced in the real world. It is also beneficial to have as few complex, visual objects in the simulation as possible because it reduces the speed of the simulation. Therefore a very simple world was used with only two objects: the drone and the marker on the ground. The marker used was a simple flat circular disk with a very distinct green color easily recognized by the image processing algorithm. This particular shape and color was chosen because it can easily be modeled in the real world by simply carving out a circular shape from green cardboard paper. Figure 4.1 shows a screenshot of environment configuration from the actual simulation along with the view of the bottom camera.

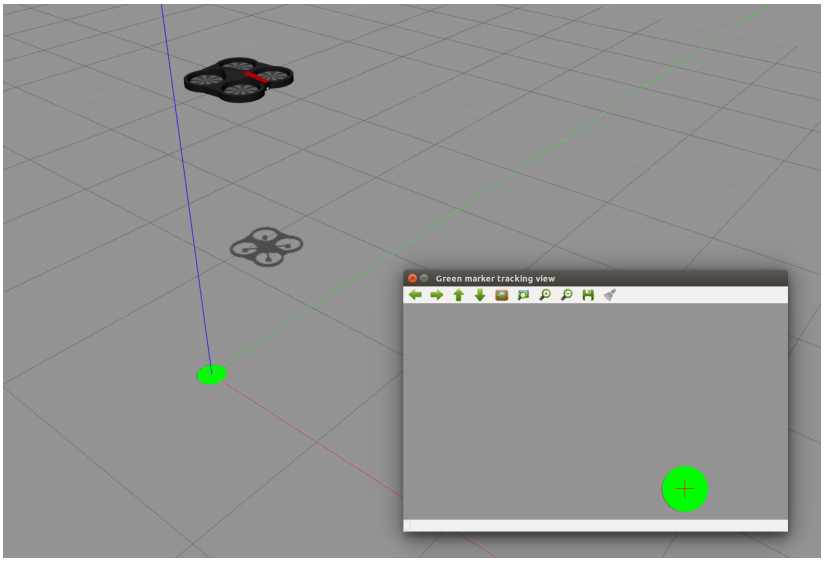


**Figure 4.1:** A screenshot showing both the simulated environment with the drone and green marker on the ground, and the view of the bottom camera of the drone. In the camera view the position of the marker is marked with a red cross showing that the object tracking method successfully detects and calculates the centroid position of the observed object. The red, green and blue rays emitted from the center of the object is the origin of the simulated world where the rays represent the x–, y– and z axis, respectively. These rays and the unit grid on the basic, grey colored ground is of course not visible to the camera, and are only shown for the viewers convenience.

### 4.1.2 Initial conditions

For starters, an initial altitude $z_0^w$ is picked from a random uniform distribution in the range $[1.5, 2.5]$ where we used $z_p^q = 2.0m$ as setpoint for the altitude of the quadrotor. Allowing higher altitudes would let the controller view more of the ground and hence giving it more room to navigate without loosing sight of the object. This would be advantageous considering the small field of view of the drone, but this height had to be restricted in order to be within a realistic height for testing in an indoor environment. Note that the dynamics of the object in the image is dependent on how close the quadrotor is to the object, and therefore we can not make the altitude setpoint in terms of relative coordinates such that it would work independent of the absolute altitude.

Given an initial altitude $z_0^w$ and the field of view of the camera we can calculate the range of possible horizontal starting positions by simple trigonometry such that midpoint of the object is visible in the image as

$$\begin{aligned} |x_o^w| &\leq z_0 \sin(\text{HFOV}/2) \\ |y_o^w| &\leq z_0 \sin(\text{VFOV}/2) \end{aligned} \tag{4.1}$$

where HFOV = 45° and VFOV = 35° are the horizontal– and vertical field of view of the camera, respectively. This is of course only valid if the world axis are aligned with the image axis, and therefore for simplicity we set the initial yaw angle $\psi_0 = 0$ because the yaw angle of the drone is irrelevant. This is because the both image axis and the body axis which the velocity commands are given in are rigidly attached to the drone itself. Lastly, we set the initial roll $\phi_0$ and pitch $\theta_0$ angles to zero as well.

To create a realistic initial condition we will add a last component to the mix: before the controller is turned on we will let the controller observe for at least one timestep while the velocity controller is following the command $a_0 = [0, 0, 0]$. This allows the controller to observe the initial drift of the quadrotor through the position and velocity of the object in the image, and gives us an initial value for the previous action, that is $a_{t-1} = a_0 = [0, 0, 0]$. The object – but perhaps not its midpoint – will still be visible in the image because the drift of the quadrotor is limited, and if it is not we will simply terminate the

episode, flag it as invalid such that the experience is not added to the replay buffer, and choose a new random initial horizontal position $(x_0, y_0, z_o)$.

Reminiscing back to Chapter 2 and Section 2.1.6 we know that the optimality of Q-learning is only guaranteed if the probability of visiting a state never becomes zero. While this do not follow follow directly when using function approximators, it is still beneficial to balance the content of the replay buffer such that agent continues to sample all possible states it may experience in the state space. Following the method outlined above we guarantee that the starting position of the object in the image can be anywhere within the image and that the agent continues to visit them perpetually. This also assures that the velocity of the object in the image is realistic with respect to how the controller would be activated in a real world scenario.

### 4.1.3 Length of episodes

In addition to the initial conditions for a training episode the length of the episode is also important to consider. To illustrate this lets consider how the agent ought to control the quadrotor in course detail as three main steps:

1. The agent starts to accelerate from the initial starting point towards the goal.

2. As it approaches the goal it has to slow down and stop at the goal.

3. The agent, now hovering above the goal, has to maintain this state indefinitely.

And now lets consider an extreme example: if we choose episodes to last forever, the agent – after it has learned the first two steps – will spend all of its time in the last step trying to maintain the goal position. This causes the replay buffer to be filled with the corresponding transitions which will eventually dominate the replay buffer completely. This is undesirable because the agent will start to over-train on this specific scenario and forget the other two scenarios. Therefore, to condition the contents of the replay buffer, a maximum episode length of 10s was used. The choice of this value is based

on what worked best in practice, but the overall training procedure was not particularly sensitive to this value.

An episode was also terminated if the altitude error was greater than 0.6m, or if the controller lost sight of the object in the image for more than 1 second. Allowing a few frames of "blindness" was useful because it allowed the agent to move more freely when the object was located at the edge of the image. This would be almost impossible otherwise because for the agent to be able to control the objects position towards the center of the image it has to tilt its camera away from the direction of the object.

### 4.1.4 Evaluation

During training different reward functions was experimented with. The cumulative reward does not necessary indicate how well the agent is doing because it can be hard to interpret, and comparing two agents with different reward functions based on how much accumulative reward they achieve does not make sense if different variables are used, or if the variables themselves are in different units. We therefore need a unified way of measuring agents performance relative to each other. Since we are training in a simulation we have the luxury of knowing the actual ground truth variables such that we can inspect how well the agent is doing with respect to the real objective, and not just the pseudo-objective. The error function used for this purpose is

$$\mathrm{P}_e = \frac{1}{N} \sum_{t=0}^{t=N} \mathrm{p}_e(t)^2 \tag{4.2}$$

where $\mathrm{p}_e(t)$ is the position error at time t, and N is the length of the episode. Both the position error of the pseudo objective and the position error measured in world coordinates – real objective – was logged. Note that while the agent seek to maximizes its cumulative reward, the accumulative error is here defined such that lower is better. This error was not used by the agent internally to train, but only used as a tool for the purpose of objectively comparing agents

Agents were evaluated periodically during training by performing an evaluation session where training was turned off and no exploration noise was injected into the output of the controller, i.e. the Ornstein-Uhlenbeck noise

used for exploration in the DDPG algorithm. This session consisted of four episodes where the drone was initiated in four different positions that were the same for each evaluation session. These positions were chosen as the four corners of the image with four different altitudes. Averaging over multiple episodes was done to verify that the agent was capable of handling different initial conditions and such made evaluations more consistent.

The evaluation episodes had a maximum episode length of 15s as opposed to the normal 10s for the training episodes. This was used to make sure that the controller successfully managed to hover over the object for extensive amounts of time.

In addition to the average error, an *evaluation reward* was also calculated, defined as the average reward of the four episodes. It is common to keep track of this reward because this is the objective the agent tries to maximize, after all, even if it is not the actual objective that we wish to optimize for.

## 4.2 Image position-based controller

This section is dedicated to presenting the results obtained from training the proposed controller in Section 3.3. Firstly, the main results from training the controller are presented, and an improvement to the initial proposed reward function is introduced, then a method that was used for improving training convergence time is described. Lastly, the controller loop rate and its influence of the controller performance is discussed separately. Discussion revolving the results follows directly as the results are being presented.

### 4.2.1 Main results

The results shown here uses the reward function that was presented in Section 3.3.5, and is repeated here for convenience,

$$r_t = e^{-\frac{{\mathrm{p}^o_{x,t}}^2+{\mathrm{p}^o_{y,t}}^2+{\mathrm{p}^q_{z,t}}^2}{2\sigma}}, \tag{4.3}$$

that is used together with a slight modification that is presented in the next section. The parameter $\sigma = 0.05$ was used, but the overall performance and

convergence of the controller however was not sensitive to this parameter. In conjecture with the reward function above the state vector

$$s_t = \begin{bmatrix} \mathbf{p}_t^o \\ \mathbf{v}_t^o \\ \mathbf{v}_{t-1}^o \\ \mathbf{v}_{t-2}^o \\ \mathrm{p}_{z,t}^q \\ \mathrm{v}_{z,t}^q \\ \phi_t \\ \theta_t \\ \mathbf{a}_{t-1} \\ \mathbf{a}_{t-2} \\ \mathbf{a}_{t-3} \end{bmatrix} \tag{4.4}$$

was used, which features all the available state variables presented in Section 3.3.4. Note that the position– and velocity of the object in the image and the previous action are vectors themselves, and the state vector therefore has dimensions 21x1.

### Training progress

After the agent had experienced roughly 5,000 simulation steps the agent started to exhibit satisfactory behaviour, but continued to make minor improvements until roughly 40,000 simulation steps. The training progress in terms of the evaluation reward is shown in Figure 4.2 where it is plotted versus the number of simulation steps experienced. This also shows that the trained controller has a very consistent performance from evaluation to evaluation. Once a trained agent hit around 5,000 training steps it stopped faulting, i.e. it never lost track of the object for more than 1 second or flew too high or too low such that an evaluation episode resulted in termination.

Training agents was quite fast and it took only about 5 minutes to perform 5,000 simulation steps which equals less than 30 minutes of real-time simulation time[1]. An Intel i7-8700k processor was used to train all neural networks

[1]In Gazebo the simulation can run faster than real-time. How fast depends on the caliber

in this project and to run the quadrotor simulation itself. No dedicated GPU was used.

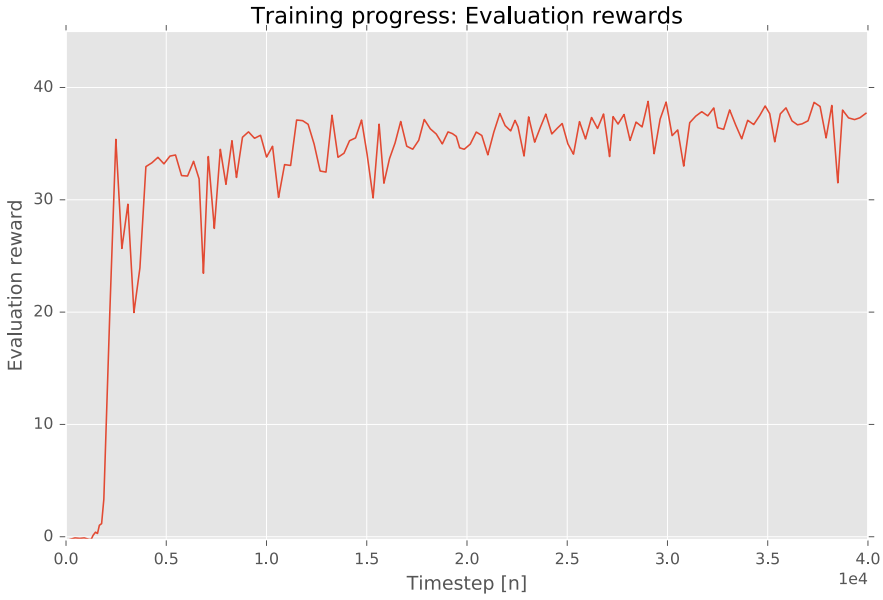


**Figure 4.2:** The evaluation reward plotted versus that number of simulation steps performed for the agent.

A controller rate $f_c$ of 3Hz was used for this agent, and because each evaluation episode lasted for 15s the maximum theoretical evaluation reward is 45. However, obtaining an evaluation reward of 45 was not possible in practice. This would require the agent to start at the setpoint in each evaluation episode – which as described in Section 4.1 is not the case – in addition to being physically capable of maintaining identical to zero positional error.

Next, let us consider the average position error for each evaluation depicted in Figure 4.3. This shows the pseudo objectives average positional error in the image given as the difference between the tracked objects position from the center of the image, versus the average horizontal position error of the quadrotor relative to the marker on the ground. This clearly illustrates that the

of the hardware that is running the program.

pseudo objective minimizes the real objective in practice. Note that the altitude error is not included in any of these errors as no pseudo objective is used for the altitude. This is because the altitude is measured directly by the quadrotor and is included in the state representation of the agent.

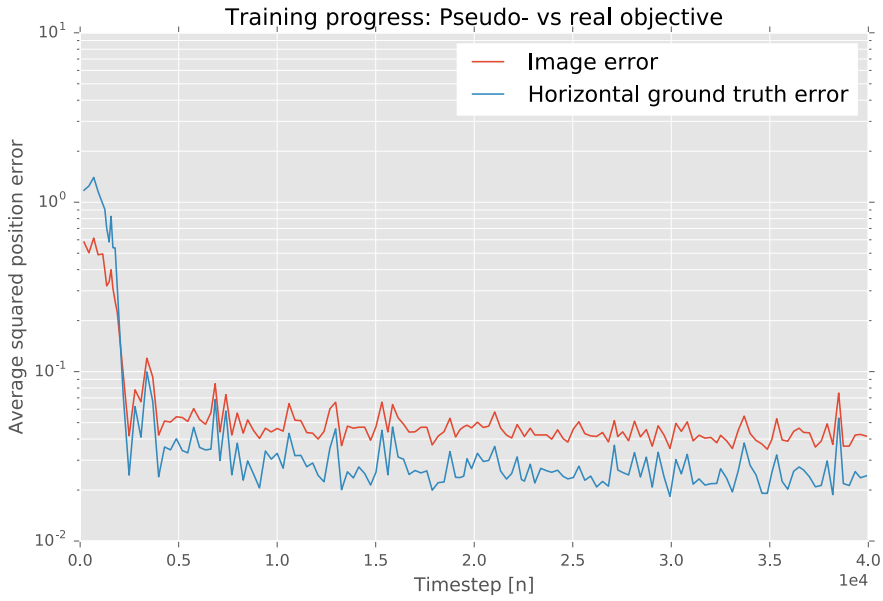


**Figure 4.3:** The average positional error for the pseudo– and real objective plotted versus the number of training steps performed. Note that the vertical axis is logarithmic.

### Image trajectory and position error

Figure 4.4 – 4.5 shows the trajectory of the object in the image and the positional error of the object in the image over time, respectively, for the four evaluation episodes taken from the best evaluation session in the same training session. Note that the image axes – as opposed to the world axes that are given in meters – are given in the normalized pixel coordinates $p_n$ as described in Section 3.3.4.

From these plots we can see that the trajectory of the object is quite volatile. This is partly due to the low controller rate, and because the small field of view

of the camera causes large motion in the image even for small offsets from zero in pitch and roll. For each of the trajectories one can also observe how the object at first appears to be moving away from the goal before it leaps towards it. These is because the camera is tilted away from the marker on the ground when the quadrotor itself starts to accelerate towards it, as discussed before.

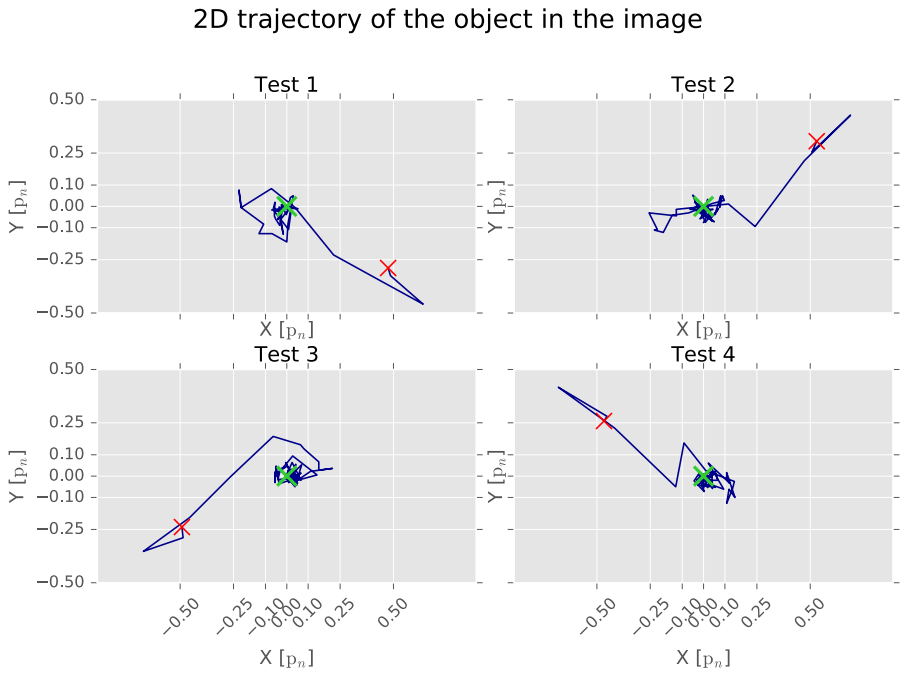


**Figure 4.4:** The trajectories of the object in the image frame during the four evaluation episodes. The red cross marks the starting position and the green cross marks the desired goal position.

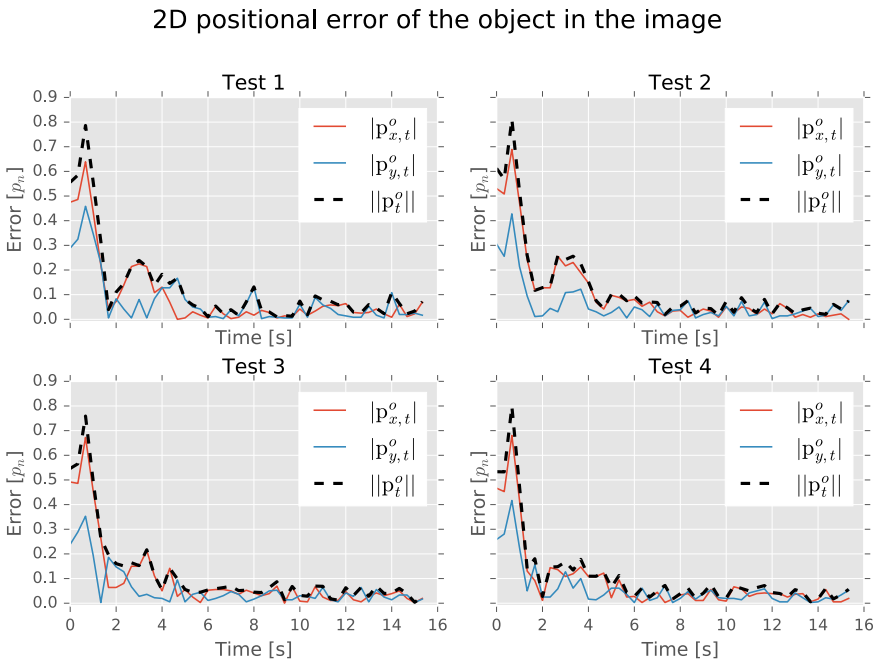


**Figure 4.5:** The positional error of the object in the image frame over time for the four evaluation episodes.

### Quadrotor trajectory and position error

In Figure 4.6 – 4.7 is the horizontal trajectory of the drone in the world frame, and the 3-D positional error of the drone with respect to the ground marker and altitude setpoint over time, respectively. In addition, Figure 4.8 shows the trajectory of the altitude of the quadrotor over time for the four test cases. Overall, these results make it clear that the agent, given an initial starting point where the object is visible in the image, successfully manages to obtain and maintain the desired 3-D setpoint.

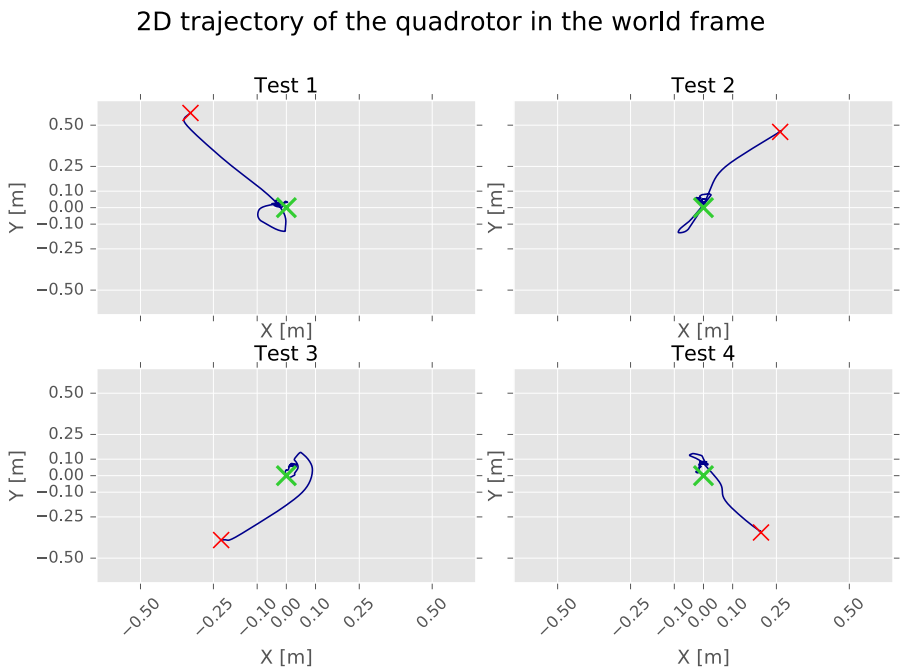


**Figure 4.6:** The horizontal trajectories of the quadrotor for the four evaluation episodes. The red cross marks the starting position and the green cross marks the desired goal position.

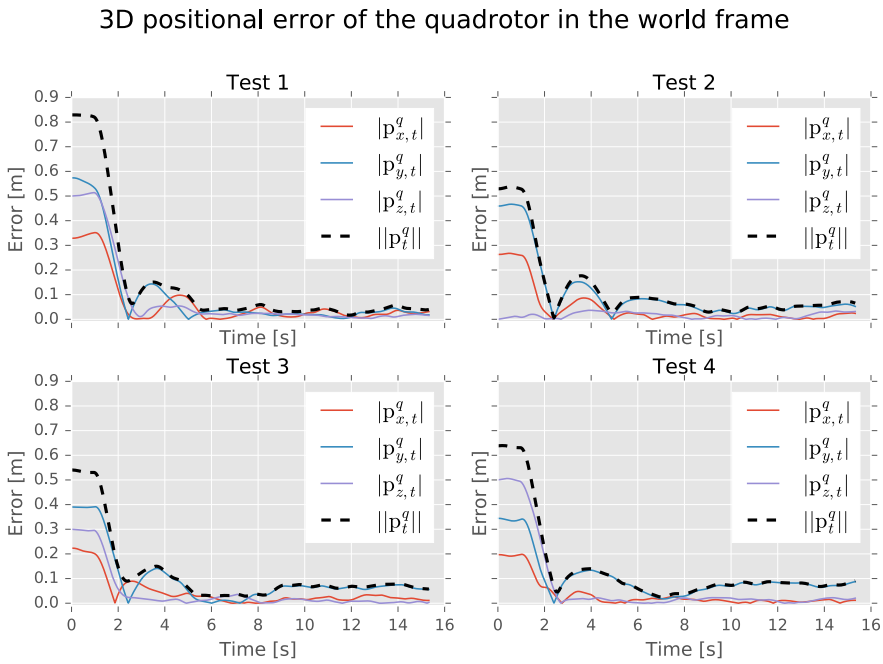


**Figure 4.7:** The positional error of the quadrotor in the world frame over time for the four evaluation episodes.

It appears that the controller is best at controlling the height of the quadrotor which evidently has the lowest average error of the three coordinate axes, as shown in Figure 4.7 and Figure 4.8. This may not be a surprise considering that the height is directly measured while the x– and y position is observed indirectly from the object position in the image. However, as shown in Section 3.2.3, the quadrotors drift in z-direction is also significantly lower than in horizontal direction. Furthermore, the simulation, as discussed in Section 3.2.3, was discovered to have a constant disturbance force which again affects the horizontal movement of the quadrotor more so than the vertical. This may indicate that the disparity in accuracy may also be a result of the fact that the velocity controller is actually more accurate in vertical direction.

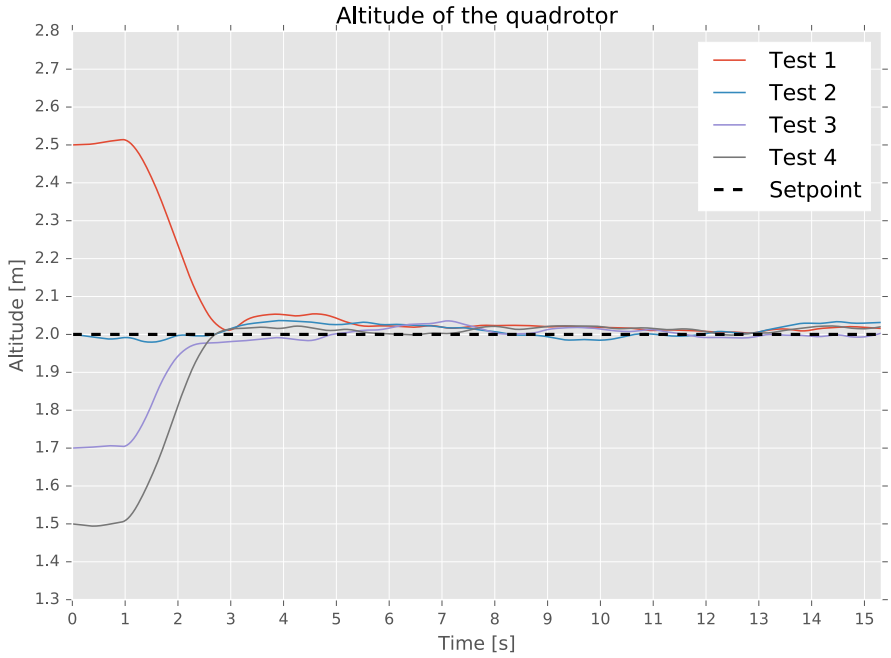


**Figure 4.8:** The altitude of the quadrotor over time for the four evaluation episodes.

Whether or not the velocity controller is more accurate in z-direction or not, some of the error in horizontal direction is still directly tied to the disturbance. This is because the controller has to position itself in a position where it can counteract the drift while positioning the object in the center of the image. Given how the dynamics of the velocity controller works this is of course only possible via a nonzero tilt.

In Figure 4.9 is the output of the position controller, i.e. the action that is sent to the velocity controller for the four test cases. By looking at the output for test case 1 we see an example how the controller manages to counteract the disturbance present in the simulation by maintaining a net positive velocity reference in the commanded y-velocity.

It is reasonable to think that the agents ability to counteract the disturbance is because of the inclusion of the past velocity and action states in the state representation, $\mathbf{v}^o_{t-1}$, $\mathbf{v}^o_{t-2}$ and $a_{t-1}$, $a_{t-2}$ respectively. Training agents without these previous state variables was also experimented with, but training would

often diverge very prematurely which is thought to be a direct of the disturbance in the simulation: if none of the state variables gives information about the current disturbance the data collected by the agent will simply appear to be noisy. But still, without these variables the training would still sometimes diverge, albeit later rather than sooner.

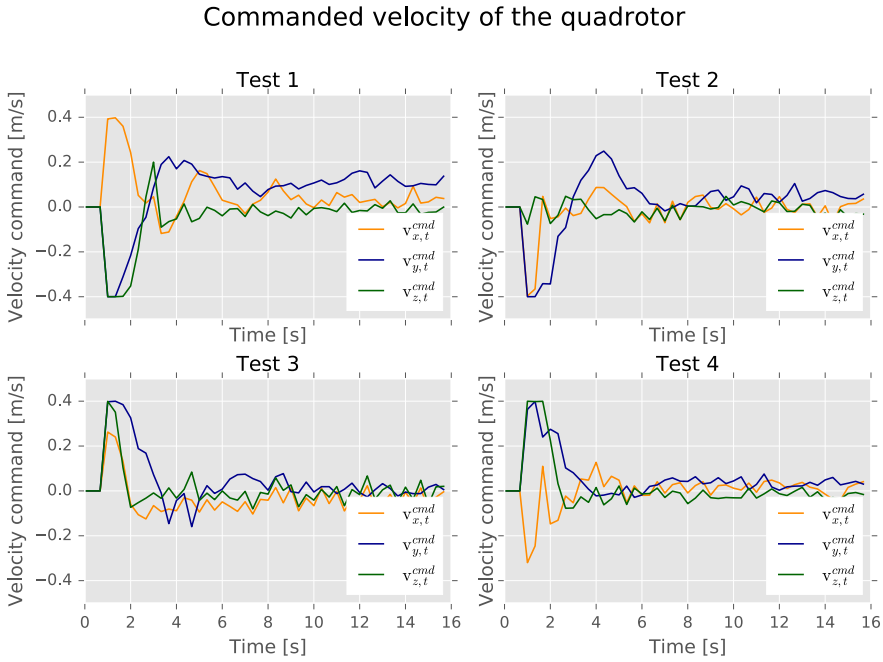


**Figure 4.9:** The proposed controllers output used as input to the velocity controller.

### 4.2.2 Penalizing controller change

The controller would under some training sessions learn a very noisy and oscillatory behaviour where it would shift its output in commanded z-velocity between maximum and minimum between consecutive timesteps. Not only would this kind of controller output tear on the durability of a real quadrotor system over time and consume more power, but it also makes the position of the object observed in the image more volatile because the actions directly affects the pitch and roll of the quadrotor. Therefore it is also desirable to deter

the agent from changing the commanded velocity in horizontal direction too fast as it directly affects the performance of the controller. To prevent this kind of control behaviour two penalties were introduced: on the change in action between consecutive timesteps, and in the absolute value of the action. The latter one was found to be effective for preventing unnecessary big leaps in the tilt of the quadrotor making the position in the image less accurate. This was was introduced in the reward function as

$$r_t = e^{-\frac{{\mathrm{p}^o_{x,t}}^2+{\mathrm{p}^o_{y,t}}^2+{\mathrm{p}^q_{z,t}}^2}{2\sigma}} - p_{\Delta a}|\Delta a| - p_{|a|}|a_t| \tag{4.5}$$

where $\Delta a = a_t - a_{t-1}$, $p_{\Delta a}$ is a constant positive scalar that functions as a penalty parameter for the change in action, $p_{|a|}$ functions as a penalty parameter for the absolute value of the action, and $\Delta t$ is the reciprocal of the controller rate $f_c$. The penalty parameters was set as $p_{\Delta a} = 0.05$ and $p_{|a|} = 0.02$, which gave good results in practice, but no exhaustive search was done to find optimal values for these parameters. Figure 4.10 shows the typical output of the controller before the action penalty term was augmented to the reward function.

It is also possible to penalize the observed velocity in the image, the offset in pitch and roll from zero, or the velocity in z-direction. However, since the change in commanded velocity directly affects the the tilt and the velocity variables, in addition to affecting durability and power consumption as previously mentioned, it was more effective to penalize the action as it is essentially the root of all evil in this context.

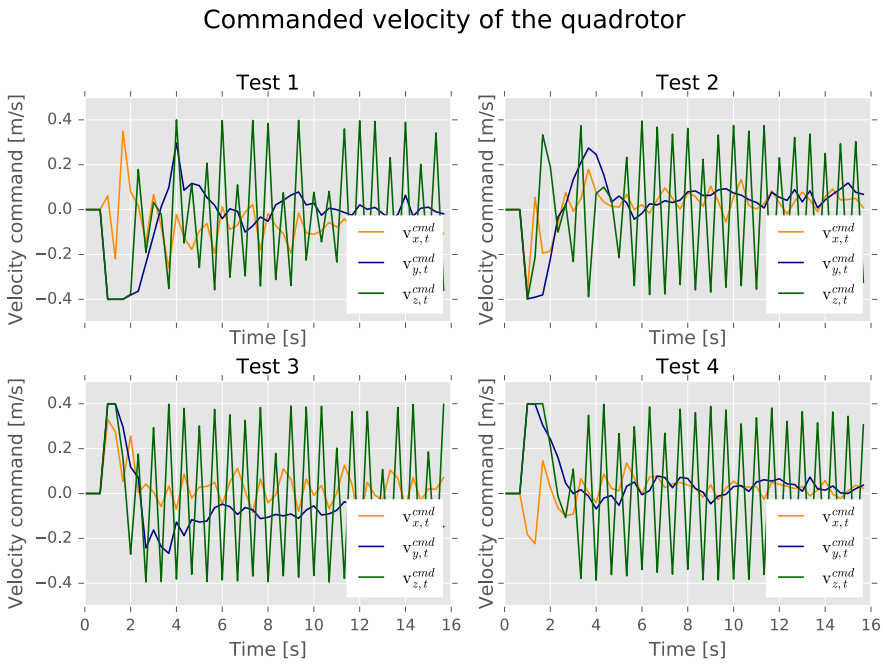


**Figure 4.10:** Agents trained without penalizing changes in the output would sometimes exhibit oscillatory behaviour in the output velocities, especially with respect to the z-velocity component.

### 4.2.3 Exploiting symmetry of dynamics

To improve convergence and data efficiency of the training procedure the symmetry of the quadrotor dynamics was exploited. To see this symmetry consider Figure 4.11: two quadrotors are arranged such that they in this moment are in equal, but opposite, horizontal distance from the goal – marked with a star for reference – and are approaching it with the same, but opposite, velocity and tilt, and from the same altitude. At the same time the marker at the ground would appear at the exact opposite side of the image, and have opposite velocity in the image. Producing these two states could also be done by sending the opposite velocity references $v^b_{x,d}$ and $v^b_{y,d}$ to the velocity controller. Hence, if we have one set of state variables we can reflect this vector about the z-axis into all four quadrants of the xy-plane and obtain three additional and equally valid set of states.

Since the two first velocity references in the action vector and the reward is also symmetric about the z-axis we can then therefore also produce three additional transitions which we can add to the replay buffer. So in summary, for each transition of states the quadrotor experiences it can add four transitions to the replay buffer. Note that the dynamics of how the marker moves in the image is not symmetric about the $x^w$– and $y^w$ axis because it is dependant how close the camera is to the marker, i.e. the altitude, else we would be able to produce eight transitions per actual transition experienced.

Figure 4.12 shows the training progress of the agent trained with and without the extra transitions acquired through symmetry. Although the final performance of the agent is the same, the agent reaches the point where it does not fail, i.e. loses sight of the object or flies too low or too high, much quicker. While this method was not of much use for the controller trained in simulation, it can be beneficial for an agent that has to explore the real world where time is crucial, and especially exploration in early stages can be tedious and be problematic with respect to safety.

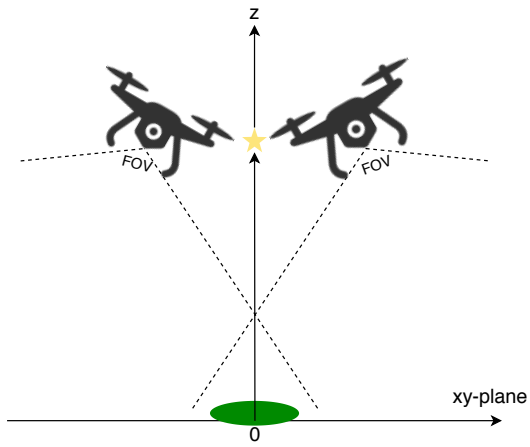


**Figure 4.11:** Two quadrotors approaches the goal from the opposite direction in the horizontal plane. By looking at the symmetry of the quadrotors dynamics we can see that the individual states of the quadrotors are opposite about the z-axis and can therefore be used to produce additional transitions for the DRL algorithm to learn from.

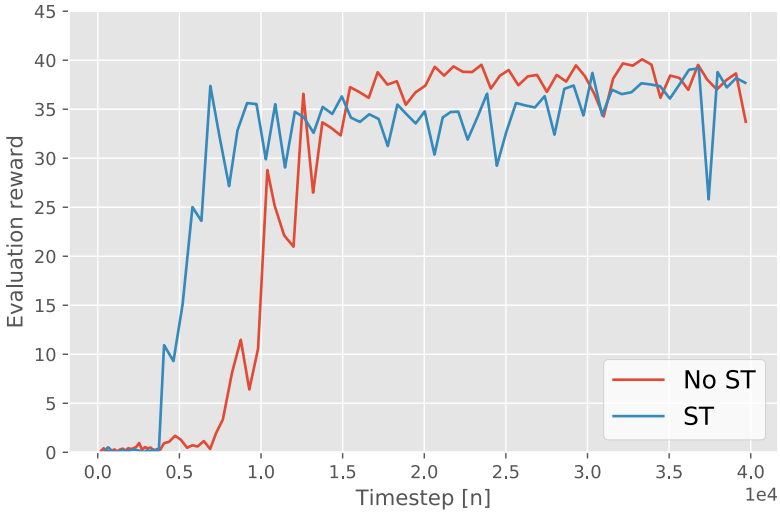


**Figure 4.12:** The training progress of the controller trained with and without using the method for generating multiple transitions per experienced transition. Here 'ST' stands for Symmetry Training.

### 4.2.4 Controller rate

The controller rate $f_c$ was set to 3Hz for the controller discussed here, as was stated initially. There was several reasons for why this rather slow control rate was chosen:

1. *Training the controller to an acceptable level of performance was slower with increased controller rate*

It seemed infeasible to adapt the controller to the real-world system if the required amount of training time was too high. Arguable the speed of convergence presented previously is more than fast enough for real-world adaption, but because we expected some missteps along the way some leeway with respect to time was planned for and contributed to the choice of a lower controller rate.

2. *The communication delay associated with the real quadrotor*

As stated previously in Section 3.2.2 the delay between the quadrotor and the off-board computer was roughly 100 milliseconds. Therefore a higher controller rate would lead to a worse ratio between the delay and controller rate making real-world adaption harder as it would affect the accuracy of the simulation to the real-world system.

3. *The overall performance seemed to dwindle in line with increased controller rate*

Normally one would expect better performance with higher controller rate, for this controller however that was not the case. While the exact reason for why this is not known, we speculate it to be because of the noisy behaviour of the position of the object in the image. It seemed difficult for the agent to not get stuck in a local optima where it would alter between the maximum and minimum commanded velocity to make the camera swipe over the marker at the ground. It was this kind of behaviour we observed before penalizing the change in actions between consecutive timesteps in Section 4.2.2. Unfortunately, no amount of penalty appeared to alleviate this problem as too much penalty would hinder the agent from moving at all and inhibit exploration.

Another approach that was tested was to increase the parameter $\gamma$ in the DDPG algorithm such that the agent would be more far-sighted. This was done to make the agent realize that initial "backwards" movement it observes when accelerating towards the goal (due to camera tilting in the opposite direction, as described previously) was necessary to obtain the more sought after – higher reward – setpoint. Unfortunately, this was also unsuccessful.

## 4.3 Ground truth-based controller

Even though the image based controller solves the problem studied in this project it is not clear how good the resulting controller is. To investigate the potential of the proposed positioning controller, and to better understand the prevalent horizontal error of the controller, a controller that used the ground truth horizontal variables was therefore constructed. This means that we in

practice will assume that a perfect state estimator system exists and gives us omniscient knowledge of the position of the quadrotor in the world.

This section first describes the alterations that was made to the original controller design presented in Section 3.3 and the training method outlined in Section 4.1 to realize this controller, and then the results obtained from training it in simulation are presented. Discussion regarding the results follows directly.

### 4.3.1 Modified state representation

The reward function first presented in Section 3.3.5 that was used with the image position based controller was modified to use the true horizontal position of the quadrotor such that we get:

$$r_t = e^{-\frac{{\mathrm{p}^q_{x,t}}^2+{\mathrm{p}^q_{y,t}}^2+{\mathrm{p}^q_{z,t}}^2}{2\sigma}} - p_{\Delta a}|\Delta a| - p_{|a|}|a_t|, \tag{4.6}$$

where $\mathrm{p}^q_{x,t}$ and $\mathrm{p}^q_{y,t}$ is the horizontal position of the quadrotor in the world frame, and $\mathrm{p}^q_{z,t}$ is the altitude of the drone. The penalty on controller changes are included, and the parameter $\sigma = 0.05$ as before. The variables are as usual given relative to the setpoint $\mathbf{p}^q = [0, 0, 2m]$ which serves as the origin in the eyes of the agent. In conjecture with the reward function above the state vector

$$s_t = \begin{bmatrix} \mathbf{p}^{q,\psi}_t \\ \mathbf{v}^{q,\psi}_t \\ \dot{\mathbf{v}}^{q,\psi}_t \\ \psi_{drift} \\ \mathbf{a}_{t-1} \end{bmatrix} \tag{4.7}$$

was used. Where $\mathbf{p}^{q,\psi}_t$, $\mathbf{v}^{q,\psi}_t$ and $\dot{\mathbf{v}}^{q,\psi}_t$ is the 3-D position, velocity and acceleration of the quadrotor in the world frame rotated with the yaw angle $\psi_t$ of the quadrotor, $\mathbf{a}_{t-1}$ is the previous action sent to the velocity controller as before. Lastly, the change in yaw angle from the previous timestep caused by the drift $\psi_{drift}$ was included in the state representation to negate the effect of the non-inertial frame created by the rotation applied to the other state variables. Note that because the state variables in (4.7) are 3-dimensional vectors, except the yaw drift, the full state vector has dimensions 13x1.

Because the velocity commands are given in the body frame and the goal position is in the world frame the position, velocity and acceleration of the quadrotor needs to be transformed into a coordinate system which rotates with the yaw angle of the quadrotor. Alternatively, the yaw angle could be included in the state representation in (4.7) such that the agent would have to learn this transformation by itself. The former option was however chosen as it was considered unnecessary to augment the state vector with the yaw angle since the transformation is known. The transformation is a simple rotation around the z-axis,

$$R_{z,-\psi} = \begin{bmatrix} \cos(-\psi) & -\sin(-\psi) & 0 & 0 \\ \sin(-\psi) & \cos(-\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.8}$$

which together with the position and velocity given in the original world frame gives us the position

$$\mathbf{p}_t^{q,\psi} = R_{z,-\psi}\mathbf{p}_t^q, \tag{4.9}$$

the velocity

$$\mathbf{v}_t^{q,\psi} = R_{z,-\psi}\mathbf{v}_t^q, \tag{4.10}$$

and the acceleration

$$\dot{\mathbf{v}}_t^{q,\psi} = R_{z,-\psi}\dot{\mathbf{v}}_t^q. \tag{4.11}$$

The new rotated frame is visualized in Figure 4.13.

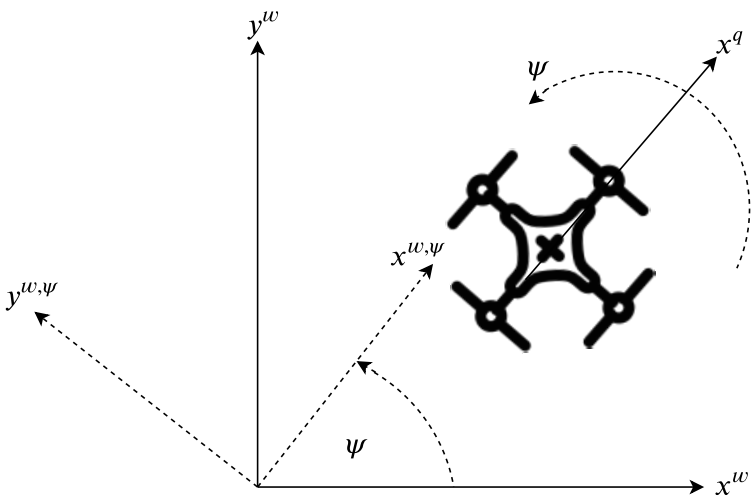


**Figure 4.13:** A new worldly coordinate system is created by rotating it with the yaw angle $\psi$ of the quadrotor. Here the new coordinate system is denoted $\{w, \psi\}$. This transformation is necessary to make the commanded velocity of the quadrotor and its position and velocity in the world frame independent of the yaw angle.

### 4.3.2 Adapted training method

The training method that was outlined in Section 4.1 was adapted when training the ground truth based controller with respect to length of episodes and the initial conditions. However, the requirements that the marker have to be visible in the image was no longer required. Instead the position of the quadrotor was initialized with an uniform distribution within an unit radius from the setpoint position, and episodes terminated if the quadrotors distance from the setpoint was greater than 1 + $\epsilon$ where $\epsilon$ was a small constant set to 0.1. The same starting positions as used with the image based controller was used in the evaluation session such that the two controllers could be compared fairly. In addition the same controller rate was used for the same reason.

### 4.3.3 Main results

#### Training progress

The training progress of the agent is showed in Figure 4.14 and the speed of convergence is on par with the image position based controller. This controller

did however stop improving after only 10,000 simulation steps, which is less than for the former controller. Note that while this agent acquired higher evaluation rewards, this is not directly comparable with the other controller because the units in the image and the world – normalized pixel and meters, respectively – are not the same.


**Figure 4.14:** The evaluation reward plotted versus that number of simulation steps performed for the agent.

Figure 4.15 shows the error in the pseudo– and real objective for the agent. Unsurprisingly, because this agent was trained using the ground truth variables the tight coherence between the two objectives has now diminished in contrast to what we saw for the image based controller.

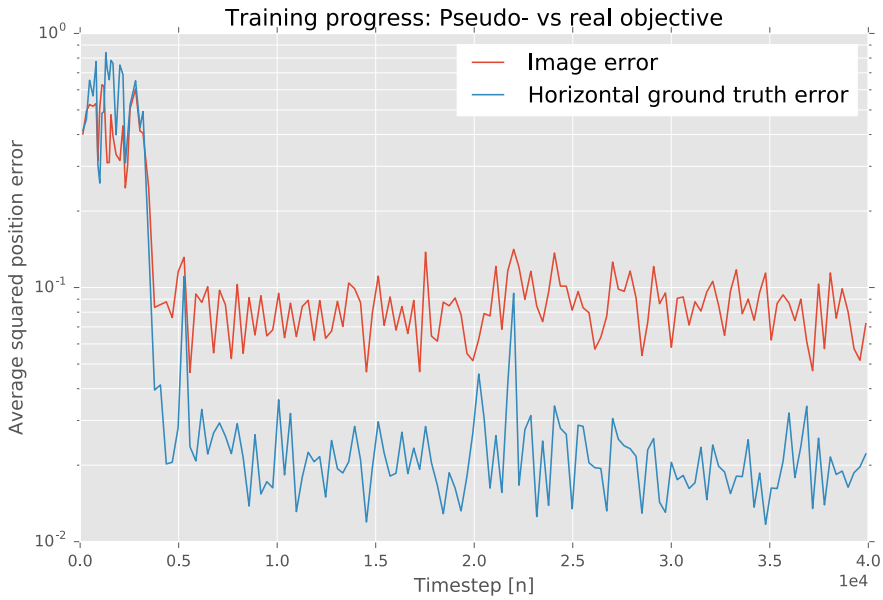


**Figure 4.15:** The average positional error for the pseudo– and real objective plotted versus the number of training steps performed. Note that the vertical axis is logarithmic.

### Image trajectory and position error

While this agent does not use the position from the image, it is still interesting to see how the position in the image behaves. Figure 4.16 – 4.17 shows that position is no longer stabilized in the middle of the image, as expected.

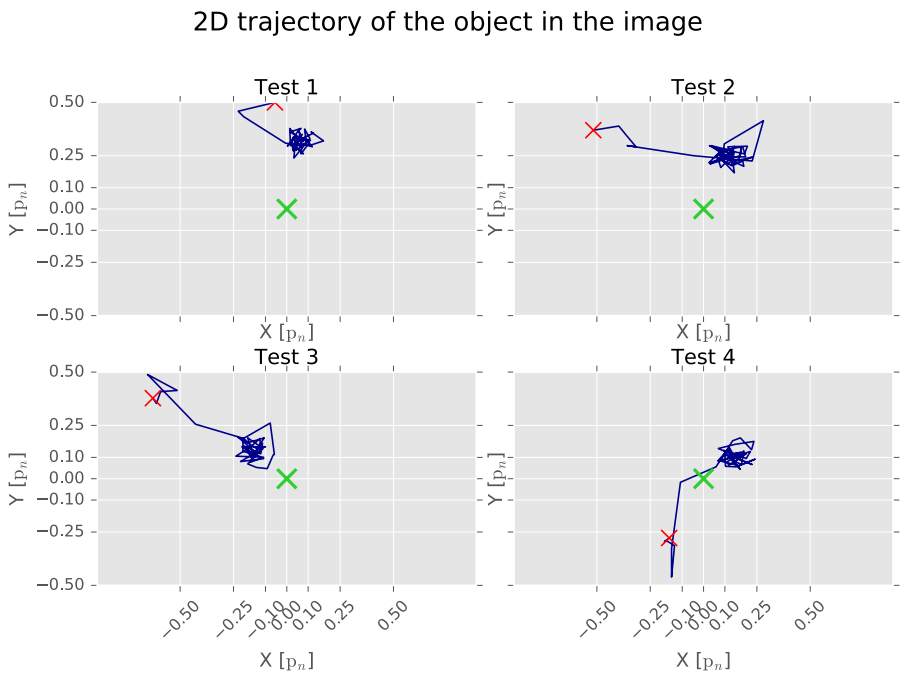


**Figure 4.16:** The trajectories of the object in the image frame during the four evaluation episodes. The red cross marks the starting position and the green cross marks the desired goal position.


**Figure 4.17:** The positional error of the object in the image frame over time for the four evaluation episodes.

### Quadrotor trajectory and position error

Figure 4.18 – 4.20 shows the trajectory of the quadrotor for the four test cases, the absolute error for the three coordinates and the euclidean error as well. It seems that the higher positional horizontal error affiliated with the image position-based controller was not due to the disturbance and drift being more prevalent in the horizontal direction as the positional error in all three dimensions are now virtually equal. It is now clear that the disturbance was the primary source of error because the agent had to maintain a nonzero tilt to achieve maximum reward. The altitude control performance does not seem to differ from the first controller at all, which is to be expected since they use the same state variables to reason about this objective.

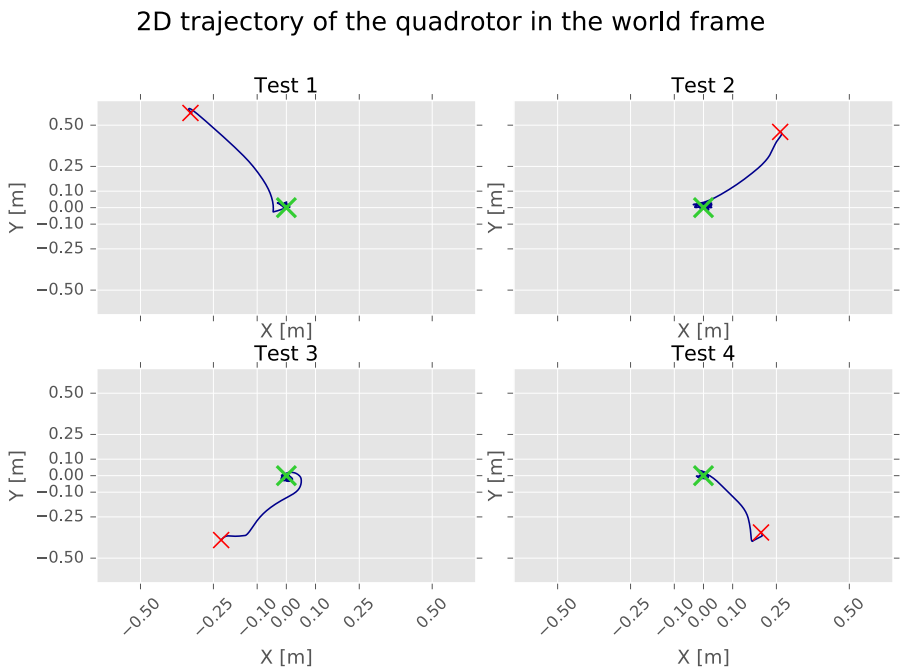


**Figure 4.18:** The horizontal trajectories of the quadrotor for the four evaluation episodes. The red cross marks the starting position and the green cross marks the desired goal position.

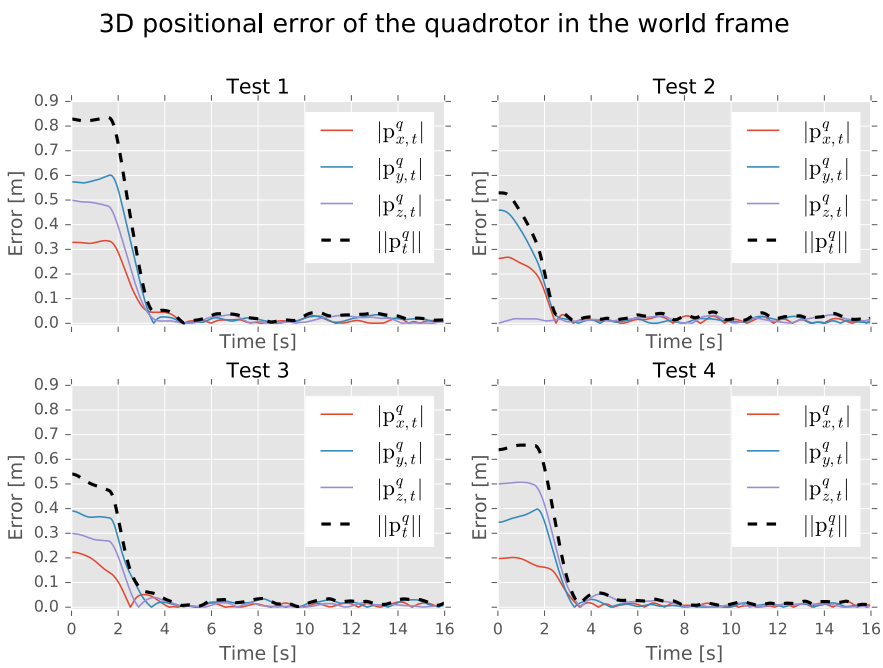


**Figure 4.19:** The positional error of the quadrotor in the world frame over time for the four evaluation episodes.

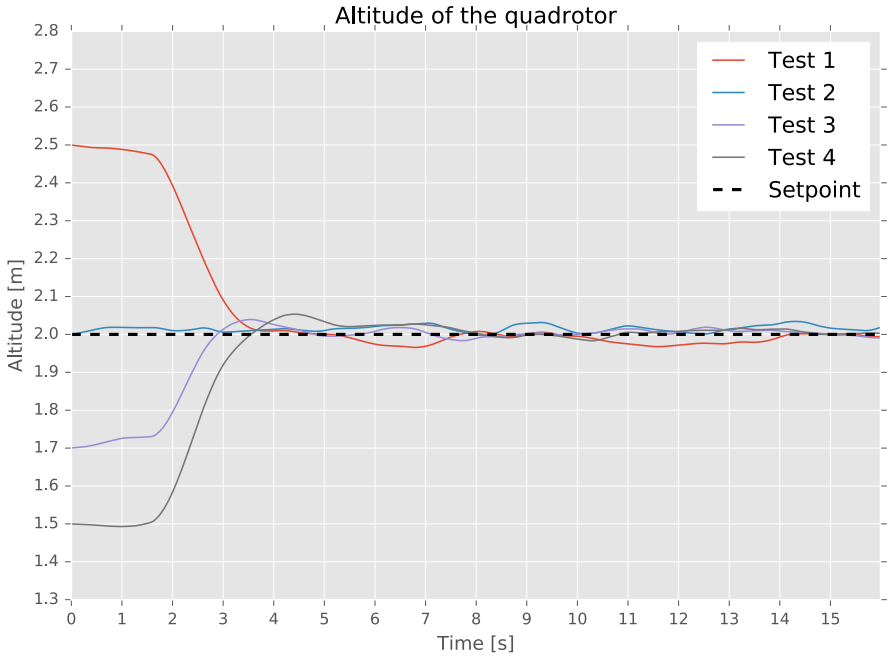


**Figure 4.20:** The altitude of the quadrotor over time for the four evaluation episodes.

As a last note it should be mentioned that the agent based on the actual horizontal position of the quadrotor in the world was able to learn the task even for higher controller rates, as opposed to the image position-based controller which struggled severely. However, similarly to the other controller the time it took to learn the task increased dramatically. The performance did not benefit from it either, but remained the same. To reduce the scope of this thesis the issue was unfortunately not explored further.

## 4.4 Comparison

Until now we have only looked at the accuracy of the two controllers presented in Section 4.2 and 4.3 for the best evaluation sessions of the two controllers, which does not necessarily tell us how accurate the controllers are on average.

To more precisely examine the accuracy of the controllers, and the ability of the controllers to maintain the hovering setpoint, a test was conducted

where both controllers were run for 100 trials of 60 seconds. The quadrotor was initialized on the setpoint each time, which is $(p^q_{x,d}, p^q_{y,d}, p^q_{z,d}) = (0, 0, 2)$ as before. The position $(p^q_x, p^q_y, p^q_z)$ of the quadrotor in the world frame during all of tests were recorded and a final mean and standard deviation (STD) was calculated. Snapshots of the agents' network weights at the time they performed the best were kept such that the exact same neural networks could be used in these trials as in the results that have been presented earlier. Multiple trials was conducted instead of letting the controller run the same amount of time in a single trial because the disturbance force in the simulation is reset when the simulation is reset, as explained in Section 3.2.3.

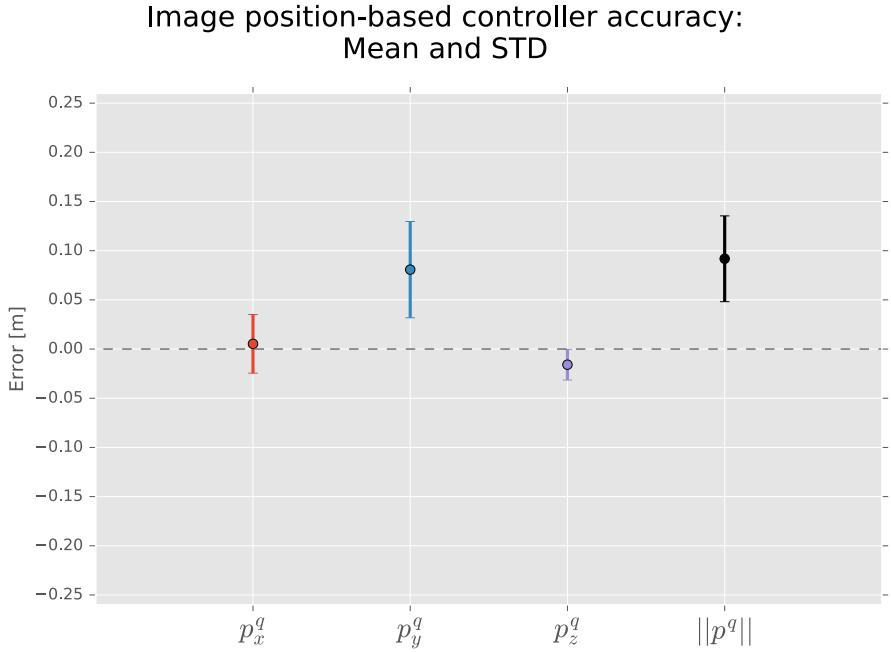


**Figure 4.21:** The image position-based controllers position error in the world frame relative to the setpoint visualized via the mean error (circle) over 100 trails of 60 seconds duration each. The bars spanning out from the mean is the standard deviation of the error.

Figure 4.21 – 4.22 shows the results for the image position controller, which mean magnitude error was ca *9cm*, and the ground truth based controller which mean magnitude error was ca *5cm*, respectively. It is clear that

the horizontal position error for the image position based controller is higher both in mean and standard deviation, which also mean that the magnitude of the error is greater in both mean and standard deviation as well. The altitude error is virtually identical for the two controllers.

For the image position-based controller it is important to note that the particular mean errors presented here is only indicative of the general magnitude of the mean. The controller trained with the same parameters could for example end up having a negative mean error in y-position instead of a positive one.

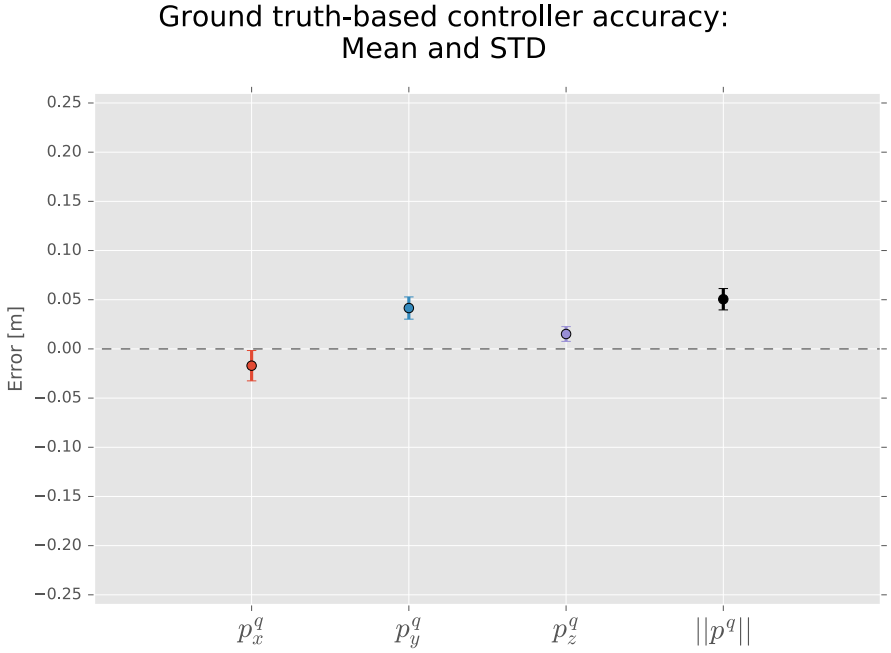


**Figure 4.22:** The ground truth-based controllers position error in the world frame relative to the setpoint visualized via the mean error (circle) over 100 trails of 60 seconds duration each. The bars spanning out from the mean is the standard deviation of the error.

The ground-truth based controller had the potential of converging to better solutions because its overall performance was not limited by the disturbance force. As can be seen the in Figure 4.22, the z-positional error is slightly less than in horizontal direction, which presumably is because the drone drifts

slightly less in z-direction. The error in x– and y direction is not the same in this case either, but there is no reason to think that the drones velocity controller is more accurate in one direction over another in this case. The issue is more likely due to the fact that training the neural networks, what the agent explores and the transitions it experiences is a stochastic process, and therefore it is unlikely that the agent will converge to a solution where it is equally proficient at reducing the error in all three coordinates at the exact same moment in time. This issue concerns both of the controllers presented here.

Chapter 5

# Real-world adaptation

The controller proposed for the Parrot AR.Drone 2 in Chapter 3 and tested in its Gazebo simulation in Chapter 4 was designed such that it would be realistic to apply the controller and training method to the real quadrotor system. In this chapter the training method that was used is presented along with the main results obtained from testing the simulation-trained before and after real-world training

## 5.1 Training method

The simulated environment was replicated in the real world – which was designed for replication in the first place – and the image processing method was tuned such that the specific color of the marker could be recognized. The real quadrotor was discovered to have two significant discrepancies with respect to being accurately simulated in the simulation, and a new method for exploration had to be constructed.

### 5.1.1 The environment

The simulation environment was recreated in a safe laboratory environment that is equipped with a safety net in case of unruly behaviour. A picture of the setup is shown in Figure 5.1 where a screenshot of the quadrotors view is

embedded which gives an impression of how narrow the field of view of the bottom camera is. The color rendering was not particularly accurate either, but because the environment was under such controlled conditions it was possible to tune the image processing method to recognize the object. To find the correct HSV values that isolates the ground marker from the rest of the image a tool from the Python package *imutils* named "range-finder" was used which allows an user to manually tune the range of minimum and maximum HSV values of a picture.


**Figure 5.1:** The simulated environment was recreated in the real world in a secure laboratory by carving out a green cardboard circle on the floor. The embedded picture shows the drones view and the red cross marks the spot of the centroid of the object as before.

During testing we unfortunately did not have access to an external positioning system, and such the state variables that are used and shown in the next session are all from the quadrotor and its internal state estimator. We therefore had no means to accurately measure the true world position of the quadrotor to check the accuracy of the controller.

### 5.1.2 Discrepancies of the simulation

With respect to how accurate the simulation was to the real-world system two significant discrepancies was discovered. The first one is related to the response of the quadrotor to velocity commands, and more specifically, the magnitude and how it affected tilt of the quadrotor. A test was conducted to analyze how the pitch and roll angles reacted to a step response in the commanded velocity for both the drone in the real world and the simulated drone. The results are shown in Figure 5.2 – 5.3, and it is clear that the real quadrotor has a much more aggressive pitch angle response than the simulated drone.

Additionally, we can now see a side-effect of the simulations lack of aerodynamic forces. While the real quadrotor maintains a nearly constant and non-zero pitch angle to maintain its velocity, the simulated quadrotor only tilts its pitch axis for a short second before it goes back to zero. This is because the real-quadrotor experiences drag forces which must be counteracted by maintaining a non-zero horizontal force component.

The displacement of the real quadrotor in horizontal direction was also considerably bigger during the test. The real quadrotor travelled approximately 4m during the two seconds it was following the commanded velocity, and the simulated drone travelled roughly 1m. So for the real drone the numerical values of the velocity command did not actually represent how fast the drone was actually moving.

Because the real quadrotor travels faster and exhibits larger tilts that directly affects how the position of the object behaves in the image, the maximum commanded velocity was scaled down to 0.075m/s to accommodate for the differences observed.

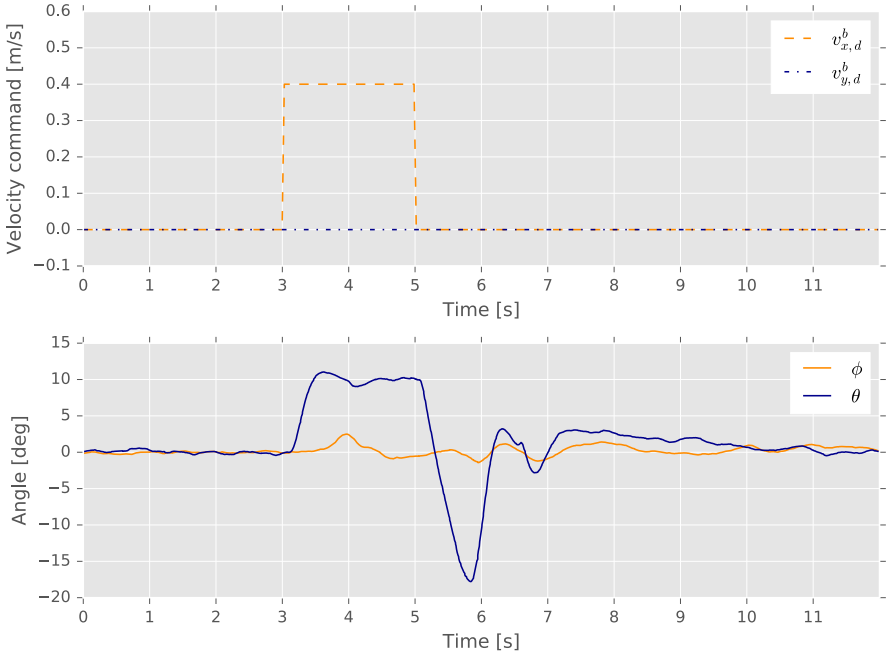


**Figure 5.2:** Real-world quadrotor: A step response test for the velocity controller where it receives the command $v^b_{x,b}$ = 0.4m/s at time t = 3s, which goes back to zero at t = 5s. The other velocity commands were set to zero, but $v^b_{y,b}$ is shown because it is directly related to the roll angle $\theta$.

The second discrepancy is related to the drift of the velocity controller. In the simulation this drift acted more like a constant disturbance force with a particular direction while being noisy at the same time, as discussed before. In the real-world however, the drift did not affect how difficult it was to move in the opposite direction of the drift, which of course is how drift should and do work. In addition to this, the real-world drift was discovered to be much smaller than the simulated one. This means that the drone was actually more stable than its simulated counterpart when hovering at still.

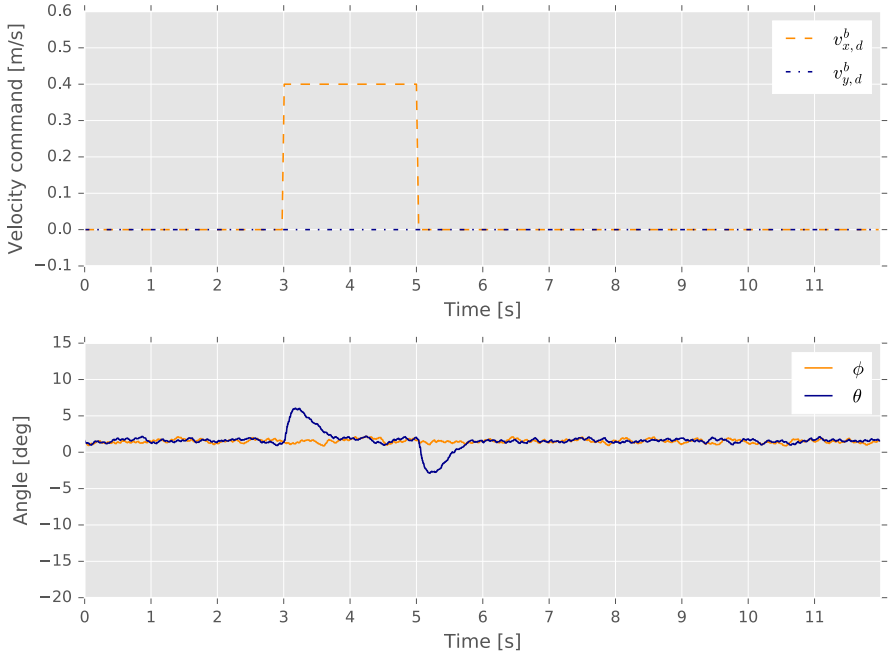


**Figure 5.3:** Simulated quadrotor: A step response test for the velocity controller where it receives the command $v^b_{x,b}$ = 0.4m/s at time t = 3s, which goes back to zero at t = 5s. The other velocity commands were set to zero, but $v^b_{y,b}$ is shown because it is directly related to the roll angle $\theta$.

### 5.1.3 Exploration method

It was desirable to adopt a strategy that allows the agent to explore the state space it operates in with minimal human intervention and supervision. For example, if a human operator has to constantly stop and manually set initial conditions for the agent to start from, the time spent training and gathering data to the replay buffer would be reduced. Not to mention that this is a tedious task in itself. This scenario is particularly unfavourable because of the short discharge time of the batteries of the Parrot AR.Drone, which lasts for only 12 minutes of flight time and only four batteries was used for this project.

To facilitate autonomous exploration a method that progressively increased the magnitude of the noise component over time was used, and when the positional error reached a certain threshold the noise was turned off for several

seconds. This can be stated as:

$$n = \begin{cases} (1 + C_g t_{on})\mathcal{N}, & \text{if } e < E_{max} \text{ and } t_{off} \geq T_{sw} \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

where $n$ is the noise component that is added to action (output) by the actor network, $C_g$ a constant that controls how fast the magnitude of the noise grows, $t_{on}$ is the timesteps since the noise was previously switched on, $t_{off}$ is the timesteps since the noise was previously switched off, $\mathcal{N}$ is the exploration noise process as in the original DDPG algorithm, and $T_{sw}$ a constant that determined how long the noise would be switched off before it was switched on again. The error $e$ is a vector consisting of the three objective variables, i.e. the horizontal position in the image and the altitude of the quadrotor $\mathrm{p}^o_{x,t}$, $\mathrm{p}^o_{y,t}$ and $\mathrm{p}^q_{z,t}$, respectively. The maximum allowed error before the noise was switched off was controlled by the constant $E_{max}$.

The idea behind this method is that the noise will force the quadrotor away from the setpoint and force the agent to experience all parts of the state space, and in the process balance the contents of the replay buffer. The noise was turned off to allow the quadrotor to recover when it was about to lose sight of the object.

## 5.2 Main results

In this section, the main results obtained from training and testing the controller in the real world environment is presented. First, the performance of an agent that was only trained prior in the simulated environment is shown, and then the performance of the agent after training is presented. In the end some potential sources of error are discussed. The agent used here is the one that was presented in Section 4.2 based on horizontal positioning via the observed position of an object in images. Discussion revolving the results follows directly.

### 5.2.1 Performance prior to training

To test the agents in the real-world the quadrotor was manually flown to an adequate position via keyboard control such that the object was roughly in the right side of the image and the altitude was circa 1.8m. Figure 5.4 – 5.7 shows the very first test was conducted with the agent where it had trained exclusively in the simulated environment prior to testing. The tests had a duration of 15s similar to the tests conducted in the simulated environment.


**Figure 5.4:** The initial real-world agent: Trajectory of the object as seen in the camera images during the test. The red cross marks the starting position and the green cross marks the goal position as usual.

Although the agent was not exactly excellent at controlling the quadrotor, it was not completely awful either. The agent clearly appears to have gotten the gist of it, but seems to have problems with braking, either too much with respect to the altitude, or too little with respect to the object position in the image. However, the performance was quite suitable for training. The initial agent was able to train almost completely autonomously using the exploration method outlined in 5.1.3, and about the only human intervention required was changing the battery every 12 minutes or so.

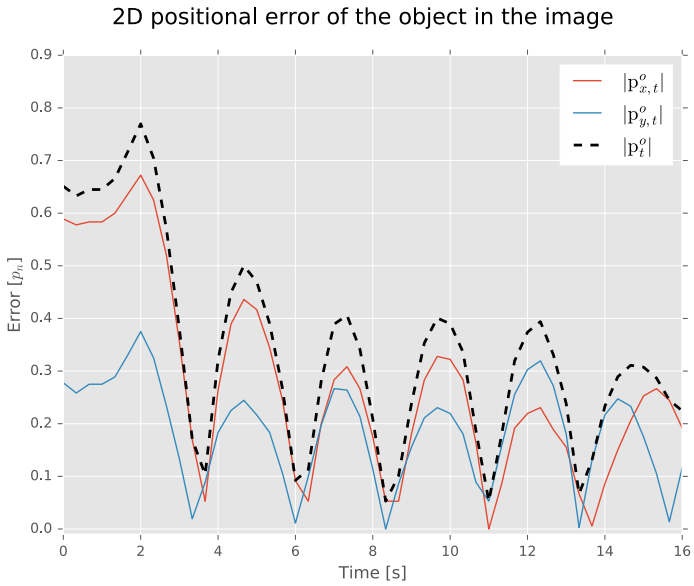


**Figure 5.5:** The initial real-world agent: Positional error of the object relative to the center of the image during the test.

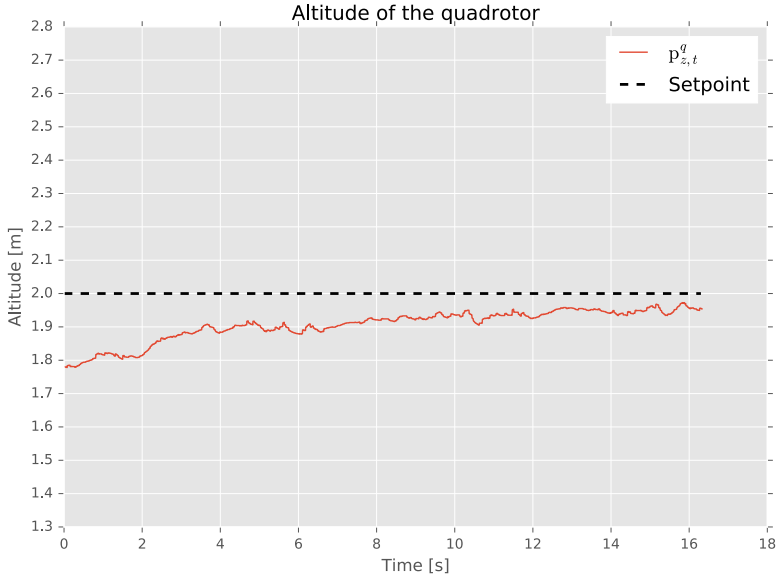


**Figure 5.6:** The initial real-world agent: The height of the quadrotor during the test.

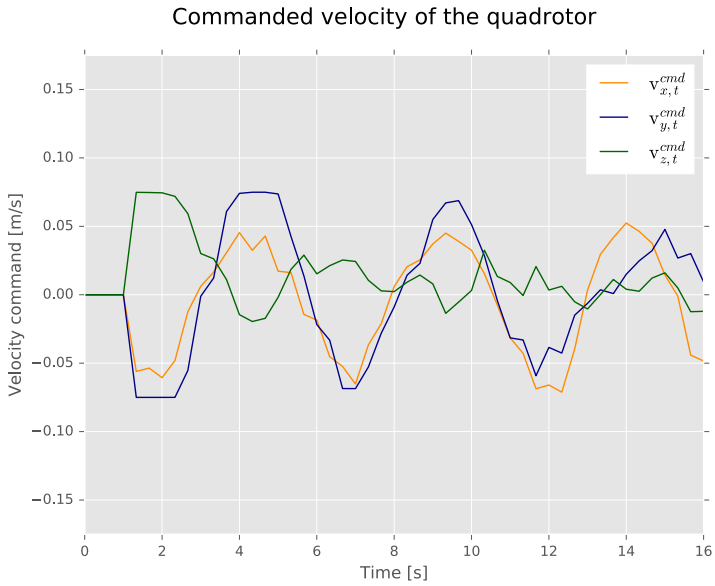


**Figure 5.7:** The initial real-world agent: The velocity references sent to the quadrotors velocity controller from the position controller during the test.

### 5.2.2 Performance post-training

The agent was trained in the real world for 10,000 steps (roughly 60 minutes of real-world training time), and the results for the test are presented in Figure 5.8 – 5.11 which shows that the agent has made significant progress. While it did not perform as good as it did in the simulation it was quite competent at controlling the position of quadrotor considering how narrow the field of view is, even though the altitude control did not improve much. Unfortunately, because of time constraints the agent was not trained further than this.

As previously stated, no method for logging the actual horizontal position was available. However, a short video was shot to demonstrate the agents ability to track the object. It can be found as an attachment to this thesis, and shows that the agent is able to track, and follow, the target as it is being displaced by a human adversary.


**Figure 5.8:** After training: Trajectory of the object as seen in the camera images during the test. The red cross marks the starting position and the green cross marks the goal position as usual.


**Figure 5.9:** After training: Positional error of the object relative to the center of the image during the test.

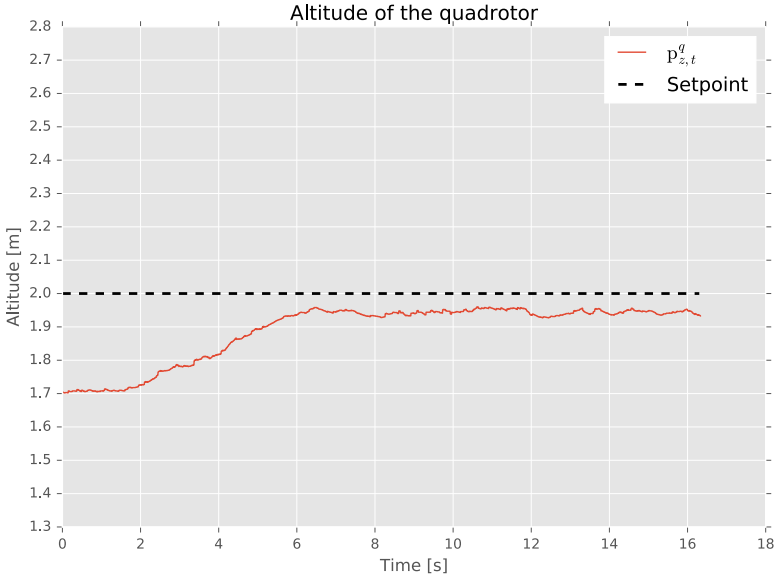


**Figure 5.10:** After training: The height of the quadrotor during the test.

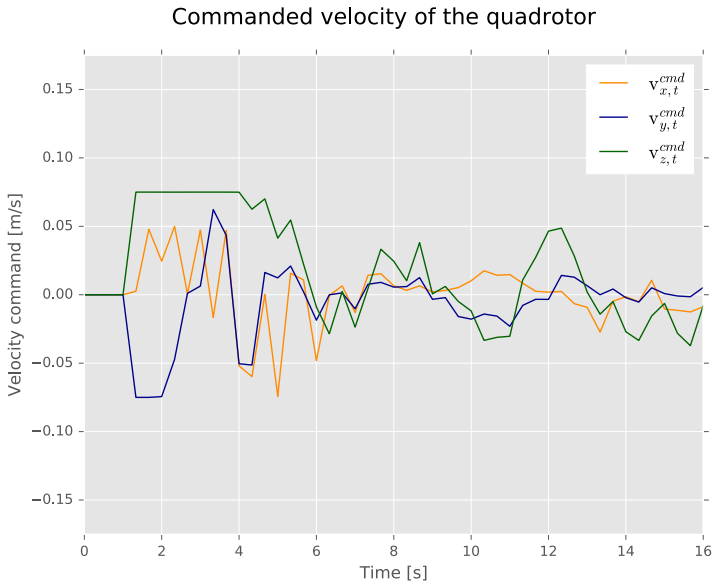


**Figure 5.11:** After training: The velocity references sent to the quadrotors velocity controller from the position controller during the test.

### 5.2.3 Potential sources of error

The agent never got to a point where it was as proficient at controlling the real-world quadrotor as it was at controlling the simulated quadrotor. While we do not know if the agent could have eventually converged to a better solution, even though it seemed as if the agent was making steady progress, there are some potential sources of error that are worth mentioning and discussing:

1. The first one is the delay associated with sending images from the quadrotor and sending back commands to the quadrotor over the WiFi connection. We do not know how much the delay varies, and the agent has no information about the delay which would otherwise allow it to adapt to it.

2. The height of the quadrotor drifted slightly, and the accuracy of the altitude measurement therefore degraded over time. To combat this training was paused periodically and the quadrotor was landed to allow for the state estimator to re-calibrate the height. Yet, it still introduces some

error because the motion of the object in the image depends directly on the height.

3. Aerodynamic forces resides in the real world. While the quadrotor operated in a 2m altitude the gusts created by the rotors will still affect the quadrotor, and the agent only has limited information through the previous actions included in the state representation and how it affected the quadrotor via the observed velocity of the object in the image to reason about the forces.

Chapter 6

# Future work

A problem that was encountered in Chapter 4 was that the learned control policy would often diverge over time, and represents a quite severe problem with DRL in general concerning stability and robustness as it would be difficult to justify the using such an approach in an online real-world setting. Divergence in DDPG is a known problem which is caused by an overestimation of Q-values, which in returns leads to the policy deteriorating because the actor networks are trained using the error in the Q-function. A more recent algorithm, known as the Twin-Delayed Deep Deterministic policy gradient [57] (TD3) which builds on the DDPG algorithm addresses this problem by analyzing the bias and variance of the Q-function. The resulting algorithm was shown to substantially improve performance over the baseline DDPG that was used in this thesis.

This is just one example of new additions to the rapidly expanding collections of DRL methods and shows that both the performance and stability of the controllers presented here can be improved by considering other DRL methods and the ones to come. Because of safety concerns we therefore expect that research on stability and explainable AI to be key in the following years before we see DRL applied to real-world control problems in robotics by the industry.

With respect to the performance of the proposed controller presented in this thesis, and the design of the controller itself, there is a lot of room for

improvement and interesting paths to explore. The internal controller of the particular quadrotor studied in this project represents a common design for velocity control of quadrotors as they assume that the velocity in x–, y– and z-direction can be controlled independently. It would be interesting to extrapolate this assumption to position control, for example by using three instances – three agents – of the DDPG algorithm (or other DRL method, like TD3) to control each of the three axes independently. This would facilitate and allow each agent to stop training when they have reached a satisfactory control performance in their respective axis. This could be a solution to the issue that was discussed in 4.4 regarding simultaneous convergence of the three objectives, and represents an interesting area of research regarding DRL optimization in multi-agent environments. Another interesting way of tackling this problem is to look at way the replay buffer in the DDPG algorithm is sampled by selectively picking transitions with high TD-error, for example by using a Prioritized Replay Buffer [14].

The learning process itself can also be improved upon. It would interesting to investigate if guided policy search, where another controller decides and help to explore in the initial training steps, would make the controller based on the image position to be able to learn even at higher controller rates. The controller based on the true horizontal position, or even the same, image position-based controller trained with a lower control rate could be used as a supervisor.

While we showed how the performance of the the controllers' differed by using both evidence of the true horizontal position and the true horizontal position itself, no comparison was done to a more traditional control theory approach. In the future it is therefore interesting to explore if the controller showcased here can challenge previously established methods in terms of accuracy, for example a position controller based on a well-tuned PID controller or a more complex approach based on system modeling like model predictive control (MPC).

The fashion in which DRL is coupled with a classical control theory approach in this project is just one way of doing it, and other methods to fuse DRL with control theory is interesting to explore. For example, a position

controller based on a PID controller where the gains of the controller are decided by a DRL approach could prove to be a viable approach.

The controller proposed in this thesis showed to be capable of handling external disturbances which is a sought after characteristic for control of quadrotors because aerodynamics are notoriously hard to model. It would therefore be interesting to explore if there are better ways of providing a DRL agent with information about the disturbance rather than the approach of including past state variables as done in this thesis.

Lastly, the range of the controller proposed in in this thesis was severely crippled by the short field of view of the camera that was used. To increase the range it could therefore be worth looking into using a camera with higher FOV, for example one that uses a fisheye lens.

Chapter 7

# Conclusion

In this thesis, a method for solving the quadrotor hovering problem in indoor environments was presented that was based on the deep reinforcement learning framework and monocular images. In contrast to other common control methods, our approach embraces a path which seeks a more sophisticated control method as opposed to a more complex state estimation technique. In addition, instead of attempting to model complex aerodynamics the control method is a model-free approach for optimal control.

The approach was realized by computing the 2-D position of a recognizable object visible in camera images, and using this directly to represent the horizontal position of the quadrotor as a pseudo-objective for controlling the true position. The proposed controller was tested in a simulation based on a commercially available quadrotor, the Parrot AR.Drone 2, and in addition to being able to solve the problem, the controller was also found capable of neutralizing disturbances present in the simulation. Furthermore, the proposed controller was compared against itself when the position extracted from the camera images was replaced with the true horizontal position of the quadrotor in the world. The results shown indicates that the controller, when using the image position, is nearly as proficient at controlling the quadrotor as when using the true horizontal position.

Simultaneously, training the proposed controller from scratch to a satisfactorily level in simulation was found to be rapid enough to be able to do even

with the most commonly available hardware within minutes. This allowed for quick prototyping and enabled us to test the proposed controller on the physical quadrotor in the real world as well. It was found that the controller, which was only trained prior by simulation, could generalize well enough with respect to the real world that training could continue practically autonomously. While no definitive result with respect to how good the controller trained in the real world turned out to be, or could be given enough time to learn, the results shown indicates that proposed method could be a viable approach. This is based on the level of accuracy it displayed in simulation, and that the time required to learn the task was quite insignificant especially compared to similar applications of DRL.

However, there some drawbacks to the approach presented here which is directly related to DRL itself. A known shortcoming with DRL is stability under training which was encountered during this project. There are no guarantee with respect to convergence, and training may even diverge completely in the midst of training. There is also no available proofs with respect to stability for the resulting controller, which is of concern to safety for robots that interacts with humans. It therefore seems evident that research on AI safety and stability will be important in the years come, which will hopefully make the powerful tool that is DRL more robust and a viable solution for a wider range of problems.

# Bibliography

[1] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. Von Stryk, "Comprehensive simulation of quadrotor uavs using ros and gazebo," in *International conference on simulation, modeling, and programming for autonomous robots*, pp. 400–411, Springer, 2012.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[5] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[6] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, *et al.*, "Learning to

navigate in complex environments," *arXiv preprint arXiv:1611.03673*, 2016.

[7] G. Lample and D. S. Chaplot, "Playing FPS Games with Deep Reinforcement Learning.," in *AAAI*, pp. 2140–2146, 2017.

[8] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[9] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[10] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning.," in *AAAI*, vol. 2, p. 5, Phoenix, AZ, 2016.

[11] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, *et al.*, "Starcraft II: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[12] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," *arXiv preprint arXiv:1606.06565*, 2016.

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[15] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, pp. 5048–5058, 2017.

[16] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *CoRR*, vol. abs/1611.05763, 2016.

[17] N. Mishra, M. Rohaninejad, X. Chen, and P. Abbeel, "A Simple Neural Attentive Meta-Learner," in *International Conference on Learning Representations*, 2018.

[18] S. Lange, M. Riedmiller, and A. Voigtländer, "Autonomous reinforcement learning on raw visual input data in a real world application," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2012.

[19] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Deep spatial autoencoders for visuomotor learning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 512–519, IEEE, 2016.

[20] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor," *IEEE robotics & automation magazine*, vol. 19, no. 3, pp. 20–32, 2012.

[21] B. Erginer and E. Altug, "Modeling and PD control of a quadrotor VTOL vehicle," in *2007 IEEE Intelligent Vehicles Symposium*, pp. 894–899, IEEE, 2007.

[22] L. V. Santana, A. S. Brandao, M. Sarcinelli-Filho, and R. Carelli, "A trajectory tracking and 3d positioning controller for the AR.drone quadrotor," in *2014 international conference on unmanned aircraft systems (ICUAS)*, pp. 756–767, IEEE, 2014.

[23] J. Kim, M.-S. Kang, and S. Park, "Accurate modeling and robust hovering control for a quad-rotor VTOL aircraft," in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pp. 9–26, Springer, 2009.

[24] Y. M. Mustafah, A. W. Azman, and F. Akbar, "Indoor UAV positioning using stereo vision sensor," *Procedia Engineering*, vol. 41, pp. 575–579, 2012.

[25] S. Grzonka, G. Grisetti, and W. Burgard, "A fully autonomous indoor quadrotor," *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 90–100, 2011.

[26] M. Achtelik, A. Bachrach, R. He, S. Prentice, and N. Roy, "Autonomous navigation and exploration of a quadrotor helicopter in GPS-denied indoor environments," in *First Symposium on Indoor Flight*, no. 2009, Citeseer, 2009.

[27] S. Weiss, D. Scaramuzza, and R. Siegwart, "Monocular-SLAM–based navigation for autonomous micro helicopters in GPS-denied environments," *Journal of Field Robotics*, vol. 28, no. 6, pp. 854–874, 2011.

[28] N. O. Lambert, D. S. Drew, J. Yaconelli, R. Calandra, S. Levine, and K. S. J. Pister, "Low Level Control of a Quadrotor with Deep Model-Based Reinforcement learning," *CoRR*, vol. abs/1901.03737, 2019.

[29] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a Quadrotor With Reinforcement Learning," *IEEE Robotics and Automation Letters*, vol. 2, pp. 2096–2103, Oct 2017.

[30] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning deep control policies for autonomous aerial vehicles with MPC-guided policy search," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 528–535, May 2016.

[31] R. Polvara, M. Patacchiola, S. Sharma, J. Wan, A. Manning, R. Sutton, and A. Cangelosi, "Toward End-to-End Control for UAV Autonomous Landing via Deep Reinforcement Learning," in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 115–123, IEEE, 2018.

[32] A. Rodriguez-Ramos, C. Sampedro, H. Bavle, P. De La Puente, and P. Campoy, "A deep reinforcement learning strategy for UAV autonomous landing on a moving platform," *Journal of Intelligent & Robotic Systems*, vol. 93, no. 1-2, pp. 351–366, 2019.

[33] Y. Wang, J. Sun, H. He, and C. Sun, "Deterministic Policy Gradient With Integral Compensator for Robust Quadrotor Control," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.

[34] H. X. Pham, H. M. La, D. Feil-Seifer, and L. V. Nguyen, "Autonomous uav navigation using reinforcement learning," *arXiv preprint arXiv:1801.05086*, 2018.

[35] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, USA: The MIT Press, 2nd ed., 2018.

[36] R. Bellman, "A Markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[37] D. P. Bertsekas, *Dynamic programming and optimal control*, vol. 1. Athena scientific Belmont, MA, 1995.

[38] H. van Hasselt and M. A. Wiering, "Convergence of model-based temporal difference learning for control," in *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pp. 60–67, IEEE, 2007.

[39] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[40] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[41] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski,

*et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[42] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[43] Y. Bengio *et al.*, "Learning deep architectures for AI," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[44] A. B. Martinsen and A. M. Lekkas, "Straight-path following for underactuated marine vessels using deep reinforcement learning," *IFAC-PapersOnLine*, vol. 51, no. 29, pp. 329–334, 2018.

[45] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[46] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[47] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[48] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[49] R. Szeliski, *Computer Vision: Algorithms and Applications*. Berlin, Heidelberg: Springer-Verlag, 1st ed., 2010.

[50] "Real Time pose estimation of a textured object." "https://docs.opencv.org/master/dc/d2c/tutorial_real_time_pose.html".

[51] N. P. Koenig and A. Howard, “Design and use paradigms for Gazebo, an open-source multi-robot simulator.,” Citeseer. http://gazebosim.org/.

[52] R. Smith *et al.*, “Open dynamics engine,” 2005.

[53] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[54] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000.

[55] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” 2015. Software available from tensorflow.org.

[56] S. Piskorski, N. Brulez, P. Eline, and F. D’Haeyer, “AR.Drone developer guide,” *Parrot, SDK*, vol. 1, 2012.

[57] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” *arXiv preprint arXiv:1802.09477*, 2018.

Lasse Hansen Henriksen

# Hovering Control of a Quadrotor Using Monocular Images and Deep Reinforcement Learning

NTNU
Norwegian University of
Science and Technology