NTNU – Trondheim
Norwegian University of
Science and Technology

TTK4550 - ENGINEERING CYBERNETICS,
SPECIALIZATION PROJECT

A reinforcement learning study for quadrotor altitude control

Project thesis

Written by Lasse Henriksen
Supervised by Anastasios
Lekkas

June 24, 2019

Department of Engineering Cybernetics
Norwegian University of Science and Technology

Abstract

The fields of artificial intelligence and thereby reinforcement learning has attracted considerable attention the last few years as several breakthroughs in research on both the particular field and the availability of computational power has surged. Closely related to reinforcement learning is the domain of control theory and the question has been raised if whether this new-found technology can compete, or even outperform, traditional control theory approaches for robotic control applications. The theory of reinforcement is therefore in this project studied thoroughly to first learn about the different algorithms and approaches and the application areas they are best suited to solve. Then, some of the algorithms are implemented on specific problems related to robotics control to gain practical experience with both the algorithms themselves, but also with artificial neural networks which is a technology important for state-of-the-art applications. In the end one of the most prominent reinforcement learning algorithm at the time is applied to devise a controller to solve a real robotic control problem: quadrotor altitude control, by using a simplified simulated version of a quadrotor as a starting point. The resulting controller is then compared to a traditional control method and the results are discussed.

Table of Contents

List of Tables

List of Figures

Abbreviations

MDP	=	Markov decision process
POMDP	=	Partially observable markov decision process
MC	=	Monte-carlo
TD	=	Temporal difference
RL	=	Reinforcement learning
DRL	=	Deep reinforcement learning
UAV	=	Unmanned aerial vehicle
DDPG	=	Deep deterministic policy gradient
DQN	=	Deep Q-Learning
MPC	=	Model predictive control
ANN	=	Artificial neural network
ODE	=	Open Dynamics Engine
URDF	=	Universal Robot Description Format
ROS	=	Robot Operating System
PID	=	Proportional-integral-derivative
LQR	=	Linear Quadratic Regulator
GA	=	Genetic algorithm
MOGA	=	Multi-objective genetic algorithm
PPO	=	Proximal policy gradient

Chapter 1

Introduction

1.1 Background and motivation

Control of rotor-based aerial vehicles have been studied intensively by the scientific control theory community all the way back from the 20st century and so the theory is well established at this point. However, despite all the research that has gone into this subject the problem still remains a hard one. This is because almost all classes of rotor aircrafts are underactuated systems which exert highly nonlinear dynamics because of the environment in which they operate. Even though fluid mechanics is also a well established field of study and good models for the forces and torques that affect the aircrafts exists, these systems are also time-variant because their dynamics depend highly on the mode of operation.

For example, a quad-rotor will experience very different environmental forces when it's hovering close to the ground versus high speed maneuvering while following a trajectory. In addition the aerodynamic forces is often dominant in small-scale rotor crafts. In this regard, the control of small-scale quad-rotors is a particularly challenging and interesting problem.

Because of the complexity of modelling aircrafts some trade-offs are often necessary between the computational time required calculate actuator inputs and the precision. Current approaches for control theory based controllers for quad-rotors control utilize different methods like linear- or nonlinear model predictive control [3], feedback linearization [4], backstepping control [5], and LQR- and PID control [6]. They have one thing

in common, namely that they are based on a known model of the system dynamics, which require modelling of the aforementioned complex dynamics, which can be tedious and error-prone. This is problematic because the performance of the controllers are highly susceptible to modelling errors which often occur in complex systems.

Another interesting approach for controlling quad-rotors, or robotic systems in general, is machine learning. And more specifically, deep reinforcement learning. This is an attractive alternative because many methods from reinforcement learning are able to learn model-free, and can thus therefore avoid some of the problems that traditional approaches have revolving system modelling.

It is therefore interesting to investigate how current methods based on artificial intelligence fares against control theory approaches in terms of performance measures like robustness and precision and for the sake of redundancy. Maybe even can the two methods be combined to form solutions that benefits from the best of two worlds. Comparing the two different methods is also of interest in a more philosophical nature with respect to ”AI vs human” intelligence. If we think about the traditional control approaches as human intelligence and DRL as the artificial intelligence (even though, of course, it is human technology after all) then all we need is a concrete task to solve.

In this project I aim to put this question to test, with quad-rotor altitude control as the measuring task between artificial- and human intelligence. Has the age of AI intellectual dominance finally dawned upon us? If that is the case, then I would like to state for the record that I for one welcome our new robot overlords [7].

1.2 Related work

Deep reinforcement learning (DRL) has shown particularly promising results where traditional control approaches are hard to apply. These are typically abstract tasks that requires an ad-hoc or a specifically tailored solutions because they’re hard to express using traditional modeling methods. Some examples of successful applications includes dexterous control of a robotic arm to do Lego stacking [8], advanced autonomous aerobatic flight of RC helicopter based on system dynamics found by model identification of the system [9][10], learning hand-eye coordination for robotic grasping

from monocular images [11] and path following for marine vessels [12] that was shown to outperform the traditional control method.

Another interesting use of DRL in robotics is to use traditional approaches to guide the agent in learning. In direct relevance to quad-rotors, [13] show that an AI controller can learn much faster by using a MPC controller to guide the search and the final controller computes motor inputs at a pace that is two orders of magnitude faster than the classic control approach. Results from [14] also confirms that the computational time is reduced by two orders of magnitude while performing tasks like in-air recovery and trajectory following.

There also exists examples of doing the opposite, improving traditional approaches by using DRL. There exists numerous examples of DRL agents that learns to tune PID controllers [15][16][17], and an example of more relevant work [18] that tunes the weight matrices of a model-predictive trajectory planning algorithm for an unmanned aerial vehicle (UAV).

DRL has also been applied to a closely related field of robotics, virtual robotics, or more commonly known as video-games. Numerous results have shown that DRL agents can learn to play better than the best previous known algorithm, and some can even perform at a human, or even super-human level. Examples of this is agents that can play Doom [19] and various Atari games [20][21][22]. Board games such as Go [23] and Chess and Shogi [24] has also been mastered by DRL agents.

1.3 Project objective

This project is an assemble of work to prepare for future research to be conducted under my master thesis. The main objectives in this project is therefore the following:

1. Study and understand the theory of reinforcement learning including basic theory and principles and up to what is currently considered state of the art algorithms.

2. Gain practical experience by implementing reinforcement learning methods for tasks related to robotics and control.

3. Study the dynamics of quadrotors, implement a reinforcement learning method to solve the quadrotor altitude control task and compare

its performance and behavior to a control method based on traditional control theory.

1.4 Outline of report

Chapter 2 introduces the basic theory of reinforcement learning in general. Chapter 3 discusses different reinforcement learning algorithms and applies each one of the actual algorithms to a specific problem. Chapter 4 introduces the dynamics of quadrotors and presents a simulated quadrotor environment. Two controllers are then presented: one being based on reinforcement learning and the other on a traditional control approach. Finally the controllers and compared and the comparative performance of the two controllers are discussed. Chapter 5 attempts to draw a conclusion of the project based on the initial research questions posed in Section 1.3 and the results presented in Chapter 4.

Chapter 2

Theory

This chapter starts off by introducing the basic principles and theory that reinforcement learning is built on, and then exploring the necessary prerequisites to understand the more complex state-of-art reinforcement learning algorithms are explored. The content discussed in this chapter is heavily based on the book "Reinforcement Learning: An Introduction" by Sutton and Barto [1]. It is rendered here for the sake of completeness, but also interpreted from a control engineers viewpoint for the sake of comparison. Only the most relevant concepts related to the project are included.

2.1 The fundamental reinforcement learning agent

2.1.1 Agent and environment

In any reinforcement learning (RL) agent there are three fundamental signals at each timestep of the agents life: the observation O_t, the action A_t and reward R_t which is consequence of its current state and the actions performed up until then. These signals represents the interaction be-

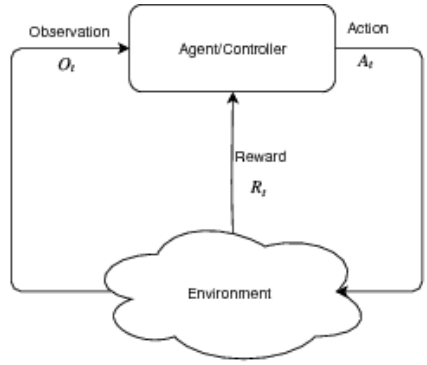


Figure 2.1: The three signals O_t, A_t and R_t summarizes the interaction between the agent and the environment.

tween the agent and the environment. Figure 2.1 encapsulates these relationships. The observation is used to construct the agent internal representation of its current state s_t in the environment.

The 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. This can be formally expressed as a Markov decision process (MDP) which is introduced in Section 2.1.3.

The distinction between the environment and the agent is not always ambiguous. Let's say the objective is to control a robotic arm and move the end-effector to a desired location, it is tempting to think of the robotic arm as the agent. It is however 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 Control theory analogy

The model in Figure 2.1 closely resembles what control engineers knows as closed-loop feedback control, in which case the action signal is known as the input signal u_t and the observation is known as y_t, but the notation can vary in different literature. The environment is instead denoted as the system, or the plant, and an additional signal is included known as the reference signal r_t (not to be confused with the reward in the agent model) and the objective is to steer the state of the system towards the r_t.

The reward signal on the other hand has no direct analogy in this context and at the core this is what separates the two methods from each other. The reward signal, as we shall see, is what enables the AI agent to learn and solve problems as opposed to traditional control methods that depend on mathematical modelling and domain knowledge.

Traditional control theory approaches also requires a high level planner in the overall system in many problem formulations. This can be illustrated

by looking back at the robotic arm example. Say the task is to grasp an object and put it somewhere, planning the successive control inputs to successfully grasping it, holding on to it, and moving it to the desired location must also be included in the solution. A RL agent on the other hand has the advantage that all of the aforementioned steps does not have to explicitly expressed, but can instead be learned by interacting with the environment.

2.1.3 Markov Decision Processes

An important underlying assumption in reinforcement learning 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) [25].

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 are simply a function of 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 the property to hold, the states of the system is required to contain all information that influence the future states of the system. This property is important because making an optimal decision at any point in time is impossible if we're not aware of the full situation. 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. This is useful in many situations and can be used to act optimal based on the agents beliefs of the world.

2.1.4 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 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's just the sum of the rewards over an episode:

$$G_t \triangleq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T. \tag{2.1}$$

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

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{N-1} R_{t+N} = \sum_{n=0}^{N} \gamma^k R_{t+n+1}. \tag{2.2}$$

The discount factor γ determines the present value of future rewards such that the agent will tend to prefer immediate rewards over future rewards. When γ approaches 1 the agent becomes more farsighted, and when it approaches 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

$$\begin{aligned} G_t &\triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4}) + ... \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.3}$$

which also simplifies notation.

2.1.5 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 called a policy. The policy of an agent is a mapping from states to probabilities of selecting actions. $\pi(a|s)$ is the probability of selecting action $A_t = a$ and state $S_t = s$ at timestep t.

By definition the value function $v_\pi(s)$ expresses the value of a state s when acting under the policy π. This is called the state-value function for policy π. For MDPs this can be expressed formally as

$$v_\pi(s) \triangleq \mathbb{E}\left[G_t|S_t = s\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+n+1}|S_t = s\right]. \tag{2.4}$$

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

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

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 (2.3). 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

$$
\begin{aligned}
v_\pi(s) &\triangleq \mathop{\mathbb{E}}_{\pi}\left[G_t | S_t = s\right] \\
&= \mathop{\mathbb{E}}_{\pi}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathop{\mathbb{E}}_{\pi}\left[G_{t+1} | S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}.
\end{aligned}
\tag{2.6}
$$

This is known as the Bellman equation for v_π. 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 estimate 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.

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 transitions between states that the agent experiences.

2.1.6 Bellman optimality equations

Broadly speaking, the reinforcement learning problems objective is to find a policy that generates the most reward over time. A policy π is considered better or equal to another policy π' if its expected return is greater or equal to that of π'. 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 is inequality have the same state-value function called the optimal state-value function v_*, and they are denoted as the optimal policy π_*. The optimal state-value function can be expressed as

$$
v_*(s) \triangleq \max_\pi v_\pi(s) \tag{2.7}
$$

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

$$q_*(s,a) \triangleq \max_\pi q_\pi(s,a) \tag{2.8}$$

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

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{1+t}) | S_t = s, A_t = a\right]. \tag{2.9}$$

These functions also satisfy the consistency condition given by the Bellman equation in (2.6). 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

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathop{\mathbb{E}}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\ &= \max_a \mathop{\mathbb{E}}_{\pi_*}\left[R_{t+1} + \gamma G_{t_+1} | S_t = s, A_t = a\right] \\ &= \max_a \mathop{\mathbb{E}}_{\pi_*}\left[R_{t+1} + \gamma v_*(S_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r | s, a)\left[r + \gamma v_*(s')\right]. \end{aligned} \tag{2.10}$$

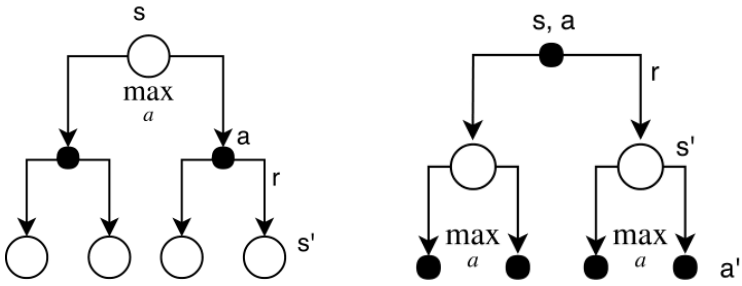
By using (2.3).

For q_* the Bellman optimality equation is given by

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

These equations can be illustrated graphically as backup diagrams, as shown in Figure 2.2. As opposed to equation (2.2), they show that instead

of taking the expected value given some policy, the action a that gives the maximum reward is always chosen.



(a) Value graph. **(b)** Action-value graph.

Figure 2.2: Backup diagrams for the Bellman optimality equations.

2.2 Learning approaches

Solving the Bellman optimality equations analytically gives a solution to the reinforcement learning problem. The class of methods that solves these equations are known as dynamic programming. Unfortunately, explicitly solving the set of Bellman equations is often impractical. There are three main reasons for this:

1. The dynamics of the environment is not accurately known.
2. The state space of the problem is too large, which can lead to the problem being computationally intractable to solve because the computational requirements grow exponentially with the number of state variables. Richard Bellman himself coined this phenomenon as the ”curse of dimensionality”.
3. The Markov property is not satisfied.

Usually we have to settle for approximate solutions because some or all of the above properties are compromised. Therefore many reinforcement learning methods are based on trying to find a way to approximate the Bellman optimality equation based on past experienced transitions. Most tasks considered to be interesting in robotics either have continuous, or very

large action- and state spaces. The main problem of this project is a good example of this. Therefore, the subsequent sections will focus on introducing several methods in reinforcement learning that aims to estimate and optimize v_π, q_π and π without analytically solving the Bellman equations.

2.2.1 Prediction- and control problem

Generally, the different learning approaches can be divided into two problems. The first one is called prediction which aims to estimate the value- of action-value functions. The second is called control which aims to optimize the policy based on the estimated value/action-value functions.

The control problem is solved by acting greedily with respect to the current estimation of the value/action-value function. If we have an estimate of the action-value function, call it $q(s, a)$, then the corresponding greedy policy is the one that deterministically chooses the action with maximal action-value, hence

$$\pi(s) \triangleq \text{argmax}_a q(s, a). \tag{2.12}$$

Solving the prediction problem is where the main differences between the methods lie, which is discussed briefly next. Readers may want to consult [1] for the specifics and different flavors of the methods discussed.

2.2.2 Monte-Carlo learning

Monte-Carlo learning is a way of learning in environments that are based on averaging samples of complete returns. Hence the environment must be separable into episodes or be episodic in nature. Similarly to other methods we discuss in this section the methods require only experience to learn. Formally we say experience is samples of states, actions and rewards gathered from real or simulated interaction between agent and environment.

Monte-Carlo methods can be separated into two categories: *first-visit MC* and every-visit MC. First-visit MC only estimates the value function as the average of return of first visits to a state s, while *every-visit MC* averages the return of all visits to s.

To illustrate the basic notion of the Monte-Carlo learning approach to estimate $\pi \approx \pi_*$, a version which uses first-visits and ES (exploring starts) is summarized in Algorithm 1, where ES is an exploration strategy that

always picks random states which assures that all states are continuously visited. Note that even though the action-value function is used in this example, using the value function instead is straight forward, and only requires an extra one-step look-ahead to find the action a to calculate $\pi(S_t)$.

Algorithm 1 Monte-Carlo with ES

Initialize storage for action-values $Q(s, a)$ and set initial values for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
Initialize $\pi(s) \in \mathcal{A}$ (arbitrarily) for all $s \in \mathcal{S}$
Initialize Returns(s,a) $\leftarrow$ empty container for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
for episode in $1 : M$ **do**
  Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs of actions and states as probability > 0 of being chosen
  Generate an episode from S_0, A_0 following π:
  $[(S_0, A_0, R_1), (S_1, A_1, R_2), ...(S_{n-1}, A_{n-1}, R_n)]$
  G $\leftarrow 0$
  for Each timestep in episode: $1 : n$ **do**
    G $\leftarrow \gamma G + R_{t+1}$
    if S_t, A_t has not already appeared in the episode during time $0 : n-1$ **then**
      Add G to Returns(S_t, A_t)
      $Q(S_t, A_t) \leftarrow$ average(Returns(S_t, A_t))
      $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$
    end if
  end for
end for

Usually ES is an unlikely assumption in real applications, and other approaches are used to ensure that the agent continuously explores all states. These can be divided into on-policy, and off-policy methods. On-policy methods evaluates or improves the policy that is used to make decisions while off-policy methods evaluate or improve a policy that is different to the policy taking the actions. Algorithm 1 is an on-policy method.

Off- and on-policy methods are not directly linked to MC learning, but to reinforcement learning approaches in general. On-policy methods are usually simpler, have lower variance and converge faster, while off-policy methods have higher variance, slower convergence, but are more power

and have more applicable potential in general. Off-policy methods are for example able to learn from other controllers, like a controller based on control theory for robotics control. The greater variance related to off-policy methods comes from the fact that it's hard to guarantee that the data received is related to the policy you're trying to learn about, and hence the estimates are less likely to be accurate.

2.2.3 Temporal-difference learning

Another important idea of reinforcement learning is the notion of temporal-difference learning. While Monte-Carlo learning estimates the value function based on averaging returns, TD learning only waits until the next timestep before it can make an update to the current value function estimate. Similarly to DP, TD also bootstraps, meaning it updates the existing value $V(s)$ of some state s based on another value $V(s^{'})$ for some other state $s^{'}$ related to s by taking an action a in s. TD can thus be viewed as taking the idea of sampling from MC and the idea of bootstrapping from DP.

To illustrate the TD learning principle is the simple TD update

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

which can be used to predict v_π under the policy π, where α is the learning rate. The full algorithm is shown in Algorithm 2. This method is called *TD(0)*, or one-step TD. TD(0) is a special case of TD(λ) which is a n-step TD method which provides a seamless relation between TD- and MC learning methods. However, TD(λ) is not discussed here as it is not used by the algorithms considering in this project.

Algorithm 2 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 general $TD(0)$ algorithm has been shown to converge to an optimal solution V_* under th following conditions [26]:

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) $< \infty$.

Both TD- and MC learning uses value- and action-value functions to form a policy. This 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 optimal control for a quadrotor - this would inevitably lead to a sub-optimal solution because information would be lost. It is also worth noting that even 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.

2.2.4 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 $\pi(a|s, \theta)$ where θ is the policy's parameters. The policy outputs a probability for choosing an action a_t given a state s_t and parameters θ_t at a time t: $\pi(a|s, \theta) = P(A_t = a|S_t = s, \theta_t = \theta)$.

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

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.14}$$

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

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t\right]. \tag{2.15}$$

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

$$\nabla J(\theta_t) = \nabla v_{\pi_\theta}(s_0). \tag{2.16}$$

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 [27] comes in, which for the episodic case can be stated as

$$\begin{aligned}\nabla J(\theta_t) &= \nabla v_{\pi_\theta}(s_0) \\ &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)\end{aligned} \tag{2.17}$$

where $\mu(s)$ is the weighted state distribution probabilities under the policy. Note that the right side of (2.17) can be calculated if the parameterized policy π_θ is differentiable with respect to it's parameters.

Parameterization can be realized by devising a parameterized numerical preference function $h(s,a,\theta)$ 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

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta))}}{\sum_b e^{(h(s,b,\theta))}} \tag{2.18}$$

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 ϵ-greedy strategy often used for exploration in tabular methods, as in Section 3.1. While this is one example of a parameterization function the choice is arbitrarily. Other examples include using a linear combination of features (states) and weights θ as $h(s,a,\theta) = \theta^T \mathrm{x}(s,a)$ or a deep artificial neural network (Section 2.3).

Section 3.2 discusses a policy gradient method called REINFORCE [28] which combines Monte-carlo learning with policy gradients. Readers may want to read Section 3.2 before continuing to the next section to see how the policy gradient theorem (2.17) is used to form a learning formula for updating the policy parameter vector θ.

2.2.5 Actor-critic methods

The policy gradient method described in the previous section relies on the episodic returns (MC approach). Monte-Carlo learning has higher variance than temporal difference learning because TD 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, MC is more prone to variance because at

each timestep there is potential variance that is injected into the return for that episode. In addition, MC 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 $\mathbf{w}$ is the parameter vector.

A popular approach to actor-critic methods is therefore to use a one-step return like $TD(0)$, though any policy evaluation technique may be used in practice. The parameter update rule for the MC case can be written in terms of the $TD(0)$ update to form an online algorithm:

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha(G_{t:t+1} - \hat{v}(S_t, \mathbf{w}))\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\
&= \theta_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\
&= \theta_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\
&= \theta_t + \alpha\delta_t \ln\nabla\pi(A_t|S_t, \theta_t).
\end{aligned} \tag{2.19}$$

For the critic part, 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

$$J(\mathbf{w}) = \frac{1}{2}\delta^2 \tag{2.20}$$

where $\delta = (R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}))$. 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

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\nabla v(S_t, \mathbf{w}_t). \tag{2.21}$$

2.3 Neural networks

For solving our quad-rotor problem it is now clear that we need to use an approach to learning that can handle continuous state- and action spaces, such as an actor-critic method. As we have already established in chapter 1 this is a highly nonlinear task, and therefore we need parameterized functions that is capable of approximating the dynamics of the system. A natural choice for a function approximation is therefore to use artificial neural networks (ANNs), which has been applied with success to similar studied problems [12][13][14] and others. When ANNs are used in combination with reinforcement learning it is often called deep reinforcement learning.

Figure 2.3 shows a generic ANN structure which can be characterized as a directed graph where each circle represents a weight and activation function, and the arrows represents the incoming signal from the feed forward operation. This operation takes an input signal, for example states variables, and does matrix multiplication with the first layer (left most row of circles), and sends the output as input to the next layer and continues to perform matrix multiplications until we get a final output from the network. The activation functions that are parameterized by the weights of the network plays an important role of introducing nonlinearity to the output of the network. This enables the network to represent any continuous function given by the sampled inputs [29]. In fact, this is true even for a shallow network structure with only one hidden layer, but experience [30] show that deeper 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.

The second fundamental operation in neural networks in addition to the feed forward operation is backpropagation. By using the chain rule known from basic calculus and a loss function that specifies the loss function parameterized by the networks weight the gradient of each weight can be found and adjusted in the direction that minimizes the loss. This is done by an update law known as the optimizer, which is exactly like gradient ascent (2.14) we saw in Section 2.2.4 and 2.2.5.

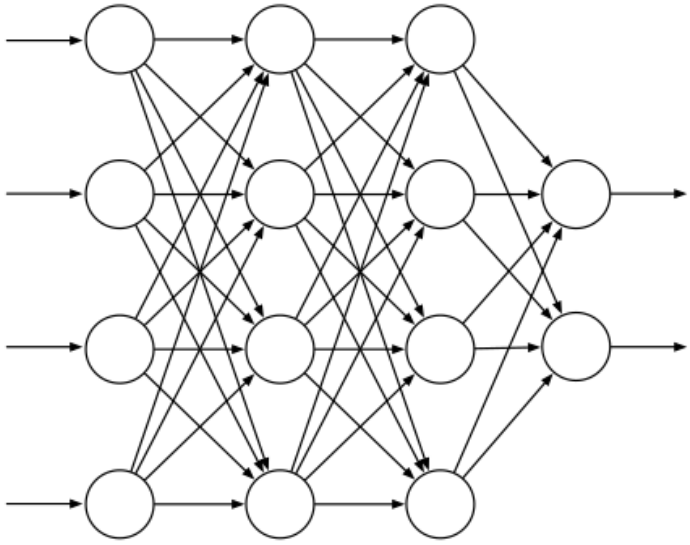


Figure 2.3: A generic feed-forward ANN with four input units, two output units, and two hidden layers [1].

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. Examples of this are batch normalization [31] and the Adam optimizer [32] which are used in state of the art applications. 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, like Tensorflow [33].

Chapter 3

Reinforcement Learning Methods and Implementations

To prepare for the quad-rotor problem three reinforcement learning methods were implemented to learn and gain practical experience: Q-learning, REINFORCE and Deep deterministic policy gradient (DDPG), which corresponds to one or more of the learning approaches discussed in the previous chapter. These three methods were chosen to represent all of the fundamental learning approaches in reinforcement learning, and to represent methods that together are able to solve a broad specter of different environments in terms of being discrete or continuous, and sequential or episodic. It is also beneficial to learn about older methods as a gateway to understanding the more complex state-of-the-art algorithms, which is why Q-learning and REINFORCE was experimented with before moving on to DDPG which is considered as one of the current state-of-the-art algorithms.

The environments that the algorithms were implemented for all are all from OpenAI Gyms [34] repository of environments and represent a variety of problems from discrete to continuous in both action- and state space.

3.1 Q-learning

3.1.1 Algorithm outline

This an online and off-policy algorithm based on temporal difference learning (Section 2.2.3). It learns on the go and updates its action-value estimate

at each time step t. In its basic form its applicable to problems which are discrete in nature. It stores a action-value $Q(s, a)$ for all possible values of states s and actions a. The pseudocode for the learning algorithm is shown in Algorithm 3, where (3.1) is the learning formula.

For each time step during an episode it performs a one-step look-ahead backup for the current state based on taking the max action-value of the current state to transition to the next state. The algorithm will therefore never explore unless it is told to do so. To incorporate exploration into the process the ϵ-greedy exploration strategy was used, which means that the agent at any time t either performs an action from the max of the action-values, or does a random one with probability ϵ. This is the off-policy part of the algorithm.

Since there are 500 different states and 6 different actions to perform in each of them, we need to store $500 \times 6 = 3000$ different action-values $Q(s, a)$.

Algorithm 3 Q-learning

Set hyperparameters like total amount of episodes to train M step size α, discount factor γ, 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 ϵ-greedy)

    Take action A, observe reward R and next state S'
    Do:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right] \quad (3.1)$$

    $S \leftarrow S'$
  end while
end for

3.1.2 Problem description

The environment that was chosen to apply Q-learning to is known as Taxi-v2. The environment is illustrated in Figure 3.1. Our goal is for the agent to perform as good as possible after a finite number of episodes.

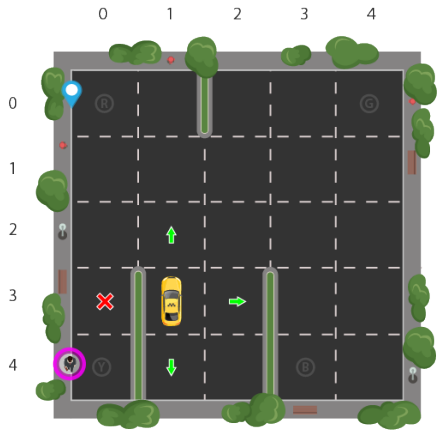


Figure 3.1: An illustration of the Taxi-v2 environment found in OpenAI Gym.

State-space

The environment is a discrete, square grid with size 5×5 and the state-space vector contains the x-y position of the taxi, the index of the location of the passenger and the index of the drop-off location. There is four possible drop-off locations (marked R, G, Y, B in Figure 3.1) for which the passenger can be at the start of any episode. The passenger can also be inside the taxi, which is the fifth possible location of the passenger. The destination location and the initial passenger location are never the same because this would be a terminal state. This means that the environment has $5 \times 5 \times 5 \times 4 = 500$ possible discrete states, for which 4 of them are terminal states.

Action-space

The agent (taxi) has six distinct actions which it can perform at any time step during an episode. These are 1) move south, 2) move north, 3) move east, 4) move west, 5) pick-up passenger and 6) drop-off passenger. The environment is completely deterministic such that the effect of any actions

always has a probability of 1 of happening when the action is chosen. Moving towards a wall inside the grid will have no effect and will cause the agent to stay put till the next time step. Note that there are walls not only on the boundaries of the world, but also inside it.

Rewards

At each time step the agent receives a reward of -1 by default. If the agent performs a drop-off or pick-up action that has no effect, i.e the passenger is not at the tile or the tile is not the destination for the passenger, the agent receives a -10 reward. A $+20$ reward is received by dropping the passenger off at the correct destination. Because of these rewards the the return of any episode is always less than 16 and depends on the initial state of the environment. This is because there is a minimum of four tiles between any two locations in the world.

3.1.3 Implementation and results

The agent was simulated inside the environment for $M = 10000$ episodes with $\alpha = 0.1$, $\gamma = 0.6$, and with three different choices for ϵ. Figure 3.2 shows the episodic returns for each different ϵ, while Figure 3.3 shows a snapshot of the agents policy after 2000 and 10000 episodes for $\epsilon = 0.1$.

By looking at Figure 3.3 we can see that the policy after 2000 episodes is far from perfect, and without the extra exploration it's easy to see that it could potentially get stuck from multiple initial states. Even after 10000 episodes we can see that there is one state in particular (bottom right) for which the agent has not yet found the optimal policy. Eventually the agent will find the optimal policy for all possible states, albeit slow, under the same conditions as for the general TD-learning algorithm discussed in Section 2.2.3. However, checking for optimality is not practical or even possible.

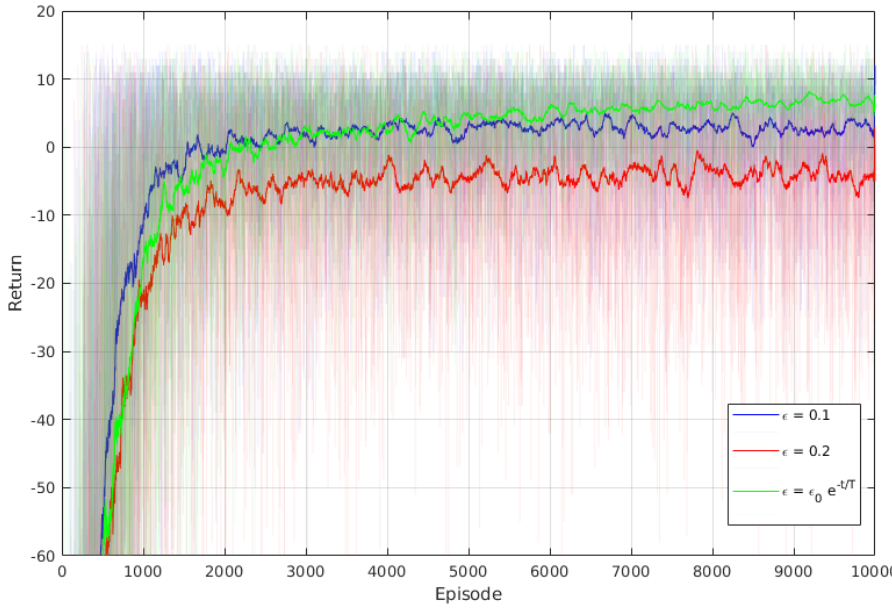
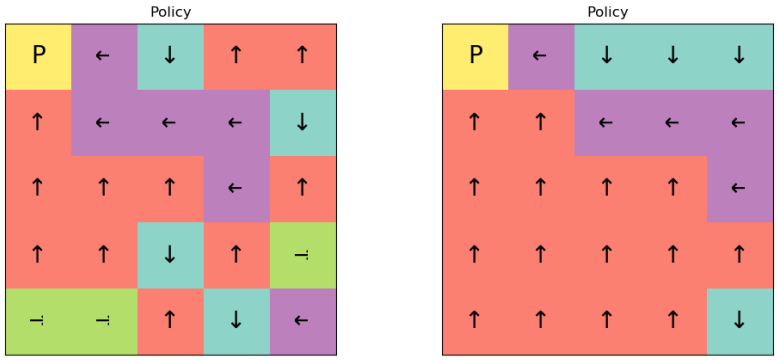


Figure 3.2: Returns for 10000 episodes of training for $\epsilon = 0.1$, (blue), $\epsilon = 0.2$ (red), and $\epsilon = \epsilon_0 e^{t/T}$ (green) with initial epsilon $\epsilon_0 = 0.2$ and time constant $T = 5000$. Opaque lines are the raw return data while solid lines are smoothed versions of the raw returns.

The consequence of this is that we can never turn off exploration completely (i.e set $\epsilon = 0$) because the agent is at risk of getting stuck in a subset of $\mathcal{S}$ forever because the policy would turn greedy and would never explore non-greedy actions. This also shows that exploration may contribute positively to the agents performance, but also inadvertently so, especially at the later stages when it has found the optimal policy for the majority of the possible configurations in the environment.

As we can see from Figure 3.2 an alternative to having constant exploration is to adjust ϵ adaptively, in this case by a exponential decay function as $\epsilon(t) = \epsilon_0 e^{-t/T}$ where $\epsilon_0 = \epsilon(0)$ and T is a time constant. This causes the agent to explore more in the initial episodes and less in the latter. The agent therefore learn faster at a cost of less rewards early on, but higher rewards later. This strategy does however increase the risk of getting semi-stuck in later episodes, like in certain states of Figure 3.3a,b as discussed above.



(a) Policy after 2000 episodes. **(b)** Policy after 10000 episodes.

Figure 3.3: Visualization of the agents policy after a) 2000 episodes and b) 10000 episodes in an initial state where the passenger is in the top-left destination location.

3.2 REINFORCE

3.2.1 Algorithm outline

Recalling the the contents of Section 2.2.4 we know that the gradient of the the performance measure $J(\theta)$ in (2.15) is

$$\nabla J(\theta_t) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{3.2}$$

Where the first sum on the right hand side weights the states based on how often they're visited. However, when the policy π is followed the expression becomes an expectation:

$$\nabla J(\theta_t) = \mathbb{E}\left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta)\right] \tag{3.3}$$

REINFORCE [28] uses this idea together with Monte-Carlo learning to form an update rule for the policy parameter vector that is based on returns. The expression in 3.3 can be simplified further, and becomes

$$\nabla J(\theta_t) = \mathbb{E}\left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right] \tag{3.4}$$

or equivalently,

$$\nabla J(\theta_t) = \mathbb{E}\left[G_t \nabla \ln \pi(A_t|S_t,\theta)\right]. \tag{3.5}$$

With this update rule we are able to handle continuous state spaces by using a parameterized policy function. The update rule for adjusting the parameters of the policy function is therefore:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G_t \nabla \ln \pi(A_t|S_t,\theta_t) \tag{3.6}$$

where α is the learning rate and γ is the usual discount factor. The pseudocode for the algorithm is shown in Algorithm 4.

Algorithm 4 REINFORCE

Initialize the differentiable policy parameterization $\pi(a|s,\theta)$, the neural network, and weights
Initialize hyper parameters: learning rate α, discount rate γ
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

3.2.2 Problem description

REINFORCE was used to solve the environment called CartPole-v0. In this environment the agent is tasked with balancing a pole on top of a cart which can move along a single frictionless axis. One end of the pole is rigidly attached by an un-actuacted joint to a fixed point on top the cart and can freely rotate around this point. A screenshot of the environment is shown in Figure 3.4.

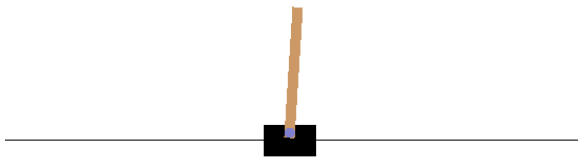


Figure 3.4: Screenshot of a single visualized frame from the Cartpole environment.

The environment is initialized with the the pole in an upright position and the cart close to the center of the axis. At each step of the environment the agent observes the linear- position and velocity of the cart, and the angular- position and velocity of the end of the pole not attached to the cart. The agent has two different actions, which is to either push the cart to the left or to the right. For every time step that the agent manages to keep the pole from falling more than 12 degrees from the vertical while keeping the cart within $\pm 2.4m$ of the starting position it receives $+1$ reward. The environment is summarized in table 3.1

Table 3.1: Summation of the Cartpole-v0 environment.

Environment	CartPole-v0	
Initial states	$x_c, \dot{x_c}, \theta_p, \dot{\theta_p} \in [-0.05, 0.05]$	uniformly distributed random values
Observations	$x_c, \dot{x_c}, \theta_p, \dot{\theta_p}$	Continuous variables
Actions	$F \in \{-1, +1\}$	Discrete variables
Rewards	$R_t = +1, \forall t$	
Termination condition	$\lvert x_c \rvert > 2.4$ or $\lvert \theta_p \rvert > 12°$ or $t = 200$	

3.2.3 Implementation and results

An ANN was used as a function approximator with four nodes as the input representing the four states, two hidden layers with size 10 and 2, and two

output nodes representing the probability of choosing either one of the two actions. For the hidden layers ReLU was used as activation functions while softmax with cross entropy was used for the output layer. As discussed in Section 2.2.4 the softmax function produces probabilities which is suitable for assigning probabilities in this case because we have a finite number of actions while it also assures continuous exploration of the action- and state space.

The agents was trained for 2000 episodes with and without the normalization of the returns and the results are shown in Figure 3.5. The same hyperparameters was used for both of the agents, with learning rate $\alpha = 0.01$ and discount factor $\gamma = 0.95$.

Because the agent receives the same $+1$ reward at each time step it struggled to separate the good actions from the bad ones. When all rewards are strictly positive it will continue to increase the parameters θ of the policy which can lead to stability issues related to the gradients computed in the network. To counteract this problem the discounted returns the agent uses for updating the parameters was normalized, which was computed as

$$G_n = \frac{G - \mu}{\sigma}. \tag{3.7}$$

Where σ is the standard deviation and μ is the mean of the discounted returns G.

If none of the termination conditions were met after 200 time steps the simulation was terminated to avoid infinite simulations, but also to set a time limit for which the attempt at balancing the pole was considered a successful episode. This is the default maximum number of steps that is used for this environment. For successful episodes training of the network is omitted. This is because it is impossible to separate a successful episode from an unsuccessful in terms of rewards when a reward of $+1$ is always given. Hence if you train on successful episodes you will effectively unlearn the sought after behaviour.

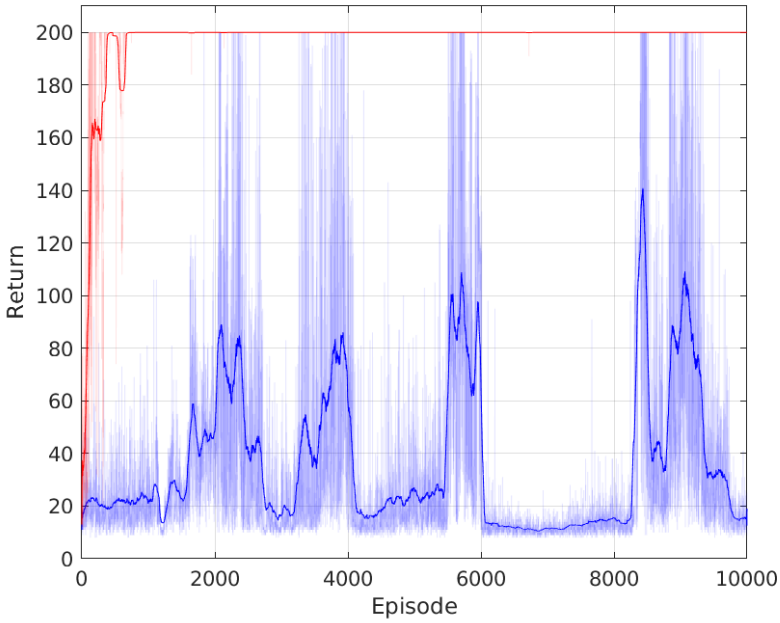


Figure 3.5: Plot showing the return history for the REINFORCE agent with (red) and without (blue) normalization in the Cartpole-v0 environment. Opaque lines are the raw return data while solid lines are smoothed versions of the raw returns.

3.3 Deep Deterministic Policy Gradient

3.3.1 Algorithm outline

The Deep Deterministic Policy Gradient (DDPG) algorithm [35] is an actor-critic algorithm that builds on the work of Silver et al [36] who found that the deterministic policy gradient exists such that deterministic policies, denoted $a = \mu(s|\theta^\mu)$, can be constructed. The deterministic policy gradient is

$$\nabla_{\theta^\mu} J \approx \nabla_a Q(s, a|\theta^Q) \nabla_{\theta^\mu} \mu(s|\theta^\mu) \tag{3.8}$$

Where θ^μ are the parameters of the policy function and θ^Q is the parameters of the action-value function, and hence $Q(s, a|\theta^Q)$ is the parameterized action-value function and $\mu(s|\theta^\mu)$ the parameterized policy function. The

deterministic policy $\mu(s|\theta^\mu)$ deterministically maps a state s to an action a given the policy parameters θ^μ.

The approach was also inspired by Mnih et al's [21] 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 handle continuous state spaces, which - as noted in Chapter 1 - was used to play several Atari games at a superhuman level. 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 [35] these are updated according to the "soft" update law:

$$\begin{aligned} \theta^{\mu'} &= (1-\tau)\theta^{\mu'} + \tau\theta^{\mu} \\ \theta^{Q'} &= (1-\tau)\theta^{Q'} + \tau\theta^{Q} \end{aligned} \tag{3.9}$$

   Where $\theta_{\mu'}$ and $\theta_{Q'}$ are the target policy- and action-value- function parameters for their respective target policy function μ' 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 the Bellman equations as in 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 **??**, but using the action-value function instead. By using the target networks to compute the returns, we get

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{s+1}|\theta^{\mu'})|\theta^{Q'}) \tag{3.10}$$

and the critic's loss function

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{3.11}$$

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:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{3.12}$$

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 5. Note that in practice the learning steps don not start before the replay buffer contains at least a mini-batch size amount of transitions.

Algorithm 5 DDPG [35]

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
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 $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}$ 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 $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{s+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i} \tag{3.13}$$

    Update the target networks:

$$\begin{aligned} \theta^{\mu'} &\leftarrow (1-\tau)\theta^{\mu'} + \tau\theta^\mu \\ \theta^{Q'} &\leftarrow (1-\tau)\theta^{Q'} + \tau\theta^Q \end{aligned} \tag{3.14}$$

  end for
end for

3.3.2 Problem description

DDPG was used to solve the environment called Pendulum-v0. The task is to swing-up and balance the pendulum in an upright position. The task is similar to the one discussed in Section 3.2.2 where we used REINFORCE, but this environment has a single continuous action instead of two discrete actions. This environment is also not under-actuated, as it only has 1 degree

of freedom while having one input that directly affects it. A visualized frame from the environment is shown in Figure 3.6.

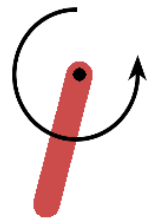


Figure 3.6: Screenshot of a single visualized frame from the Pendulum environment. The red stock is attached to a actuated rotational joint and can swing around without friction.

The system has two states, the pendulums angle to the vertical and its angular velocity. The initial conditions for the states are in the range $-\pi$ to π and -1 to -1 for the angle and angular velocity respectively. Both the sine and cosine component of the angle, and the angular velocity is available for observation. The system has one input, which is the generalized force (torque) applied by the rotational joint to the system which is in the range -2 to 2. The reward the agent receives is negative, and quadratically proportional to the angle error, the angular velocity and the force applied to the pendulum:

$$R = -\left[\theta^2 + 0.1\dot{\theta}^2 + 0.001F^2\right] \tag{3.15}$$

The reward function has its maximum in the perfect upright position where $\theta = \dot{\theta} = F = 0$. The amount of force used is penalized in order to avoid so-called bang bang control where the controller switches between the two

input extremes $F = -2$ and $F = +2$. This can cause an oscillatory behaviour close to the desired position, and in a real system this behaviour is also unwanted due to the wear and tear of the actuator and extra power consumption. Table 3.2 sums up the environment in the reinforcement learning context.

Table 3.2: Summation of the Pendulum-v0 environment.

Environment	Pendulum-v0	
Initial states	$\theta \in [-\pi, \pi], \dot{\theta} \in [-1, 1]$	uniformly distributed random values
Observations	$cos(\theta), sin(\theta), \dot{\theta}$	Continuous variables
Actions	$F \in [-2, 2]$	Continuous variables
Rewards	$R = -\left[\theta^2 + 0.1\dot{\theta}^2 + 0.001F^2\right]$	
Termination condition	Episodes lasts until $t = 200$	

3.3.3 Implementation and results

For the implementation of the DDPG agent the same hyper-parameters as in the original paper [35] was used. Two hidden layers for the critic- and actor network was used with 400 and 300 neurons respectively. The discount factor was set to $\gamma = 0.99$, minibatch size to 64, Adam optimizer with learning rate $1e4$ and $1e3$ for actor and critic respectively, the soft target updates used $\tau = 0.001$, and L2 regularization of the critic networks weight was used with a factor of 0.001. The Ornstein-Uhlenbeck process noise used $\theta = 0.15$, $\sigma = 0.2$, and $dt = 0.05$ which is the time step used by the environments internal integrator.

Figure 3.7 shows the training history returns for both training and evaluation of the agent over the training session which lasted for 1000 episodes. The returns appear quite noisy, but this is mostly due to the fact that the pendulum may start in any position, even the goal position. The time and effort needed to swing up to the right position therefore depends on the initial conditions.

Already before 200 episodes of training the agent has learned to swing up to the desired position and balance it, but as Figure 3.8 shows it is not perfect even after 1000 episodes. The agent may have learned to swing up and balance, but it fails to balance the pendulum perfectly upright and

has a stationary error where it keeps the pendulum slightly off the desired position with a constant force to support it.

This was found to be due to the γ parameter which was then increased to $\gamma = 0.999$. By increasing γ the agent will tend to value long-term rewards higher. A new agent was thus trained for 1000 episodes. The training history is shown in Figure 3.9, and as we can see by inspection, the evaluation is marginally better. By testing the new agent it was easy to see why the evaluation was better, as it now successfully manages to swing it up to a near perfect upright position while keeping it there at almost no extra cost. This is shown in Figure 3.10 for an arbitrary starting position.

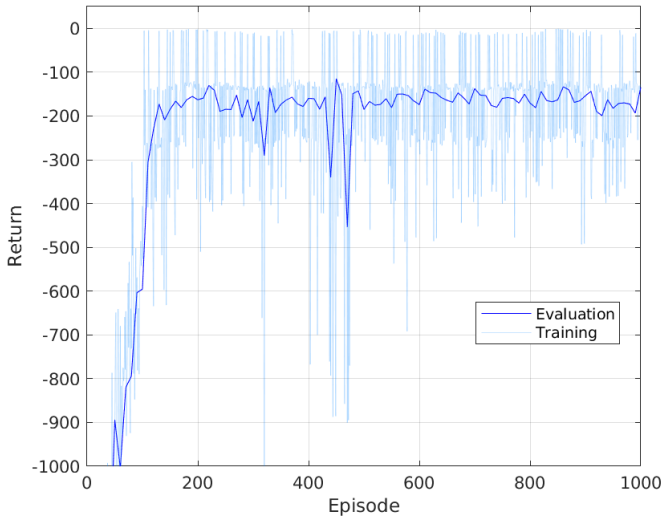


Figure 3.7: Return history during training for the DDPG agent in the Pendulum-v0 environment. Evaluation of the agent was peformed every 50'th time step, and the evaluation reward was calculated by taking the average reward over 30 trials.

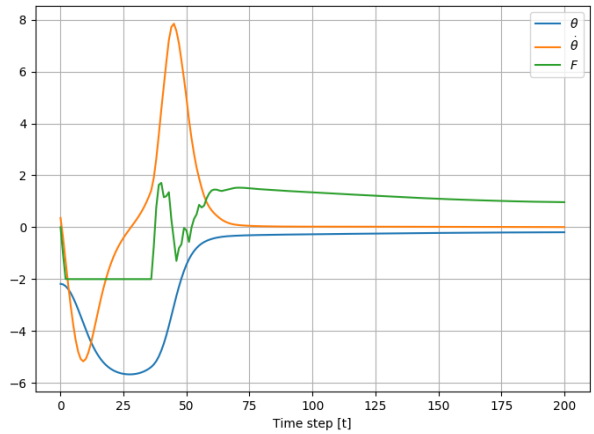


Figure 3.8: Plot showing the state variables θ and $\dot{\theta}$ and action F for a trial episode with the trained agent after 1000 episodes.

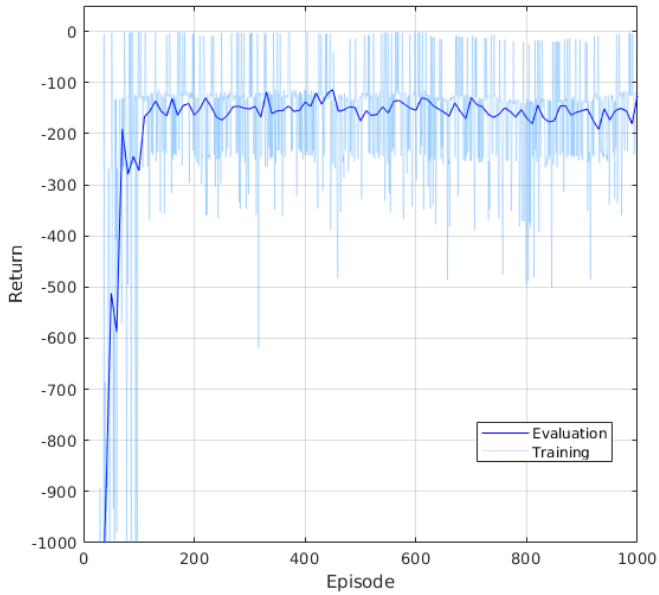


Figure 3.9: Return history for the improved DDPG agent with $\gamma = 0.999$.

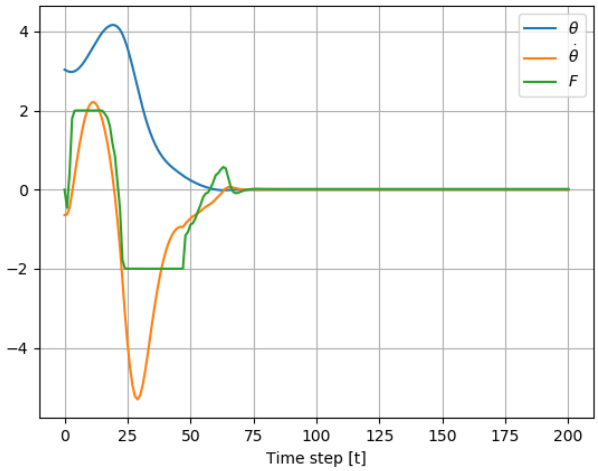


Figure 3.10: Plot showing the state variables θ and $\dot{\theta}$ and action F for the improved DDPG that used $\gamma = 0.999$.

Chapter 4

Quadrotor Altitude Control

This chapter presents the work and results of the main objective of this report, which is to perform altitude control for a quadrotor in a simulated environment. The simulation frameworks that were used to carry out the simulation and provide convenient functionality is introduced briefly and their selection justified. To simulate a quadrotor an existing implementation was used and the dynamics quadrotors and the specific implemented dynamics for the motor and propeller thrust was studied to gain a better understanding of the problem. Several designs for an altitude controller based on the DDPG algorithm was tested and the different designs was compared to each other. Lastly, the RL-based controller was compared to a traditional control design approach using a PID controller and the results are discussed with respect to performance and potential application areas for the respective controllers.

4.1 Simulation framework

The Gazebo Simulator

The robotics simulator Gazebo [37] 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 [38] (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 the format 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 any 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. However, for this project no simulated sensors were used, and the ground truth for each state variable was used instead because sensor fusion and state estimation is out of scope for this project. However, it was still an important factor for choosing Gazebo as the simulation framework because using real simulated sensors will eventually be necessary with respect to future work.

Robot Operating System

Robot Operating System [39] (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. 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, pause and un-pause it, and get the state of objects inside the

simulation.

4.2 Quadrotor simulation

To simulate a quadrotor the ”hector_quadrotor” package from the official ROS repository was used. A screenshot of the quadrotor in shown in Figure 4.1. The simulation is implemented in Gazebo and features a quadrotor model that has an implementation of the Extended Kalman Filter for state estimation using simulated sensors, a position, velocity and orientation controller implemented as separate cascaded PID controllers, and realistic motor- and propeller thrust dynamics. The controller, sensors and state estimator is not relevant for this project and they were turned off in order to speed up the simulation and is therefore not discussed further.

The dynamic models used in the design of the simulation can be found in [2] and the parameters by visiting the official ROS repository. The models used are based on wind tunnel tests and are given next together with the equations of motion for quadrotors to gain an understanding of the dynamics of the system and for the sake of completeness.

For a quadrotor modelled as a rigid body moving in a 3-dimensional space with forces F and torques M acting on it the equations are:

$$\begin{aligned} \dot{\boldsymbol{p}}^{\boldsymbol{n}} &= \boldsymbol{v}^{\boldsymbol{n}} \\ \dot{\boldsymbol{v}}^{\boldsymbol{n}} &= m^{-1}\boldsymbol{C}_{\boldsymbol{b}}^{\boldsymbol{n}}\boldsymbol{F} \\ \dot{\boldsymbol{\omega}}^{\boldsymbol{b}} &= \boldsymbol{J}^{-\mathbf{1}}\boldsymbol{M} \end{aligned} \tag{4.1}$$

where p^n and v^n are the position and the velocity of the body's center of gravity in the inertial navigational frame. The angular rate ω^n is expressed in the body frame coordinates and C_b^n the rotation matrix parameterized by the objects pitch, roll and yaw angles that transform a vector from the inertial- to the body frame. The mass m and inertia $\boldsymbol{J}$ of the quadrotor needs to be known and can be calculated by knowing the weight of all the components of the quadrotor. The relation between the frames and the forces and torques acting on the quadrotor body is shown in Figure 4.2.

The forces and torques acting on the system can be divided into drag- and motor thrust induced components as well as the gravity force. The drag induced forces are:

$$\begin{aligned} \boldsymbol{F_d} &= -\boldsymbol{C_{d,F}} \cdot \boldsymbol{C_n^b} \cdot |\boldsymbol{v^n} - \boldsymbol{v_w^n}|(\boldsymbol{v} - \boldsymbol{v_w}) \\ \boldsymbol{M_d} &= -\boldsymbol{C_{d,M}} \cdot |\boldsymbol{\omega^b}|\boldsymbol{\omega^b} \end{aligned} \tag{4.2}$$

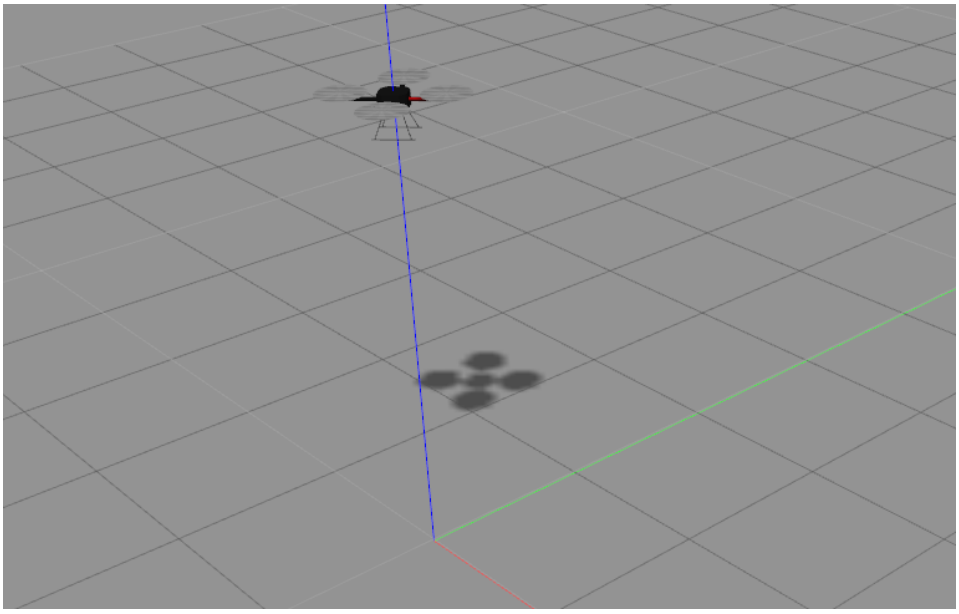


Figure 4.1: A screenshot of the simulated quadrotor. Since no cameras or other sensors were used graphics were not important and therefore only a plain and gray grid plane was present in the simulation beside the quadrotor itself.

Where $\boldsymbol{C_{d,F}}$ and $\boldsymbol{C_{d,M}}$ are the diagonal drag coefficient matrices. The gravity force is given by:

$$\boldsymbol{F_g} = m \cdot \boldsymbol{C_n^b} \cdot \begin{bmatrix} 0 & 0 & g_e \end{bmatrix}^T \tag{4.3}$$

The propulsion system consists of four brushless DC motors and the rotor speed dynamics of each motor depends on the induced anchor voltage and electromagnetic torque produced by the motors:

$$\begin{aligned} U_a &= R_a I_a + \psi \omega_M \\ M_e &= \psi I_A. \end{aligned} \tag{4.4}$$

The rotor speed dynamics of each motor:

$$\dot{\omega_M} = \frac{1}{J_m} \cdot (M_e - M_m) = \frac{1}{J_m} \cdot \left[\frac{\psi}{R_A} \cdot (U_A - \psi\omega_M) - M_m \right] \tag{4.5}$$

where the $M_m = k_T \cdot T$ where k_T is a constant and T the thrust. This term accounts for the bearing- and drag friction of the rotors.
Next the thrust forces can be calculated by using the motor speed dynamics given by (4.5):

$$T = C_{T,0}\omega_M^2 + C_{T,1}v_1\omega_M \pm C_{T,2}J^2 \tag{4.6}$$

The thrust coefficient $C_T(J)$ are found by dividing the former equation by ω_M^2 and using the performance factor $J = v_1/\omega_M$:

$$C_T(J) = C_{T,0} + C_{T,1}J \pm C_{T,2}J^2 \tag{4.7}$$

where the parameters $C_{T,i}$ have been identified in a wind tunnel. The velocity of each rotor is in general different and can be found if the linear- and angular velocity together with the distance between the rotors and geometric center l_M is known:

$$(v_1)_i = -\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \cdot \left(v^b + (\omega^b \times e_i) \cdot l_M \right) \tag{4.8}$$

where e_i are the units vectors for each motor. By using the numbering shown in Figure 4.2 one can see that they must be $e_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$, $e_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$, $e_3 = -\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$, and $e_4 = -\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$. Finally, the thrust- forces and torques that gives the overall wrench of the quadrotor can be found by using (4.6) to find the resulting force produced by each rotor:

$$\begin{aligned} \boldsymbol{F}_M^b &= \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^4 F_i \end{bmatrix} \\ \boldsymbol{M}_M^b &= \begin{bmatrix} (F_4 - F_2) \cdot l_M \\ (F_1 - F_3) \cdot l_M \\ -M_1 + M_2 - M_3 + M_4 \end{bmatrix} \end{aligned} \tag{4.9}$$

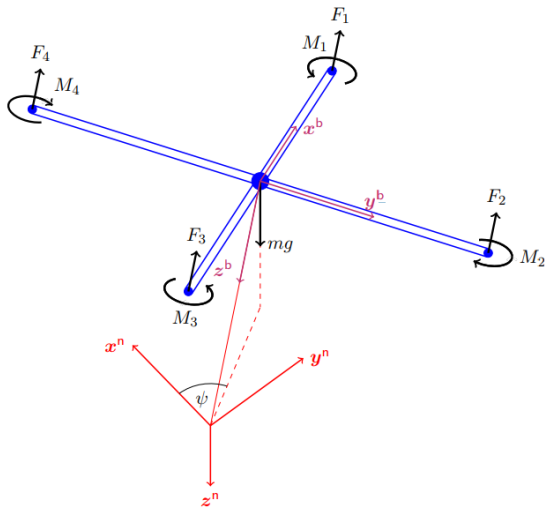


Figure 4.2: Sketch of a quadrotor [2] showing the moments and forces each of the rotors exert on the body, the center of gravity and the relation between the body and inertial frame.

Then by using equations (4.1) and (4.9) the motion of the quadrotor can be solved for. The calculation of thrust- and drag forces acting on the quadrotor are implemented as two Gazebo plugins, and the full motion of the quadrotor is then calculated by the Open Dynamics Engine.

It is worth mentioning that there is two crucial modelling shortcoming with respect to realistically simulating a quadrotor, one being the effect of so called blade flapping. This effect is significant in both small and large scale aircrafts, but since the task we are interested in is altitude control, meaning that the horizontal movement of the rotors is minimal, the effect it has is minimal. The second is the nonlinear forces present when the quadrotor is close to the ground due to aerodynamics, but since we're only interested in altitude control we consider only in-air initialization and the ground is thought to not exist. More on quadrotor modelling can be found in [40] [41], and why the unmodelled dynamics of the system mentioned above is neglected is justified by a simplification of the problem that is introduced in the next section.

4.3 Altitude control

Next two controllers for doing altitude control is presented and their performance shown using the simulation environment introduced in the previous section. To restrict the problem to altitude control only, one crucial simplification was made: by initializing the quadrotor with inclination identical to zero the quadrotor will not tilt as long as all the four rotor thrusts are equal. Hence, the controllers only needed to output one voltage which was used for all four motors. While this is of course unrealistic, as even the slightest offset or disturbance will cause the quadrotor to become unstable in the real-world, it still serves well as a way to restrict the scope of the problem. In effect this means that we assume that a perfect orientation controller for stabilizing the pitch and roll of the quadrotor is already implemented and/or there is no disturbances such as wind-gusts acting on the quadrotor.

The controller rate of all controllers were set to 100Hz such that each action/input was repeated for 10ms, and the state variables used are the noiseless ground truth variables. Fast response of controllers, zero steady state error, and smooth behaviour of the input to the system was the main criteria for evaluating controllers.

4.3.1 Using reinforcement learning

The environment

There were mainly four different states that were used in the design of controllers: the positional error in the z-direction, defined as the difference between the quadrotor z position and the desired z position $\mathrm{z_d}$, $\tilde{\mathrm{z}} = \mathrm{z} - \mathrm{z_d}$; the velocity in the z-direction, $\dot{\mathrm{z}}$; the acceleration in the z-direction, $\ddot{\mathrm{z}}$; and the previous input to the environment produced by the actor network, u. The previous input was used as a state instead of the actual action to keep all the state variables continuous, and keep all the states in a similar range so input normalization was not necessary.

To ensure that the agent continuously explored a wide set of different starting states the initial state variables were set to random values. The quadrotor was initialized in a random position in the range 1 to $5m$, with an initial velocity between -2 and +2$\frac{m}{s}$, and acceleration and motor voltages equal to zero. It is worth noting that setting the initial value $\mathrm{u} = \ddot{\mathrm{z}} = 0$ was not a choice: it was not possible to decide their initial values.

The output u of the actor network, which is -1 to 1, were mapped to be in the discrete range of accepted voltage levels for the motors, ranging from 0 to 255. This voltage, denoted v, is the action decided by the agent. More precisely the voltage level is a 8-bit encoder value that maps to the actual voltage level by the simulation itself. Each episode in the environment had a timestep limit of 500, which is equal to $5s$ using a 100Hz controller loop. To restrict the training area and the range for which the actor network had to generalize, the episodes were terminated if the quadrotor flew more than $2m$ away from $\mathrm{z_d}$, which was set to 3 in all cases. The environment as defined in the reinforcement learning sense above is summarized in Table 4.1.

Table 4.1: Summation of the simulated quadrotor environment.

Environment	Quadrotor	
Initial states	$\tilde{\mathrm{z}} \in [2, 4]$ $\dot{\mathrm{z}} \in [-2, 2]$ $\ddot{\mathrm{z}} = 0$ $\mathrm{u} = 0$	uniformly distributed random values
Observations	$\tilde{\mathrm{z}}, \dot{\mathrm{z}}, \ddot{\mathrm{z}}, \mathrm{u}$	Continuous variables
Actions	$\mathrm{v} \in [0, 255]$	Discrete variables
Termination condition	$\lvert\tilde{\mathrm{z}}\rvert > 2$ or $n = 500$	

Algorithm

The DDPG algorithm from Section 3.3 was used to create an altitude controller. It was found that in the algorithm required a very high discount factor to converge to a steady state error with $\tilde{\mathrm{z}} = 0$, and therefore $\gamma = 0.99999$ was used. This caused a problem related to the size of action-values $Q(s, a)$ in the critic network due to the use of L2 regularization in the original algorithm. This becomes apparent when considering the upper bound of a value function $V(s)$ for a state s given a maximum reward r_{max}, which is:

$$V(s) \leq \sum_{n=0}^{\infty} r_{max}\gamma = \frac{r_{max}}{1-\gamma}. \tag{4.10}$$

Hence, if the maximum reward the agent can experience is 1, the maximum value function would be $\frac{1}{1-0.99999} = 100000$. The use of L2 regularization

of the critic networks was therefore omitted, and rewards were kept low to avoid numerical instability in the networks. The other hyper-parameters of the algorithm were kept the same.

To evaluate the goodness of the controller a test was constructed where the quadrotor was initialized at three different positions with zero initial velocity: one meter below the desired altitude, at the desired altitude, and one meter above the desired altitude. The evaluation score was then calculated to be the average reward of the three scenarios.

One learning step was performed for each step in the environment, and evaluation was performed between every episode of learning. In general, training was not particularly stable and so the evaluation fluctuated rapidly between episodes. To obtain the best possible solutions, the algorithm was therefore altered to include a form of precision training: stop generating new training data when the evaluation reaches a certain level, and instead sample the replay buffer and train for only 10 steps at a time between evaluations. The decision for when to enter precision training was decided manually based on experience of what constituted a high return.

Reward functions

The first reward function used was a naive approach only incentivising $\tilde{\mathrm{z}} \rightarrow 0$ by giving maximum reward when $\mathrm{z} = \mathrm{z_d} \rightarrow \tilde{\mathrm{z}} = 0$ and monotonically increasing in the direction of $\mathrm{z_d}$:

$$R_1 = K_{\tilde{z}} e^{-|\tilde{z}_t|} \tag{4.11}$$

where $K_{\tilde{z}}$ was a constant set to $1\mathrm{e}{-2}$ and the subscript n denotes the timestep. The agent used the position, velocity and acceleration for state variables with this reward function. This did however not work at all, as the agent would always learn to either put maximum- or minimum force all of the time no matter the circumstance. A lesser state representation using a combination of only the position and velocity or acceleration was also tested, but gave very poor results,

The learning issue was speculated to a consequence of the slowness of the rotor wind-up dynamic compared to the controller loop. And so to combat this problem a new reward function was devised that applied a simple piece of domain knowledge: to produce force equal to the force of gravity and a voltage level of 121 is required. This is the minimum required

force necessary to hover, and so it makes sense to guide the agent towards this input. Denoting the voltage that produces a force equal to the force of gravity as $\mathrm{v_g}$ and the maximum voltage as $\mathrm{v_{max}}$ we write the second reward function as:

$$\begin{aligned} R_2 &= K_{\tilde{\mathrm{z}}} e^{-|\tilde{z}_\mathrm{t}|} - K_\mathrm{v} \left[\frac{\mathrm{v_t} - \mathrm{v_g}}{\mathrm{v}_{max}} \right]^2 \\ &= R_1 - K_\mathrm{v} \left[\frac{\mathrm{v_t} - \mathrm{v_g}}{\mathrm{v}_{max}} \right]^2 \end{aligned} \tag{4.12}$$

where K_v is a constant which was set to $1\mathrm{e}{-3}$, and $\mathrm{v_{max}}$ is used as a normalizing constant. The agent proceeded by adopting a strategy reminiscing the infamous bang-bang control that were avoided in Section 3.3 by introducing a penalty to the square of the force used. The evaluation of the best agent obtained is shown in Figures 4.3-4.5, and the evaluation reward history is shown in Figure 4.6.

A reward function using the squared position error $\tilde{\mathrm{z}}^2$ in the exponent of the Euler function instead of the absolute value was also experimented with. This however yielded worse results as the agent struggled to find that $\tilde{\mathrm{z}} = 0$ gives the most reward, and convergence to good solutions took longer time in general.

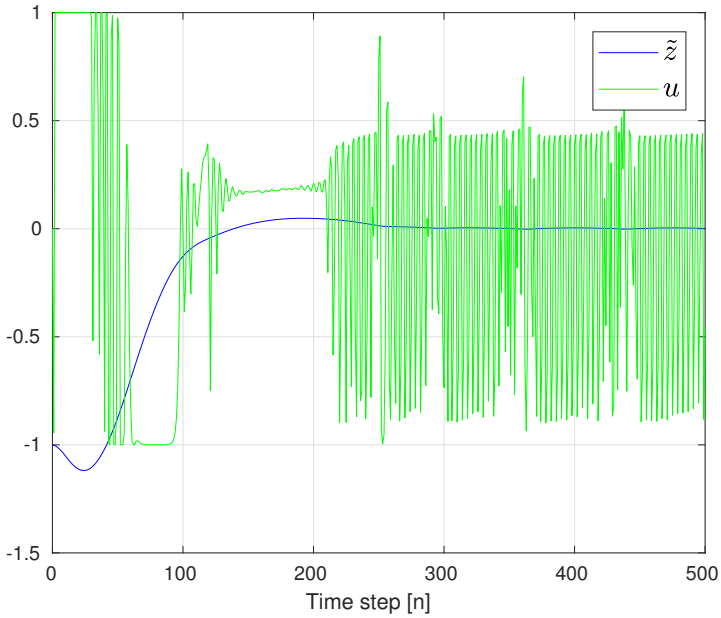


Figure 4.3: Test case 1 for R_2: $\tilde{z_0} = -1$.

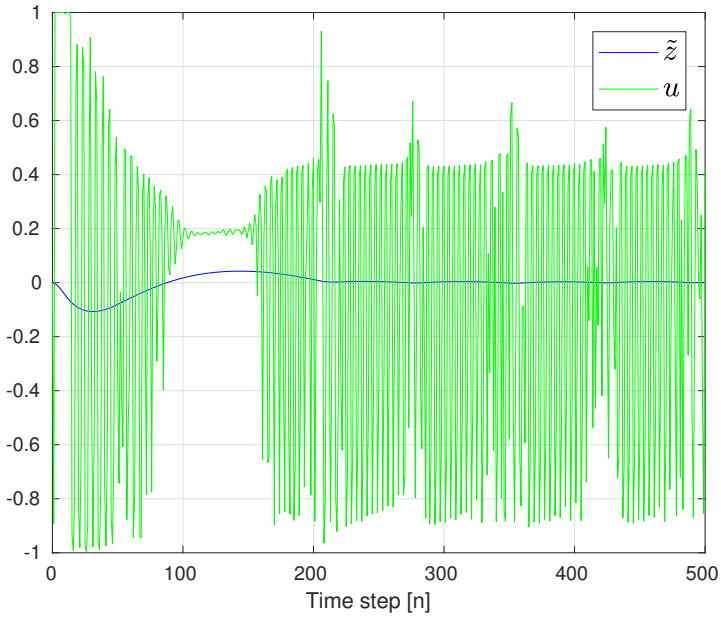


Figure 4.4: Test case 2 for R_2: $\tilde{z_0} = 0$.

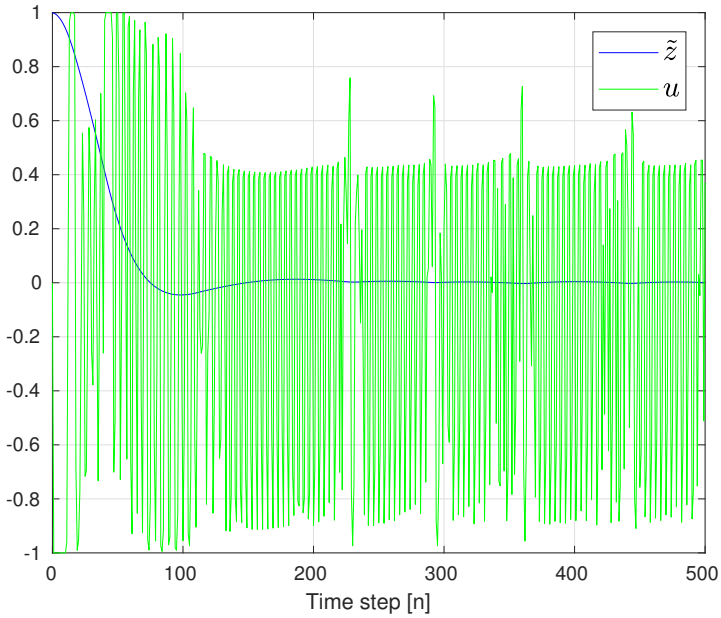


Figure 4.5: Test case 3 for R_2: $\tilde{z_0} = 1$.

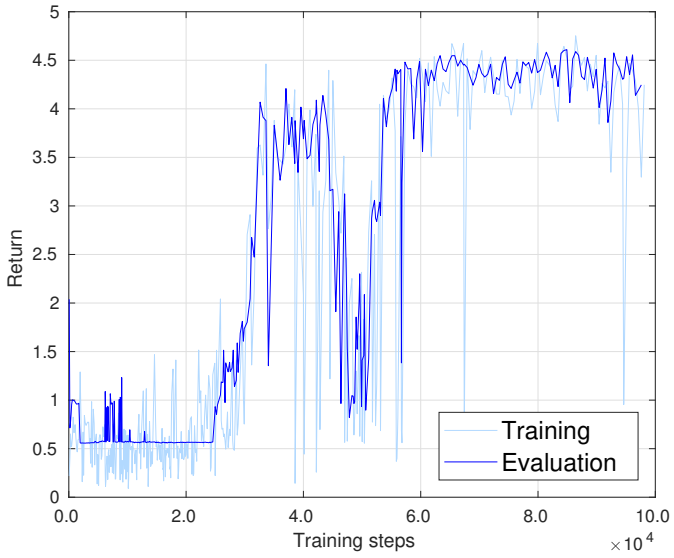


Figure 4.6: Return history for the agent using reward R_2. Training data is the return of single episodes and evaluation data is the average of the three test cases as previously defined.

To counter the problem associated with R_2, the current motor voltage, or equivalently, the previous input to the system u was used in the state augmentation for the system to allow the agent to pick actions depending on the previous action it picked. A new reward function was proposed:

$$\begin{aligned} R_3 &= K_{\tilde{\mathrm{z}}} e^{-|\tilde{z}_\mathrm{t}|} - K_\mathrm{v} \left[\frac{\mathrm{v_t} - \mathrm{v_g}}{\mathrm{v}_{max}} \right]^2 - K_{\Delta\mathrm{v}} \left[\frac{\Delta \mathrm{v}}{\mathrm{v}_{max}} \right]^2 \\ &= R_2 - K_{\Delta\mathrm{v}} \left[\frac{\Delta \mathrm{v}}{\mathrm{v}_{max}} \right]^2 \end{aligned} \tag{4.13}$$

where $\Delta \mathrm{v} = \mathrm{v_t} - \mathrm{v_{t-1}}$ and $K_{\Delta\mathrm{v}}$ a constant set to $1\mathrm{e}{-3}$. This reward function penalizes change in the input to the system. The evaluation of the best agent obtained is shown in Figures 4.7-4.9, and the evaluation reward history is shown in Figure 4.10. While this greatly reduced the the fluctuations in u, it did not eliminate the erratic behaviour completely, and it is easy to see even for the human eye that that the given solution does not follow an optimal policy. The learning process was also uncertain in the sense that the agent was not guaranteed to converge to a good solution in any reasonable time, and therefore several runs of the algorithm was required to obtain the results shown here.

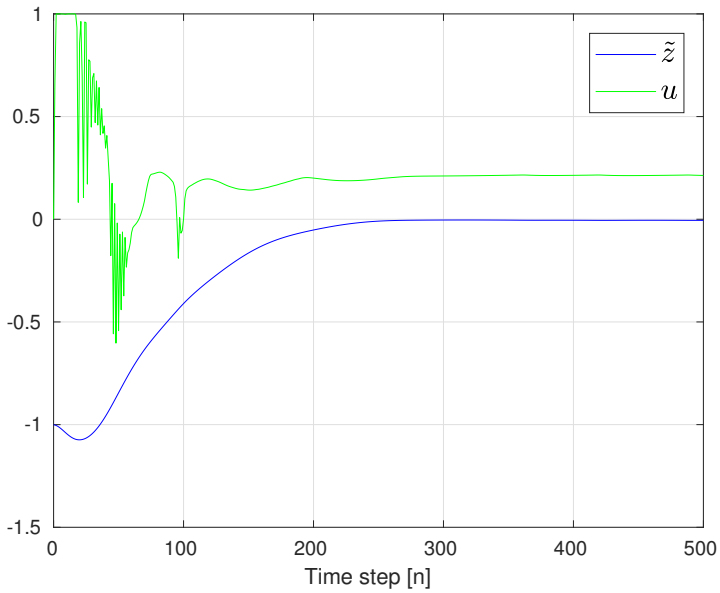


Figure 4.7: Test case 1 for R_3: $\tilde{z_0} = -1$.

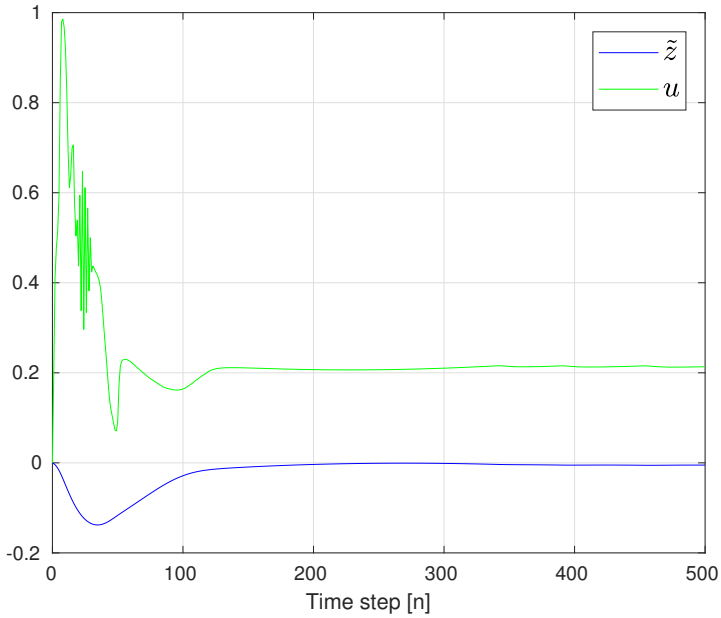


Figure 4.8: Test case 2 for R_3: $\tilde{z_0} = 0$.

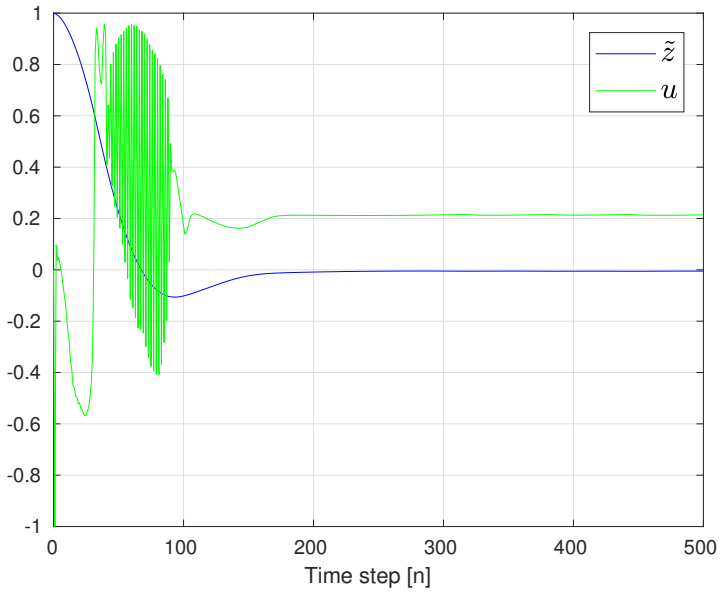


Figure 4.9: Test case 3 for R_3: $\tilde{z_0} = 1$.

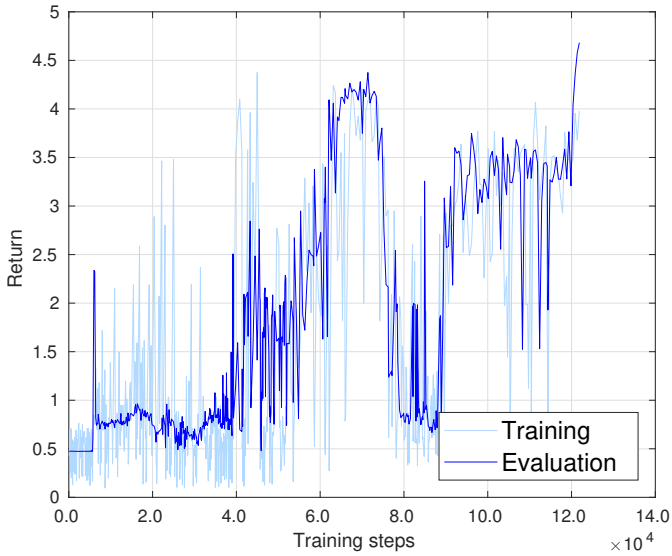


Figure 4.10: Return history for the agent using reward R_3. Training data is the return of single episodes and evaluation data is the average of the three test cases as previously defined.

The issues with the learning process related to R_3 was thought to be a result of the two penalties being conflicting: they both try to penalize the input to the system, and so learning about this penalty becomes difficult. A different reward function was therefore proposed to incentivize velocity in the direction of the desired altitude $\mathrm{z_d}$ instead of the derivation of v from $\mathrm{v_g}$ that was used in R_2 and R_3:

$$R_4 = \begin{cases} K_{\tilde{z}} e^{-|\tilde{z}_t|} - K_{\Delta \mathrm{v}} \left[\frac{\Delta \mathrm{v}}{\mathrm{v}_{max}} \right]^2, & \text{if } \mathrm{sgn}(\dot{z}) \neq \mathrm{sgn}(\tilde{z}) \text{ or } |\dot{z}| < \epsilon) \\ -K_{\Delta \mathrm{v}} \left[\frac{\Delta \mathrm{v}}{\mathrm{v}_{max}} \right]^2, & \text{otherwise} \end{cases} \tag{4.14}$$

where $K_{\dot{z}}$, $K_{\Delta \mathrm{v}}$ as before and u used in the state augmentation as for R_3. The constant ϵ was set to $0.02 \frac{m}{s}$ and was designed such that the agent was allowed to move with small velocities even though it was moving away from the target. It was included in order to avoid an oscillating movement around the setpoint $\mathrm{z_d}$.

The evaluation of the best agent obtained is shown in Figures 4.11-4.13, and the evaluation reward history is shown in Figure 4.14. The erratic

behaviour in the input was nearly gone and the performance overall was much better as the agent's response was faster, $\tilde{z}$ converged to zero, and the velocity converged to zero as well. Training was also more stable where an agent would almost always converge to a good solution.

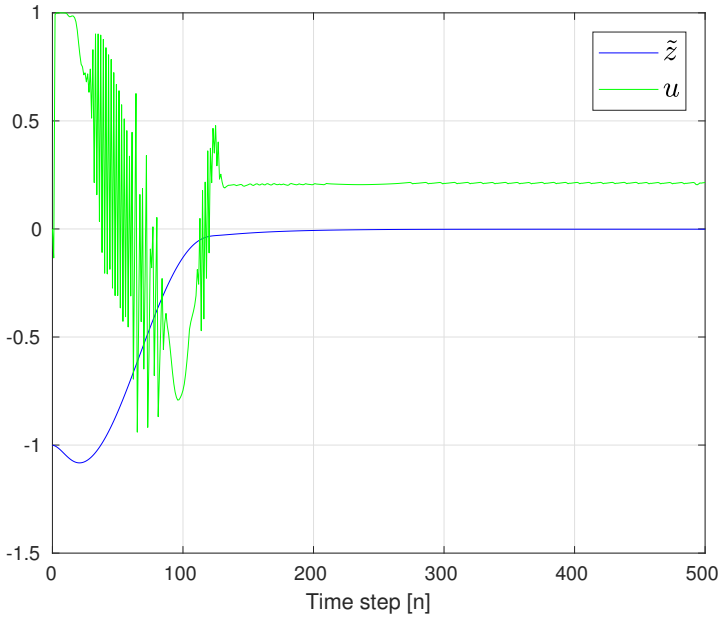


Figure 4.11: Test case 1 for R_4: $\tilde{z_0} = -1$.

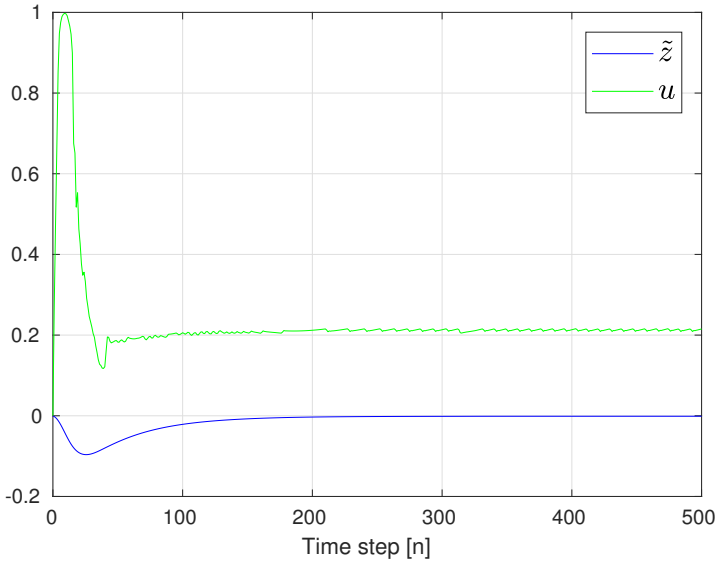


Figure 4.12: Test case 2 for R_4: $\tilde{z_0} = 0$.

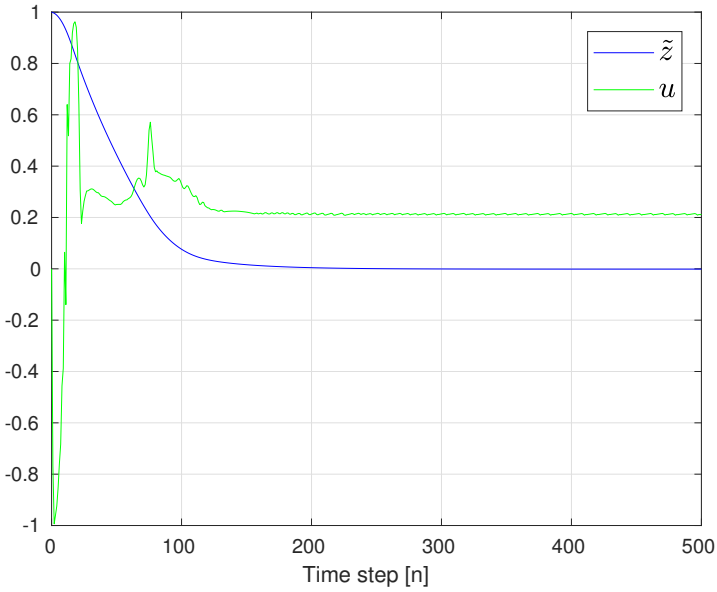


Figure 4.13: Test case 3 for R_4: $\tilde{z_0} = 1$.

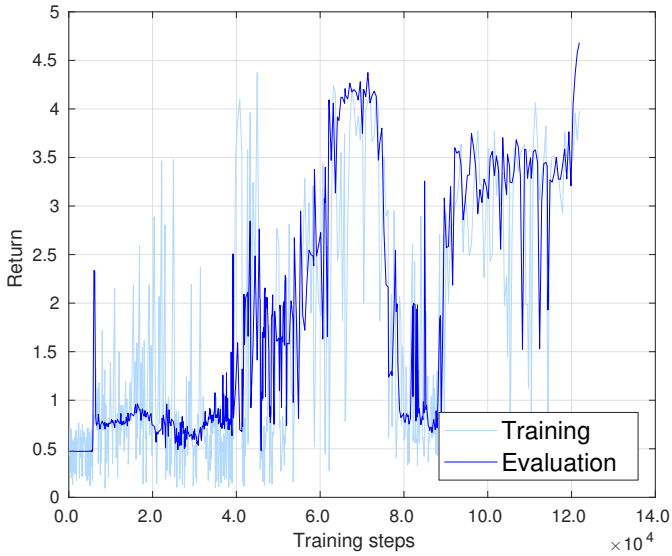


Figure 4.14: Return history for the agent using reward R_4. Training data is the return of single episodes and evaluation data is the average of the three test cases as previously defined.

4.3.2 Using a PID controller

A proportional-integral-derivative (PID) controller was picked to represent the control theory approach in the project because it is a very popular choice in the literature when it comes to altitude control of quadrotors. Figure 4.15 shows the block diagram of the controller, where the position error $\tilde{z}$ is used as input and the 'T' block is the mapping from continuous to discrete voltage levels in the range 0 to 255. The PID controller used clamping to prevent integrator wind-up, limiting the integrator term to 0.5. Since ground-truth state observations were used no extra features were used to prevent spikes from the derivative term that is associated with noisy sensors. A value equal to the voltage required to produce force equal to the force of gravity, referred to as $\mathrm{v_g}$ in Section 4.3.1, was added to the output of the controller in order to center the contribution of the PID controller around the stabilizing input which is $\mathrm{v_g}$.

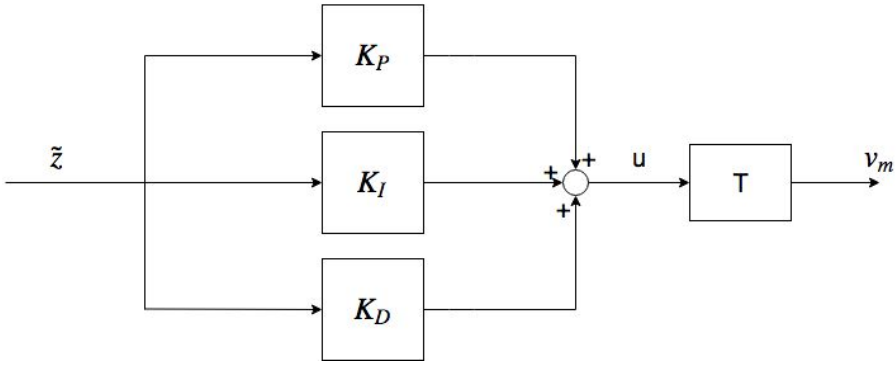


Figure 4.15: Block diagram of the PID controller.

To tune the controller a single-objective genetic algorithm was used based on [42][43]. To evaluate the goodness of each set of PID controller gains the fitness used was:

$$fitness = \frac{1}{N}\sum_{n=0}^{N}\tilde{z}^2 \tag{4.15}$$

Where the trajectory $\tilde{z}$ depends on the PID gains K_P, K_I and K_D. The objective subject to optimization is therefore:

$$\text{argmin}_{K_P,K_I,K_D}\left[\frac{1}{N}\sum_{n=0}^{N}\tilde{z}^2\right] \tag{4.16}$$

Where n is the timestep and N is the length of the episode, where $N = 500$ was used as with the RL-based controller. The same test that was used to evaluate the RL-based controller was used to evaluate the fitness of each set of PID gains as well.

As for the genetic operators the mutation operator added a value from the normal distribution $\mathcal{N}(0, \sigma)$ with $\sigma = 0.05$ to one of the PID gains picked at random, and the crossover operator performed an single arithmetic recombination for one of the PID gains picked at random. For each individual created the chance of performing a mutation was 0.4 and crossover 0.6. A population size of 50 was used with a generation size of 60. Elitism was used to transfer the best $5\%(= 3$ rounded up$)$ of individuals from the previous generation to the next.

The best PID gains found with this tuning method for this particular problem were the following:

$$K_P = 2.239,\ K_I = 0.056,\ K_D = 1.041 \tag{4.17}$$

and the performance on the test cases are shown in Figure 4.16-4.18. The performance overall was good, but the performance of the controller was highly dependent on the starting position: starting beneath the setpoint resulted in a slight overshoot while starting at the setpoint or above resulted in no overshoot at all. The controller also had problems removing the steady-state error.

Tuning a PI controller was also tested, but failed to give reasonable results. The slowness of the rotor wind-up dynamics and the overall nonlinear dynamics of the system can contribute to explain why this approach failed.

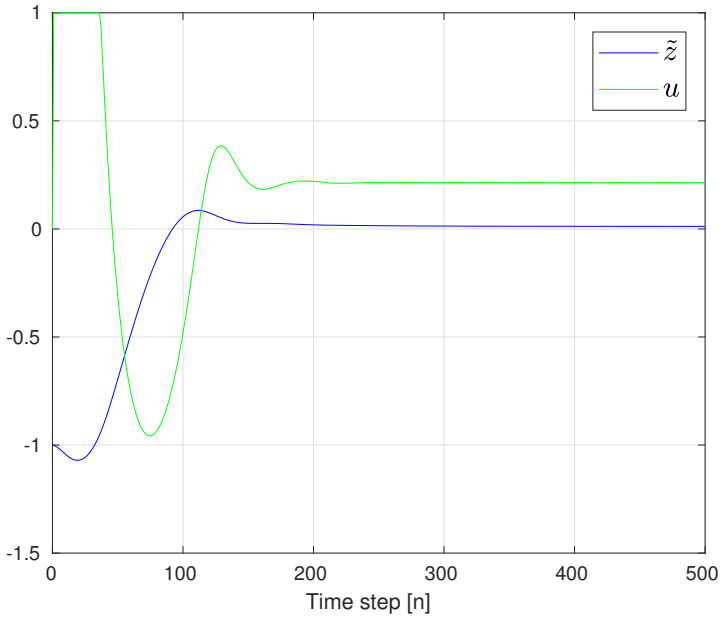


Figure 4.16: Test case 1 where $\tilde{z_0} = -1$ using the PID controller.

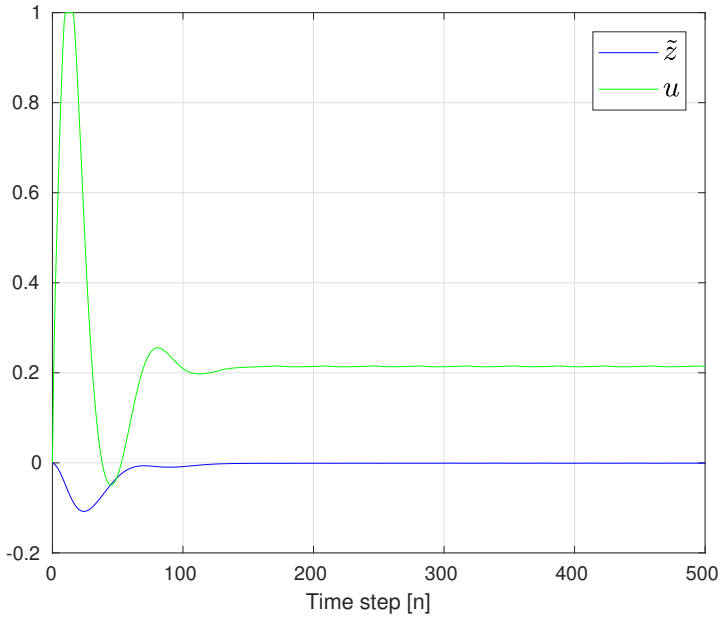


Figure 4.17: Test case 2 where $\tilde{z_0} = 0$ using the PID controller.

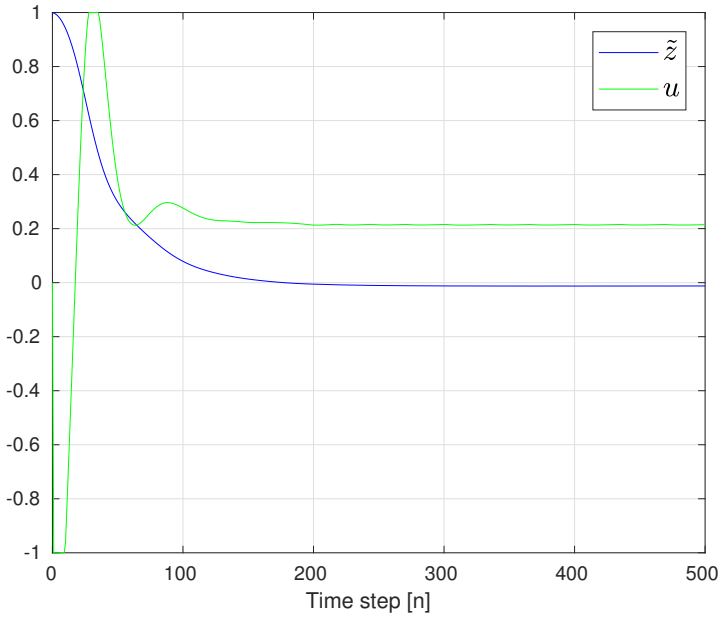


Figure 4.18: Test case 3 where $\tilde{z_0} = 1$ using the PID controller.

4.4 Comparison and discussion

The performance of the reinforcement learning based controller using reward function R_4 from Section 4.3.1 and the PID based controller from Section 4.3.2 was compared using the test cases defined in Section 4.3. The plots are shown in Figure 4.19-4.21.

From the plots it can be seen that that the RL-based controller design exhibited a more erratic control input behaviour, and converged much faster to a steady state error. The PID controller on the other hand has a faster response time, but has some overshoot for the case where $\tilde{z_0} = -1$, and the behaviour of the step response for $\tilde{z_0} = -1$ and $\tilde{z_0} = 1$ was different as opposed to the RL-based controller which had responses that were very much identical. This shows that the RL-based controller was actually more predictable when it comes to trajectory it took to reach the goal, and it's performance was not dependent on the starting position.

The difference in speed of convergence of $\tilde{z}$ to zero for both controllers were small, where the PID controller had a slightly faster response, but only the RL-based controller managed to drive the steady state error to zero in all three cases shown. It is possible to derive a PID controller that has zero steady state error as well, but it would come at a cost of a slower response. This trade-off is based on preference and the application area of the controllers, like how much overshoot is acceptable, and what is best comes down to the particular case. The results do however suggests that the RL-based controller may give better performance with respect to the trade-off between fast response, steady state error and overshoot. This of course is only reasonable as the RL controller is a nonlinear controller that can exhibit far more sophisticated behaviour than the linear PID controller.

For the price of a more complex controller comes also the cost of increased computational time to compute controller inputs. For the controllers written in Python with a Intel i5-7600k processor and a Nvidia 1060 GTX graphics card it took approximately $8\text{e}{-4}$ ms and $4\text{e}{-6}$ ms respectively for the RL- and PID controller to compute a single input. This a difference of two orders of magnitude, which is significant. The difference would be even higher if it was not for the neural networks being GPU accelerated. It may therefore not be feasible to run the RL-based controller on smaller platforms which often use less computationally capable devices such as micro-controllers. This is also fortified by the fact the RL controller uses high level libraries such as Tensorflow that may not be available on a

more bare-bone operative system.

Altogether, considering the potential performance gain discussed above, computational times, and input behaviour, the results here may suggest that the use of a RL-based controller is better suited for higher-end systems that require higher precision with respect to control. However, more research should be put into both PID- and the RL controller before any assertive conclusion can be drawn about the comparative performance of the controllers. This includes using more realistic environments that have noisy sensors and dynamics and that is not restricted to only moving along the z-axis.

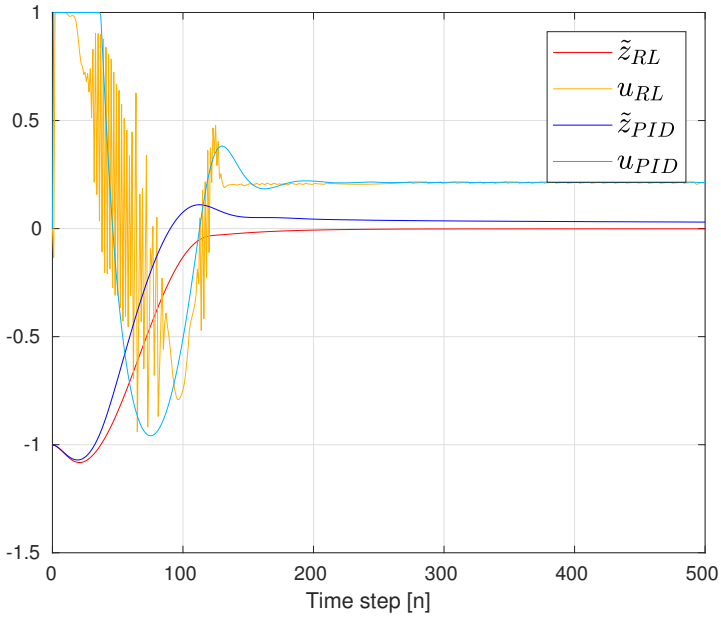


Figure 4.19: Plot of test case 1 where $\tilde{z_0} = -1$ showing the performance of the RL controller using reward R_4 defined in 4.3.1 versus the PID controller from 4.3.2.

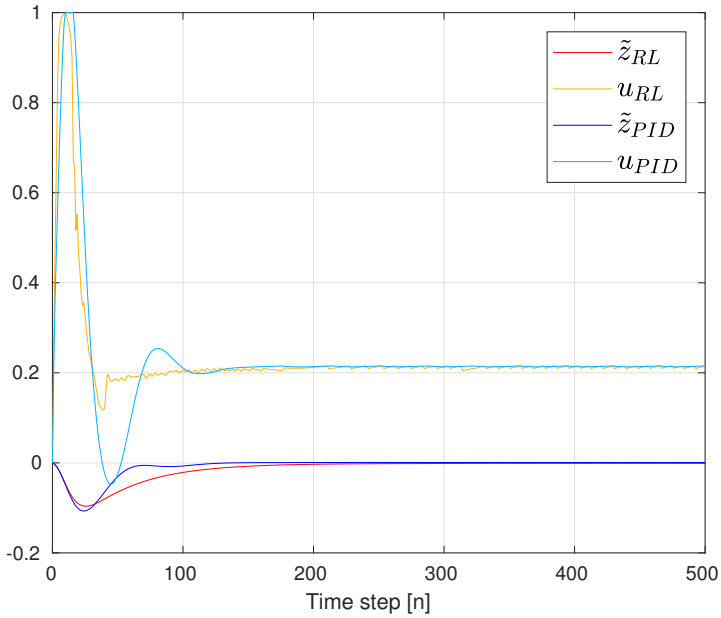


Figure 4.20: Plot of test case 2 where $\tilde{z_0} = 0$ showing the performance of the RL controller using reward R_4 defined in 4.3.1 versus the PID controller from 4.3.2.

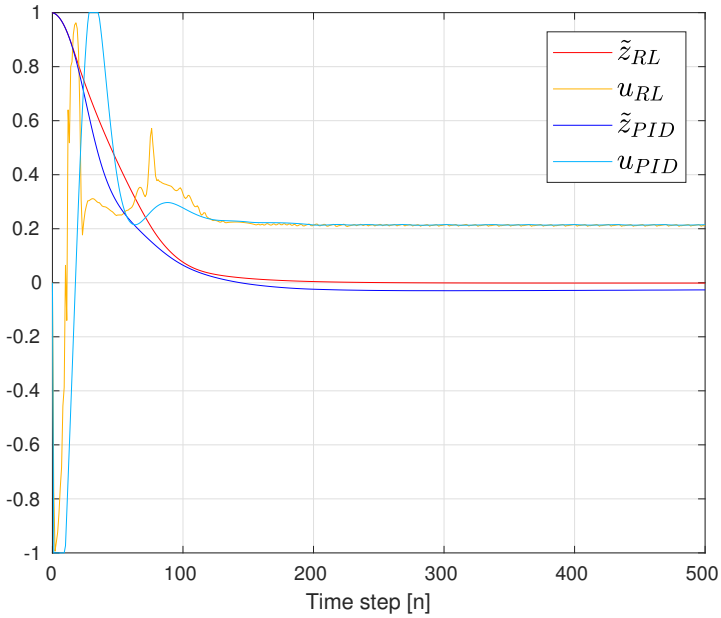


Figure 4.21: Plot of test case 3 where $\tilde{z_0} = 1$ showing the performance of the RL controller using reward R_4 defined in 4.3.1 versus the PID controller from 4.3.2.

4.5 Future Work

The results discussed in Section 4.4 were not decisive in determining whether one approach was better than the other. It is clear that the parameters of each of the algorithms and controllers were not optimally tuned, and it would therefore be interesting to perform parameter optimization in order to draw a stronger conclusion about the results that was obtained. The genetic algorithm that was used to tune the PID controller did not have a way to specifically evaluate the steady state error of the controller gains, which can explain why the best gains it found resulted in some overshoot and steady state error. A possible solution to this is to explore the use of a multi-objective genetic algorithm (MOGA) instead to define a trade-off between the three objectives that is steady-state error, response time and overshoot.

The RL-based controller was not tuned optimally either, and more experimentation with reward functions could yield better results. As mentioned in Section 4.3.1, the training was unstable and convergence was not

certain. A possible further improvement could therefore also be to investigate other reinforcement learning based methods. One interesting approach that has emerged recently is the Proximal Policy Optimization (PPO) algorithm [44] which has been shown to give competitive results to other methods for robotic applications while being more stable and reliable.

While the RL-based controller performs well in the simulated environment, the real-world is far more complex, and so it would be interesting to apply the proposed controller to a real-world system. Unfortunately, training from scratch in the real-world is generally not feasible. This is due to the potential hazard of destroying the quadrotor every time the RL controller takes over the control, and the time consuming and tedious work that is involved in resetting the quadrotor, charging batteries and not having the luxury of generating training samples at a speed higher than real-time. Performing transfer learning with the learned controller to a real system is therefore interesting to investigate.

In this project noiseless, ground-truth variables was used as states from a simulated environment, but real-world sensors are of course noisy. Reinforcement learning is in general designed to handle stochastic environment such that an agent can still perform optimally under the influence of noise and so it would be interesting to see how the proposed controllers perform in the presence of noisy state variables. The algorithm used, Deep Deterministic Policy Gradient[35], has also been shown to produce good results only using raw pixels as input to the networks. It is therefore compelling to explore the possibility of performing altitude control and other control tasks using a camera as this would reduce the need for state estimation in the system and other expensive sensors and systems.

Chapter 5

Conclusion

During this project a strong theoretical foundation was built and practical experience was obtained with reinforcement learning. Both older- and newer methods were studied and their application areas acquainted with. A close understanding of approximate methods that use neural networks as function approximators has been obtained. The use of these methods require experience with neural networks and design of reward functions that can only be obtained by experimenting with them in practise.

Valuable knowledge was especially obtained in the robotics control domain where a controller based on one of the state-of-the-art algorithms in reinforcement learning, DDPG, was designed and successfully applied to the quadrotor altitude control task. Along with the dynamics of quadrotors that was studied this project has functioned as a stepping to be able to solve more complex tasks related to quadrotors and control in the future.

The main results were not conclusive with respect to the superiority of either one of the controllers, but it did at least give a strong implication that reinforcement learning is a competitive approach to control theory. The next years research certainly has the potential to bring breakthroughs to both fields.

Bibliography

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

[2] 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.

[3] M. Kamel, M. Burri, and R. Siegwart, "Linear vs nonlinear mpc for trajectory tracking applied to rotary wing micro aerial vehicles," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 3463–3469, 2017.

[4] A. Nemati and M. Kumar, "Non-linear control of tilting-quadcopter using feedback linearization based motion control," in *ASME 2014 Dynamic Systems and Control Conference*, pp. V003T48A005–V003T48A005, American Society of Mechanical Engineers, 2014.

[5] T. Madani and A. Benallegue, "Backstepping control for a quadrotor helicopter," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pp. 3255–3260, IEEE, 2006.

[6] L. M. Argentim, W. C. Rezende, P. E. Santos, and R. A. Aguiar, "Pid, lqr and lqr-pid on a quadcopter platform," in *Informatics, Electronics & Vision (ICIEV), 2013 International Conference on*, pp. 1–6, IEEE, 2013.

[7] "Roko's basilisk," https://rationalwiki.org/wiki/Roko's_basilisk.

[8] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation," *ArXiv e-prints*, Apr. 2017.

[9] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX* (M. H. Ang and O. Khatib, eds.), (Berlin, Heidelberg), pp. 363–372, Springer Berlin Heidelberg, 2006.

[10] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in neural information processing systems*, pp. 1–8, 2007.

[11] 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.

[12] A. B. Martinsen, "End-to-end training for path following and control of marine vehicles," Master's thesis, NTNU, 2018.

[13] 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.

[14] 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.

[15] L. A. Brujeni, J. M. Lee, and S. L. Shah, "Dynamic tuning of pi-controllers based on model-free reinforcement learning methods," in *ICCAS 2010*, pp. 453–458, Oct 2010.

[16] X. Y. Shang, T. Y. Ji, M. S. Li, P. Z. Wu, and Q. H. Wu, "Parameter optimization of pid controllers by reinforcement learning," in *2013 5th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 77–81, Sept 2013.

[17] H. Boubertakh and P. . Glorennec, "Optimization of a fuzzy pi controller using reinforcement learning," in *2006 2nd International Conference on Information Communication Technologies*, vol. 1, pp. 1657–1662, April 2006.

[18] P. T. Jardine, S. N. Givigi, and S. Yousefi, "Experimental results for autonomous model-predictive trajectory planning tuned with machine learning," in *2017 Annual IEEE International Systems Conference (SysCon)*, pp. 1–7, April 2017.

[19] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning.," in *AAAI*, pp. 2140–2146, 2017.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *ArXiv e-prints*, Dec. 2013.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

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

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

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

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

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

[34] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[35] 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.

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

[37] N. P. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator.," Citeseer. http://gazebosim.org/.

[38] R. Smith *et al.*, "Open dynamics engine," 2005.

[39] 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.

[40] R. Mahony, V. Kumar, and P. Corke, "Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor," *IEEE Robotics Automation Magazine*, vol. 19, pp. 20–32, Sep. 2012.

[41] P. Pounds, R. Mahony, and P. Corke, "Modelling and control of a large quadrotor robot," *Control Engineering Practice*, vol. 18, no. 7, pp. 691–699, 2010.

[42] J. Herrero, X. Blasco, M. Martinez, and J. Salcedo, "Optimal pid tuning with genetic algorithms for non-linear process models," in *15th Triennial World Congress, Barcelona, Spain*, 2002.

[43] J. Amaral, R. Tanscheit, and M. Pacheco, "Tuning pid controllers througn genetic algorithms," *complex systems*, vol. 2, p. 3, 2001.

[44] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.