The Landing of a Quadcopter on Inclined Surfaces using Reinforcement Learning

Rein Fris

Master of Science Thesis

TU Delft

ENGIE

The landing of a quadcopter on inclined surfaces using Reinforcement Learning

by

R. Fris

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Februari 13th at 02:00 PM.

Student number: 4288114
Project duration: October 1, 2018 – December 1, 2019
Thesis committee:

Prof. dr. ir. R. Babuška,	TU Delft, supervisor
Ir. B.F. Ferreira de Brito,	TU Delft
Dr. W. Pan,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Acknowledgements

I would like to thank my supervisors, Tom Kraaij, Peter Borsje, but especially Robert Babuška for helping me out with problems during the development and writing of this thesis. Furthermore, I would like to thank Tim de Bruin and Bruno Ferreira de Brito for their input.

A special thanks to ENGIE for providing the tools, space, and freedom to research a topic that satisfied my interests, and kept believing in my ideas. It allowed me to learn a great amount about Reinforcement Learning and Deep Learning. I am also thankful for having the experience of being at the company, and learning how businesses operate.

Last but not least I would like to thank my family, and especially my girlfriend for all the support during my entire academic journey.

R. Fris
Zaandam, January 31, 2020

Abstract

Deep Reinforcement Learning (DRL) enables us to design controllers for complex tasks with a deep learning approach. It allows us to design controllers that are otherwise cumbersome to design with conventional control methodologies. Often, an objective for RL is binary in nature. However, exploring in environments with sparse rewards is a problem in RL, and finding positive reward becomes exponentially more difficult with increased environment complexity.

For this project, our objective is to design an RL based controller for the landing of a quadcopter on inclined surfaces. Landing is defined as reaching these inclined surfaces with reasonable speed, such that no damage is done to either the quadcopter or the surface to land on upon impact.

We aim to use a binary reward for this task. We use methods to aid exploration in sparse reward environments, namely Hindsight Experience Replay (HER), and non-optimized demonstrations. HER can resample goals from the demonstrator data and the policy rollouts. The resampling of goals is done by considering a portion of the visited states during policy rollouts as the intended goals. The demonstrations are non-optimized in the sense that the demonstrations do not follow the same objective as ours. We consider demonstrations valid if these demonstrations are obtained from arbitrary stable policies.

Our results show that the RL system does generalize to other goals when using HER and demonstrations. The demonstrations are not imitated as were to happen in pure imitation learning. HER, on the other hand, enabled us to receive reward in our complex environment, while also allowing us to experience multiple goals in one policy rollout. We found that lack of HER and demonstrations were not able to overcome the problems of exploration in sparse reward environments. We were able to land our quadcopter on a surface with an inclination of $\frac{\pi}{8} rad$ with $51,200$ Q-function updates approximately 40% of the time. For angles with an inclination of $\frac{\pi}{4} rad$ the success rate is approximately 10%. We suspect that with increased training performance will increase.

We found that landing a quadcopter on inclined surfaces using an RL controller is feasible. Our trajectories clearly showed a swinging motion which in theory should be a valid control strategy for this problem. This swinging motion results in dead spots with the quadcopter being in a state with a minimal translational and rotational velocities under a relatively large angle. Further research is needed to increase the accuracy and robustness of our RL based controller.

List of Symbols

This list of symbols describes used symbols that are used throughout multiple chapters of this document

α	Learning Rate
γ	Discount Factor
μ	Dynamics Parameters
$\pi(\cdot)$	Policy
τ	Target Update Rate
$A(s,a)$	Advantage Function
a	Action
e_p	Position Error
e_v	Velocity Error
g	Goal State
$h_t(s_i, a_i)$	History of States and Actions
$Q(s,a)$	Action-Value Function
r	Reward Value
$R(\cdot)$	Reward Function
s	State
$V(s)$	Value Function

Contents

Objective & Thesis Outline

The goal of this research is to develop a controller for a quadcopter with the objective of landing this quadcopter on inclined surfaces using Reinforcement Learning (RL). Landing is defined as reaching these inclined surfaces with reasonable speed, such that no damage is done to either the quadcopter or the surface to land on upon impact.

The main question is *"What reinforcement learning algorithm is best suitable to train a quadcopter to land on inclined surfaces?"* Due to the vast amount of RL algorithms, we limit ourselves to Actor Critic methods and deep RL. This is due to recent popularity in these techniques. The three sub-questions we ask for this research are:

1. *What algorithms minimize bias in the trained policy?*
2. *What strategies are required for the successful training due to complexity within our objective?*
3. *What methods are required to obtain a robust RL trained controller which successfully performs outside of simulation?*

This thesis consists of 9 chapters. The chapters are structured as follows:

- Chapter 1 introduces our research objective and gives a mathematical description of our quadcopter environment.
- Chapter 2.1 provides background information about the reinforcement learning problem and its associated mathematical principals. The general mechanics and benefits of Actor Critic methods are introduced as last.
- Chapter 2.2 describes the notion of deep learning and deep neural networks. This chapter introduces the mathematical foundations of deep learning and different architectures of deep neural networks.
- Chapter 2.3 combines the methods of Chapters 2.1 and 2.2 and introduces the technique of deep reinforcement learning through recently developed algorithms.
- Chapter 3 proposes solutions from literature to train robust reinforcement learning policies that can transfer successfully to the real world.
- Chapter 4 provides solutions for the problem of exploration in complex sparse reward environments.
- Chapter 5 is the chapter where the findings of the previous chapters are implemented. We translate the findings of the literature end to apply it to our control problem.
- Chapter 6 evaluates the performance of different algorithms applied to our control problem.
- Chapter 7 shows our conclusions based on the performance of the implemented techniques. The chapter gives directions on how to continue with research using other reinforcement learning algorithms.

1

Introduction

Unmanned aerial vehicles (UAVs) are becoming more present in society. AUVs are of interest for operations high above ground, which normally is a dangerous task when operated by humans. Thnks to their capability to reach high heights, its small size, ease of operation and relatively low cost, quadcopters (also called quadrotors or drones) are especially of interest to perform these dangerous tasks high above ground. However, to perform operations with drones near to other objects, manual control is unfeasible due to catastrophic human error, and a computer-controlled quadcopter is a more feasible solution. This can be done in multiple ways.

In cooperation with ENGIE, I investigated how this control problem can be solved through *Reinforcement Learning* (RL). A controller is designed for quadcopters using an autonomously learned control strategy, which trains based upon a certain objective. In our case, this objective is the landing of a quadcopter on inclined surfaces using a Reinforcement Learning approach. Landing is defined as reaching these inclined surfaces with reasonable speed, such that no damage is done to either the quadcopter or the surface to land on upon impact. The aim of this research is to safely land a camera on top of a surveillance camera.

We use Deep Neural Networks (DNN) as function approximators for solving this RL problem. This promising technique, known as *Deep Reinforcement Learning* (DRL), has shown impressive results in controlling complex robotic systems. *Actor Critic methods* (ACs) have been proven to perform well on these tasks. In ACs, two networks are trained. One network performs the actions in the environment, which is called the actor network. The critic network functions evaluates the quality of the actions taken by the actor netowrk. This construction allows for training the appropriate actions, which yield the optimal performance. When the system is in operation, and not being trained, only the actor-network performs the actions, and the value network is not in operation.

Figure 1.1: An illustration of our objective of safely landing a camera on top of a surveillance camera.

This chapter will introduce the quadcopter and will discuss the challenges of applying RL to real systems, and what has been done to solve this and what the objective of this Thesis is.

1.1. Quadcopter

This research aims to develop a controller design procedure for complex non-conventional control tasks by incorporating Reinforcement Learning (RL) controllers into conventional controllers, where RL should enhance the control of a quadcopter which allows for complex landing maneuvers on inclined surfaces. In Figure 1.1 this concept is visualized. This means that we aim to perform actions with quadcopters that lie outside the "stable" regions (the stable regions being hovering and steering under an angle, for example). This technology is developed for quadcopter applications that aim to operate the quadcopter in close proximity to other objects, performing challenging tasks. We propose an RL-controller structure that aims to leverage stable drone flight using conventional controllers, e.g. non-linear PD controllers [35]. The RL agent seeks to find a symbiotic collaboration between the conventional onboard controller and the benefits of the trained policy of the RL agent. This method allows for relatively quick development of specific control tasks design because we can leverage the stability of the conventional controller and we can leverage the ability of an RL controller to solve complex tasks.

The following sections will describe the system, how its dynamics are modeled and how it can be controlled via a non-linear controller. All information from the following sections are from [18] for the system model and quaternion math on quadcopters and [35] for the system model and non-linear control of the quadcopter.

1.1.1. System Description

The quadcopter is a UAV that is propelled by four rotors. The state of a quadcopter is described by

$$s = \left[x_{global}, y_{global}, z_{global}, q_0, q_1, q_2, q_3, \dot{x}_{global}, \dot{y}_{global}, \dot{z}_{global}, w^x_{local}, w^y_{local}, w^z_{local}\right]^T$$

where x_{global}, y_{global} and z_{global} are the position coordinates in the global reference frame of the quadcopter, q_0, q_1, q_2, q_3 is the rotation of the quadcopter expressed in unit quaternion notation, $\dot{x}_{global}, \dot{y}_{global}$ and $\dot{z}_{global}$ are the time derivatives of the position coordinates and w^x_{body}, w^y_{body} and w^z_{body} are the rotational velocities expressed in the body frame coordinates which are the derivatives of the roll, pitch and yaw angles.

The low-level control input to the quadcopter is

$$a = \left[\omega_1^{des}, \omega_2^{des}, \omega_3^{des}, \omega_4^{des}\right]^T$$

with $\omega_1^{des}, \omega_2^{des}, \omega_3^{des}$ and ω_4^{des} as the desired rotational velocities of the rotors (as illustrated in Figure 1.2). These desired rotational velocities are controlled by the on-board computer of the quadcopter. The control input to the controller might, and likely will, differ from the low-level control input (i.e. the controller can take a variety of high-level control inputs). The motor dynamics are relatively fast compared to the rigid body dynamics and aerodynamics. Thus for the controller development in this work we assume the desired rotational velocities can be instantaneously achieved [42].

1.1.2. System Dynamics

The quadcopter has two pairs of rotors. One pair rotates in the clockwise direction and the other pair rotates in the counterclockwise direction, as shown in Figure 1.2.
The force and moment generated by the quadcopter in the local coordinate frame is modeled as follows:

$$\begin{bmatrix} F_u \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & k_f L & 0 & -k_f L \\ -k_f L & 0 & k_f L & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} {\omega_1}^2 \\ {\omega_2}^2 \\ {\omega_3}^2 \\ {\omega_4}^2 \end{bmatrix} \tag{1.1}$$

where k_f and k_m are constants which map the rotor velocities to resulting forces and moments created by the rotor velocities.

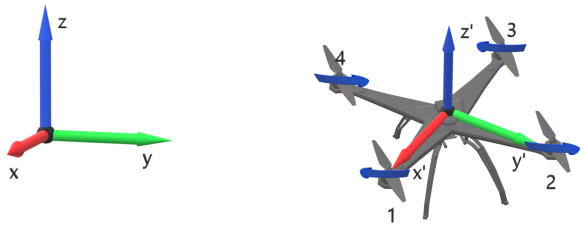

Figure 1.2: The global reference frame g, consisting of x, y, and z axis, and the body-fixed reference frame b, consisting of x', y', and z' axis. Rotors 1 and 3 spin in counterclockwise direction and rotors 2 and 4 spin in the clockwise direction. The thrust of all rotors is directed in the positive z' axis.

The force exerted by the quadcopter in the global frame is described by

$$\vec{F}_{rotor} = R_l^g \begin{bmatrix} 0 \\ 0 \\ F_u \end{bmatrix} \tag{1.2}$$

where R_l^g is the rotation matrix from the local coordinate frame to the global coordinate frame. Due to the design of a regular quadcopter without tilting rotors, the force exerted by the motor is always in the positive z direction of the local coordinate frame. The gravitational force in the global frame is described by

$$F_g = m \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} \tag{1.3}$$

where m is the mass of the quadcopter and g is the gravitational constant. The drag force in the local coordinate frame is modeled by

$$\vec{F}_{drag,local} = -\frac{1}{2}\rho \begin{bmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{bmatrix} \begin{bmatrix} C_{dx} & 0 & 0 \\ 0 & C_{dy} & 0 \\ 0 & 0 & C_{dz} \end{bmatrix} v_{local} \, \|v_{local}\| \tag{1.4}$$

where ρ is the mass density of the fluid, A_x, A_y and A_z are the areas of the drone in x, y, z directions respectively, C_{dx}, C_{dy} and C_{dz} are the drag coefficients of the drone in x, y, z directions respectively, and $v_{local} = [v_x, v_y, v_z]^T_{local}$. The drag force expressed in the global coordinate frame is

$$\vec{F}_{drag} = R_l^g \vec{F}_{drag,local} \tag{1.5}$$

Adding (1.2), (1.3) and (1.5) gives the total force F_{total} on the system in the global reference frame;

$$\vec{F}_{total} = \vec{F}_{rotor} + \vec{F}_g + \vec{F}_{drag}. \tag{1.6}$$

The rotation of the quadcopter is expressed in unit quaternions, which eliminate the chance of singularities which Euler angles might suffer from, known as ”gimbal lock”. Since we aim to explore randomly in the state space of the quadcopter, it is likely that it would encounter situations where gimbal lock becomes a problem, if we were to choose Euler angles instead of the quaternion rotation notation. A quaternion $\vec{q}$ is a hyper complex number of rank 4;

$$\vec{q} = q_0 + q_1\hat{i} + q_2\hat{j} + q_3\hat{k} \tag{1.7}$$

or in vector notation

$$\vec{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T. \tag{1.8}$$

The rotation matrix from the local coordinate frame to the global coordinate frame is as follows:

$$R_l^g = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}. \tag{1.9}$$

When modeling the quadcopter for simulation purposes we need to create the equations of motions. This differential equation should be a function of

$$\begin{bmatrix} \dot{\vec{p}} \\ \dot{\vec{q}} \\ \dot{\vec{v}} \\ \dot{\vec{\omega}} \end{bmatrix} = f\left(\vec{p}, \vec{q}, \vec{v}, \vec{\omega}, \vec{\omega}_{rotor}\right) \tag{1.10}$$

where $\vec{p}$ is the position vector in the global coordinate frame, $\vec{q}$ is the orientation expressed in unit quaternions, $\vec{v}$ is the linear velocity vector expressed in global coordinates, $\vec{\omega}$ is the angular velocity vector expressed around its local coordinate axes and $\vec{\omega}_{rotor}$ is the angular velocity vector of the rotors, which is controlled by the controller.

We can formulate the equations of motion using the Newton-Euler equations (1.11) to describe the translation and rotation dynamics of a rigid body.

$$\begin{bmatrix} F \\ \tau \end{bmatrix} = \begin{bmatrix} m & 0 \\ 0 & I_{cm} \end{bmatrix} \begin{bmatrix} a_{cm} \\ \dot{\omega} \end{bmatrix} + \begin{bmatrix} 0 \\ \omega \times (I_{cm} \cdot \omega) \end{bmatrix} \tag{1.11}$$

F Is the force experienced by the system as described by (1.6) and τ the torque exerted by the system from (1.1).

The time derivative of a quaternion in a local reference frame is described by

$$\dot{\vec{q}} = \frac{1}{2} \begin{bmatrix} 0 \\ \omega \end{bmatrix} \vec{q} \tag{1.12}$$

where ω is the rotational velocity vector around the local reference frame.

The translational and rotational dynamics of the system can be described by (1.11) and (1.12) which result in;

$$\begin{cases} a_{cm} = m^{-1} F_{total} \\ \dot{\vec{q}} = \frac{1}{2} \begin{bmatrix} 0 \\ \omega \end{bmatrix} \vec{q} \\ \dot{\omega} = I_{cm}^{-1} \cdot \tau - I_{cm}^{-1} [\omega \times (I_{cm} \cdot \omega)] \end{cases} . \tag{1.13}$$

1.1.3. Non-linear Control

When controlling a quadcopter that should allow for large rotations, a non-linear controller is needed, instead of a controller based on a linearized model. The error metrics on position and velocity are defined as

$$\vec{e}_p = \vec{r} - \vec{r}_{des} \tag{1.14}$$

and

$$\vec{e}_v = \dot{\vec{r}} - \dot{\vec{r}}_{des} \tag{1.15}$$

respectively. The vector describing the desired total force applied to the quadcopter is computed using

$$\vec{F}_{des} = -K_p \vec{e}_p - K_v \vec{e}_v + \vec{F}_g + \vec{F}_{drag} + m\ddot{\vec{r}}_{des}\,, \tag{1.16}$$

where K_p and K_v are positive definite gain matrices. With this force vector we can compute the desired F_u using

$$F_u = \vec{F}_{des} \cdot z_b. \tag{1.17}$$

The body frame z-axis z_b of the quadcopter should align with the desired force vector of the drone, since the quadcopter can only produce a force in this direction. With this information we can specify the desired rotation matrix R_{des}. The error metric for rotation can be defined as

$$\vec{e}_r = \frac{1}{2} \left(R_{des}^T R - R^T R_{des}\right)^\vee \tag{1.18}$$

where ∨ transforms a skew-symmetric matrix back into a vector.

The angular velocity error is the difference between the actual and desired angular velocity in body frame coordinates:

$$\vec{e}_{\omega} = \omega_l^g - \omega_{l\ des}^g \tag{1.19}$$

where ω_l^g and $\omega_{l\ des}^g$ are the angular velocity and desired angular velocity respectively of the quadcopter in the body frame. $\omega_{l\ des}^g$ Can be computed using the discrete time derivative of R_{des}. Now the torques of the axes on the quadcopter are defined as

$$\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = -K_r \vec{e}_r - K_{\omega} \vec{e}_{\omega} \tag{1.20}$$

where K_r and K_{ω} are positive definite gain matrices. We can now compute the rotor velocities by substituting (1.17) and (1.20) into (1.1) and solving for ω_i. In practice, this is done by inverting the linearization of (1.1) about $\omega_i = \sqrt{\frac{F_u}{4k_f}}$ [35].

1.1.4. RL Controller Design

We propose to use an RL controller where the agents learn to control high-level inputs to a conventional controller from [35]. For the control inputs used by the RL controller, we propose to use a part of) the error metric e_p (1.14). The PD controller regulates the low-level motor actions. This allows the RL agent to exploit properties of stability of the conventional controller. This method could be considered as an automatic reference trajectory generator, which is trained to plan the optimal reference signal at every time step for a given controller.

1.2. Summary

This chapter introduced the basics of the quadcopter in mathematical formulations, and what our objective is of the research. The state-space of the system has been described as well as its low-level control input to the motors of the rotors. Furthermore, a mathematical description of the forces on the quadcopter has been given, and we specified its reference frame with respect to a global reference frame. Lastly, a non-linear PD controller has been formulated and we proposed how we implemented an RL controller with this non-linear PD controller symbiotically.

2

Deep Reinforcement Learning Background

This chapter will give a theoretical background of *Deep Reinforcement Learning* (DRL) methods. First, we will explain what Actor Critic (AC) Methods are and why they are recommended to be used in robotics and it will also introduce the *Reinforcement Learning* (RL) framework. Then we will give a brief overview of what neural networks are and we give a selection of neural network architectures. Lastly, we combine neural networks and RL for DRL. We will discuss the most notable DRL algorithms, namely; Deep Q-Learning (DQN), Deep Deterministic policy gradient (DDPG) and its extensions, Proximal Policy Optimization (PPO), and Soft Actor Critic (SAC). But we start by discussing the possible side effects of sparse and dense reward functions.

2.1. Actor Critic Methods

AC methods are a subset of RL algorithms [53]. An AC algorithm is a variant of the policy gradient methods, which uses (an estimation of) a value function for bootstrapping. This method introduces bias, and lower variance which increases learning rates.

2.1.1. Reinforcement Learning

RL is a machine learning approach, which aims to maximize the notion of cumulative reward. By training which actions in an environment an agent should take to maximize this reward, a policy (often called controller in classical control theory) is found. In supervised learning, a policy is trained through data that is known beforehand and the performance of the trained policy can be compared to its known reference data, whilst RL generates data itself through experimentation and has to train itself on this data.

Markov Decision Process

The problem of RL can be described as a Markov Decision Process (MDP). An MDP is a 6-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \mathbb{T}, \gamma >$ with respectively; state, action, reward, environment transition dynamics, terminal indicator and discount factor. An MDP consists of a set of states S, a set of actions A, let $s \in \mathbb{R}^S$ be the state of the agent in the MDP, and let $a \in \mathbb{R}^A$ be the action. Let the environment transition dynamics be transitioning to a next state s', from state s performing action a be;

$$P(s'|s,a) = \mathbb{P}\{S_{t+1} = s'|S_t = s, A_t = a\} \tag{2.1}$$

and let the reward function be;

$$r = R(s,a) \tag{2.2}$$

in which a scalar reward r is received after performing action a in state s [53].

We consider episodic tasks, where at termination at the episode is indicated with the boolean terminal indicator $\mathbb{T}$, which is 1 at a termination state and 0 otherwise.

Policy

The policy is the decision-making mechanism of the agent, in which the agent maps states to actions. An agent using a deterministic policy, $a = \pi(s)$, will execute an action deterministically, i.e. it will always execute the same actions in particular states. Stochastic policies, $\pi(a|s) = \mathbb{P}[a_t = a|s_t = s]$, will pick an action a according to a probability distribution given a state.

Value Functions and Bellman Equations

In RL, an agent seeks to maximize the cumulative discounted reward called the *discounted return*, which is a measure of long-time performance. This return is defined as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.3}$$

Where $0 \leq \gamma < 1$ is the discount factor, which weights the value of future rewards [53]. To evaluate the quality of state action pairs, the action value, and value functions (Equations 2.4 ,2.9 and 2.5) can be used.

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s, a \right] \tag{2.4}$$

$$V(s)_\pi = \max_a Q_\pi(s,a) \tag{2.5}$$

The Bellman equations expresses the optimal value function for a optimal policy [53]. When a policy π takes actions such that the discounted return is maximized.

The optimal action value function can be defined as;

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{2.6}$$

and the optimal value function can be defined as

$$V^*(s) = \max_a Q^*(s,a) \tag{2.7}$$

Expanding Equation 2.7 gives;

$$V^*(s) = \max_a \mathbb{E}_{\pi*} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s, a \right]$$

$$V^*(s) = \max_a \mathbb{E}_{\pi*} \left[R(s,a)_{\pi*} + \gamma \sum_{s'} P_{\pi*}(s'|s,a) V^*(s') \right] \tag{2.8}$$

$$Q^*(s,a) = \max_a \mathbb{E}_{\pi*} \left[R(s,a)_{\pi*} + \gamma \sum_{s'} P_{\pi*}(s'|s,a) \max_{a'} Q^*(s',a') \right] \tag{2.9}$$

Equations 2.8 and 2.9 are referred to as the bellman optimality equation for $V^*(s)$ and $Q^*(s,a)$, respectively.

Exploration vs Exploitation

The trade-off of exploration and exploitation is an important subject in RL [53]. In RL we seek to find an optimal policy, this involves trying different strategies (exploration), but also training our policy based on the best strategy found yet (exploitation). To have an RL agent that performs well after training, it should have explored enough of the environment to find an optimal policy. It should also train on the optimal policy found, to ensure the quality of this policy.

2.1.2. Actor Critic Background

An AC algorithm is a variant of traditional policy gradient methods, which uses (an estimation of) a value function for bootstrapping [9, 60]. This method allows for continuous actions due to the policy gradient [53] and this method introduces bias, and lower variance which increases learning rates from the bootstrapping of the value function on the policy gradient. The actor in an AC algorithm can be compared with a traditional policy function. The actor is responsible for executing an action a given current state s. The critic can be compared with a traditional value function such as $Q(s,a)$ or $V(s)$, it outputs the estimated value of being in a state based on the rewards it received in the past, using the actions and states presented by the actor. In Algorithm 1 the Vanilla AC approach is demonstrated briefly.

Algorithm 1 Vanilla Actor Critic

while not converged **do**
  take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
  fit $V^\pi_\phi(s)$ using target $r + \gamma V^\pi_\phi(s')$
  evaluate $A^\pi(s,a) = r(s,a) + \gamma V^\pi_\phi(s') - V^\pi_\phi(s)$
  $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \ln \pi_\theta(a_i|s_i) A^\pi(s_i, a_i)$
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end while

Here θ are the parameters for the parameterized policy, and $\nabla_\theta J(\theta)$ is the policy gradient. The learning rate is denoted as α. The policy gradient should be maximized to its global maximum using gradient ascent. Both functions for the actor and the critic can be represented in a deep neural network.

AC algorithms, especially integrated with deep neural networks, have shown promising results in continuous control challenges, with continuous actions [34, 37]. Due to their ability to be applied on continuous control tasks with continuous observations, it is a popular approach for the control of physical systems, in which discretization of the state and action space is often unfavorable [28].

2.2. Deep Neural Networks

Thanks to increasing computing power capabilities and access to big data, deep neural networks (DNNs) have been of major influence on the development of machine learning. Deep neural networks are non-linear function approximators consisting of multiple layers between the input and output layer of an Artificial neural network (ANN). Machine learning with Deep neural networks is often referred to as Deep Learning [30]. Tasks often solved through Deep Learning include image recognition [43], speech recognition [7] and robot control [3, 34, 38]. DNNs are capable of approximating any non-linear function, given pairs of input-output data. Through backpropagation, the DNN is able to adapt its internal parameters that change its output so that more closely resembles its intended output.

2.2.1. Configuration

Broadly speaking, Artificial Neural Networks (ANNs) consist of multiple layers of interconnected artificial neurons. This is roughly modeled after biological brains. An artificial neural network is able to approximate non-linear functions. The working principle of these neural layers will be further elaborated below.

Artificial Neuron

A Deep Neural Network is built up using artificial neurons, which receive a real valued input and give a real valued output which is often a non-linear transformation of the input. These

neurons are interconnected and consist of multiple layers.

$$a^l = f\left(\begin{bmatrix} w^l_{0,0} & w^l_{0,1} & \cdots & w^l_{0,k} \\ w^l_{1,0} & w^l_{1,1} & \cdots & w^l_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ w^l_{j,0} & w^l_{j,1} & \cdots & w^l_{j,k} \end{bmatrix} \begin{bmatrix} a^{l-1}_0 \\ a^{l-1}_1 \\ \vdots \\ a^{l-1}_k \end{bmatrix} + \begin{bmatrix} b^l_0 \\ b^l_1 \\ \vdots \\ b^l_j \end{bmatrix}\right) \tag{2.10}$$

As seen in Equation 2.10, the operation of a single layer in the neural network is a linear matrix operation, in the form of $W\vec{x}+\vec{b}$. To be able to approximate any non-linear function, an activation function is used (unless this activation function itself is linear). which is further elaborated below. The activation function is the non-linear transformation over the input signal. a^{l-1}_k And a^l are the inputs and outputs of the artificial neuron in layers l and $l-1$ respectively, f is the activation function, $w^l_{j,k}$ are the weights, where j and k are the number of outputs and inputs of the artificial neuron respectively and b^l_j are the biases.

Activation Functions

The activation function is essential for the approximation of non-linear systems. Any non-linear non-complex differentiable function could be an activation function. The most popular activation functions are the *sigmoid*, *hyperbolic tangent* (tanh) and the *rectified linear unit* (ReLU), which are shown below.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.11}$$

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.12}$$

$$f(x) = max(0, x) \tag{2.13}$$

Generally, ReLU performs better since it does not suffer from the vanishing gradient problem [20]. The output of the layers are often not ReLU, but *softmax* for classification, *sigmoid* when the outputs need to be bounded between 0 and 1, and *Tanh* when the output is bounded between -1 and 1. The function can be discarded when the output is unbounded.

2.2.2. Backpropagation

Backpropagation is the process of calculating the gradients in for each neuron of a neural network to be able to adjust its weights and biases using gradient descent methods in order to train the model. *Stochastic gradient descent* (SGD) updates the parameters in the direction which minimizes a loss function.

$$\theta \rightarrow \theta - \alpha \nabla_\theta L(\theta) \tag{2.14}$$

where $L(\theta)$ is a differentiable loss function, which is aimed to be minimized.

ADAM [27] is an extension on SGD which combines the advantages of two recently popular optimization methods: the ability of AdaGrad [14] to deal with sparse gradients, and the ability of RMSProp [55] to deal with non-stationary objectives. ADAM varies its learning rates depending on the data distribution from its training data. ADAM is a popular optimizer [44] and accessible in Machine Learning libraries such as Tensorflow.

2.2.3. Network Architectures

Within Deep Neural networks, there are broadly three main types of architectures, namely the *Feed-Forward Neural Network*, the *Convolutional Neural Network* and the *Recurrent neural network*, which will be further elaborated below.

Feed-Forward Neural Networks

The Feed-Forward Neural Network (FNN) is the most basic and most common DNN structure. It is the basis of all neural networks. An FNN consist of more than two layers, which are called the input layer, hidden layers, and output layer. The input layer receives information, and the output layer computes the output of the neural network. The hidden layers manipulate

the data from previous layers (which could be from an input layer or other hidden layers). In FNNs, the input layer receives a 1-dimensional input for each input node, this information flows towards the output layers in one direction.

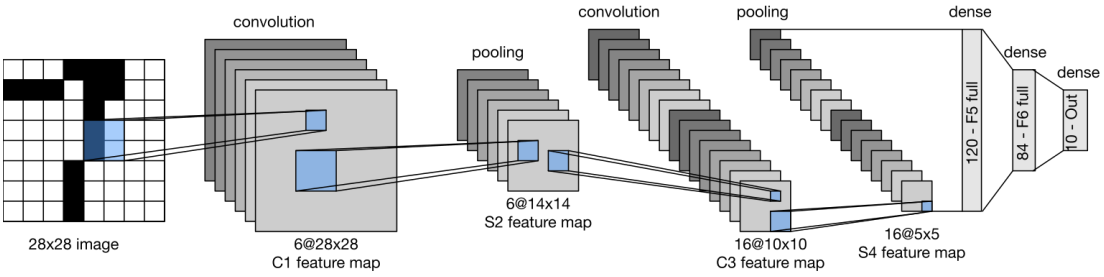

Figure 2.1: LeNet5 [29], an example of a Convolutional Neural Network. The image is processed by 2 convolutional layers and 2 pooling layers before entering the dense layers, commonly found in regular Feed-Forward Neural Networks

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are often used when handling non-scalar data, such as images or waveforms. In the input layers and, depending on the architecture, an arbitrary number of following layers, the CNN manipulates signals with convolutional operations which pass through an activation function and pooling layers. Convolutional layers apply a convolution to the input signal. These convolutional layers are trained to detect features from the data it is trained on. Pooling layers reduce the dimensionality of the signal. This modified signal then proceeds towards the following layers. A popular example is *max pooling*, which returns the max values of the pooling sections. Another example is *average pooling*, which outputs the mean value of the pooling sections. When the signals pass through the convolutional layers and the pooling layers, until the data is reduced to a set of features, it often passes through a regular FNN. The LeNet5 architecture [29] is given as an example for this, shown in Figure 2.1.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) can work with sequential data, thanks to their ability to use the internal state to process sequential inputs. This is often used in text or speech recognition or in writing, where an order of occurrence is of importance to predict the next values. A naive implementation of RNNs often experiences the problem of vanishing and exploding gradient, making it harder to train such networks. Since the layers and sequential steps of RNNs relate to each other through multiplication, it is very probable that the gradients will explode or vanish, due to backpropagation. The *Long Short Term Memory* [24] (LSTM) structure gives a solution to this problem.

LSTMs maintain a more constant error while backpropagating through time than naive implementations of RNNs. The gate structure of the common LSTM cells consist of an *input gate*, a *forget gate* and an *output gate*. This cell ”remembers” information over a given time interval and the three gates regulate the flow of information into and out of the cell. These gates are all trained individually, based on sequential data. This LSTM is connected to more LSTMs to create an LSTM network. Due to its efficiency, LSTMs, or close variants of it, are the standard for most neural networks working on sequential data.

2.3. Deep Reinforcement Learning

In *Deep Reinforcement Learning* (DRL), conventional *Reinforcement Learning* is combined with function approximators using *Deep Neural Networks*. It has gained lots of interest in the last couple of years, thanks to its performance on different tasks such as Atari Games [36], and high dimensional continuous control problems [34].

2.3.1. Sparse Versus Dense Rewards

As discussed in Section 2.1, the training of an RL agent is based on rewards. The agent needs to experience rewards during exploration often enough to be able to train successfully. For environments with sparse rewards, obtaining rewards during exploration becomes less likely when environments become more complex. When unequal values of rewards are obtained with very low probability during exploration, there is a risk that the RL algorithm is not able to train a successful policy. This is when there have not been enough experiences for the agent to learn from the rewards, i.e. the rewards were all equal in value. Dense rewards, however, do not suffer from this problem. When designing a reward function that outputs dense rewards, often these rewards are based on metrics that indicate the quality of states and actions based on a given objective. For our quadcopter environment, we could design a reward function that penalizes differences between the state of the quadcopter and its specified goal, e.g. the distance between the quadcopter and the goal. Here the reward would suggest that moving away from the goal is always negative, but our findings in Chapter 6 suggest that this is not the case for every state. These rewards directly map the quality of states and actions every simulation timestep. For every simulation timestep, there is information available, and this presence of unequal rewards allows for easier training of a policy. A large problem that occurs when dealing with dense rewards, is that a sub-optimal policy is trained. Dense rewards are often carefully designed to aid the learning process towards a desired policy. A policy that performs optimally is favored over a non-optimal policy based on a dense reward function. By implementing a dense reward function, we limit the capabilities of the RL algorithm to find an optimal policy. In this study we use a sparse reward, to avoid sub-optimality of the trained policy due to bias in the policy that might occur due to a dense reward function. How we solve the problems of sparse reward functions will be discussed in Chapter 4.

2.3.2. Deep Q-Learning

Q-learning is an off-policy algorithm for estimating Q-values. Q-values are learned iteratively using the Q-value update

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left(r_t + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right) \tag{2.15}$$

where $Q\left(s_t, a_t\right)$ is the Q-value at state s_t with action a_t, α is the learning rate, γ is the discount factor, and r_t is the reward at time t.

In Deep Q-learning [36] the Q-values are approximated using deep neural networks. The Q-values are parameterized with parameters θ and denoted by $Q(s, a|\theta)$. The network is trained by minimizing the following loss function at every iteration;

$$L\left(\theta\right) = \mathbb{E}_{\left(s_t, a_t, r_t, s_{t+1}\right) \sim U(D)}\left[\left(\underbrace{r_t + \gamma \max_a Q\left(s_{t+1}, a|\theta'\right)}_{target} - Q(s_t, a_t|\theta)\right)^2\right]. \tag{2.16}$$

θ are the parameters of the Q-network and θ' are the target Q-network parameters. θ' are not updated at every iteration unlike θ, but they are either updated slowly towards θ [34] or θ' are copied from θ periodically [36].

The loss function is similar to supervised learning, in which we often minimize the squared

error loss. Here the error between the target from (2.16) and the Q-function is minimized using the squared error loss.

Networks are unable to be trained on temporally correlated data due to overfitting to this data, which will not generalize well to the entire system. A stable version of Deep Q-learning has been accomplished in [36]. Essential for this stability is the use of *experience replay*. Experience replay is the sampling of data, which is stored in a so called *replay buffer*, for training. In a replay buffer the agents' experiences (s_t, a_t, r_t, s_{t+1}) at each time step are stored. This data is sampled in a random manner, this random sampling of data breaks temporal correlation between the data.

The use of DQN is impractical for tasks with continuous actions spaces. DQN relies on a finding the action that maximizes the action-value function. This would require an iterative optimization process at every step, which is too computationally expensive. Deep Deterministic Policy Gradient provides a solution for agents with continuous actions.

2.3.3. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an AC method which allows for control in continuous space and action spaces [34]. It is an off-policy algorithm which makes use of deterministic policies. Determinisitc policies will always return the same action per state, whereas stochastic policies will sample an action according to a ditribution over actions per state. AC networks contain two networks (in the case of deep learning), namely, one policy network (actor) and one value network (critic).

The critic loss function looks as follows:

$$L(\theta) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[\left(y_t - Q(s_t, a_t|\theta)\right)^2\right] \tag{2.17}$$

where

$$y_t = r_t + \gamma Q\left(s_{t+1}, \pi(s_{t+1}|\phi')|\theta'\right) \tag{2.18}$$

where γ is the discount factor determining the agent's horizon, θ and ϕ are the parameters of the Q-network and the policy network $\pi(s_t)$ respectively and θ' and ϕ' are the network parameters used to compute the target. The target networks are slowly updated with τ as described in Algorithm 2. The policy is updated by using the deterministic policy gradient theorem [52] given by

$$\nabla_\phi J = \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta)|_{s=s_t, a=\pi(s_t)} \nabla_\phi \pi(s|\phi)|_{s=s_t}\right] \tag{2.19}$$

Recurrent Deterministic Policy Gradient

Recurrent Deterministic Policy Gradient (RDPG) [23] is a variation on DDPG, which allows the agent to train on recurrent data, i.e. sequences of input data. RDPG trains a recurrent action-value function $Q\left(s_t, a_t, \eta_t\right)$ where $\eta_t = \eta(h_t)$ is the value function's internal state. It trains a recurrent policy $\pi(s_t, \zeta_t)$ where $\zeta_t = \zeta(h_t)$ is the internal state of the policy. To be able to train on recurrent data, recurrent neural networks should be applied. A feed-forward network fails tasks involved in real-world robotic tasks, whereas a recurrent neural network can use its past observations to analyze the dynamics of the world and adjust its behavior accordingly. The internal states η_t and ζ_t are handled within the RNN structures. The internal states for the policy is essential to derive the underlying dynamics of the system[1]. RNNs are further elaborated in Chapter 3.2.

[1] It is unclear from the literature where the inclusion of recurrent data in the value function network is more beneficial than not having it in the value network. In future research it might be interesting to investigate the benefits or consequences when discarding it for dynamics randomization.

Algorithm 2 DDPG

Initialize critic network Q_θ, and actor network π_ϕ with random parameters θ, ϕ
Initialize target networks $\theta' \leftarrow \theta, \phi' \leftarrow \phi$
Initialize empty replay buffer R
for $t = 1$ **to** T **do**
  Select action with exploration noise $a_t \sim \pi_\phi(s_t) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r_t and new state s_{t+1}
  Store transition tuple (s_t, a_t, r_t, s_{t+1}) in R

  Sample mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
  $a \leftarrow \pi_{\phi'}(s_{i+1})$
  $y \leftarrow r_i + \gamma Q_{\theta_j'}(s_{i+1}, a)$
  minimize using SGD $L = N^{-1} \sum (y - Q_{\theta_i}(s_i, a_i))^2$
  Update ϕ by the deterministic policy gradient:
  $\nabla_\phi J = N^{-1} \sum \nabla_a Q_\theta(s_i, a_i)|_{a_i = \pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i)$
  Update target networks:
  $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$
  $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
end for

2.3.4. Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed Deep Deterministic policy gradient [19] (TD3) is an improvement upon DDPG. This variant on the DDPG algorithm is more sample efficient, increases stability and overall performance.

The three components introduced upon DDPG are first; the Clipped Double Q-learning, inspired by Double DQN [57]. This method allows for the underestimation of the actions in the Q-function. This may still be suboptimal, but it is preferred over the overestimation of the actions since the overestimation may develop into a more significant bias over many updates. Besides, an inaccurate value estimate may lead to poor policy updates. Suboptimal actions might be highly rated by the suboptimal critic, reinforcing the suboptimal action in the next policy update, which in turn will become suboptimal. The value of underestimated actions will not be explicitly propagated through the policy update, which allows it to yield to an optimal policy. This method also incentives safer policy updates with stable learning targets, due to a preference of states with low-variance value estimates.

The second improvement is that policy updates occur after the error in the critic network is minimized, by delaying the policy updates until the value error is as small as possible. The use of a value estimate with low variance should result in higher quality policy updates. Due to this more efficient approach, the performance improves while using fewer policy updates.

Lastly, the introduction of a regularization strategy for deep value learning, called *target policy smoothing*. This method should reduce overfitting towards narrow peaks in the value estimate. By smoothing out the Q-function over similar actions, by sampling actions from the target policy with clipped Gaussian noise, the aforementioned sharp peaks in the value estimates are reduced. This will generalize the policy, making the policy more robust.

2.3.5. Other Algorithms

This section describes; Proximal Policy Optimization (PPO) and Soft Actor Critic (SAC). These are two state-of-the-art algorithms for RL, with stochastic policies.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy policy gradient algorithm using stochastic policies [51]. It uses a trust region for its policy update steps. This means that the policy updates during training are not too large, which would otherwise result in a significant

Algorithm 3 RDPG

Initialize weights for actor and critic θ and ϕ.
Initialize weights for target networks θ' and ϕ'.
for episodes $= 1, M$ **do**
  for $t = 0, T-1$ **do**
    Compute memories ζ_t and η_t
    $\hat{a}_{t+1} \leftarrow \pi_\theta(s_{t+1}, \zeta_{t+1})$
    $\hat{a}_t \leftarrow \pi_\theta(s_t, \zeta_t)$
    $y_t \leftarrow r_t + \gamma Q_\phi(s_{t+1}, \hat{a}_{t+1}, \eta_{t+1})$
    $\Delta q_t \leftarrow y_t - Q_\phi(s_t, a_t, \eta_t)$
  end for
  Update actor and critic network using BPTT
  $\nabla_\phi = \frac{1}{T}\sum_T \Delta q_t \frac{\partial Q_\phi(s_t, a_t, \eta_t)}{\partial \phi}$
  $\nabla_\theta = \frac{1}{T}\sum_T \Delta q_t \frac{\partial Q_\phi(s_t, \hat{a}_t, \eta_t)}{\partial a} \frac{\partial \hat{a}_t}{\partial \theta}$
  update actor and critic with Adam
  Update target networks
  $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
  $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
end for

Algorithm 4 TD3

Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize replay buffer R
for $t = 1$ **to** T **do**
  Select action with exploration noise $a_t \sim \pi_\phi(s_t) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r_t and new state s_{t+1}
  Store transition tuple (s_t, a_t, r_t, s_{t+1}) in R

  Sample mini-batch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
  $\tilde{a} \leftarrow \pi_{\phi'}(s_{i+1}) + \epsilon$, $\epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r_t + \gamma \min_{j=1,2} Q_{\theta_j'}(\pi_{\phi'}(s_{t+1}), \tilde{a})$
  minimize using SGD $L = N^{-1}\sum(y - Q_{\theta_j}(s_i, a_i))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J = N^{-1}\sum \nabla_a Q_\theta(s_i, a_i)|_{a_i=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i)$
    Update target networks:
    $\theta_j' \leftarrow \tau\theta_j + (1-\tau)\theta_j'$
    $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
  end if
end for

performance loss. PPO aims to maximize an objective function using minibatch stochastic gradient descent. The objective function uses a probability ratio $r_\pi = \frac{\pi(a_t|s_t)}{\pi_{old}(a_t|s_t)}$, which is the probability of taking the given action under the current policy π to the probability of taking that same action under the old policy π_{old}. Clipping discourages big changes to the new policy. This can be necessary since a large increase in policy update is only beneficial for special conditions. These changes in the policy are often beneficial for the policy in general. The size of the policy update is limited by ϵ. If both r_π and A are positive or negative[2], the policy update gets restricted by ϵ. The advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ indicates that the quality of an action in a particular state is better or worse than the quality of being

[2]If A is positive the action taken was beneficial, if A is negative, the action was taken was undesirable

in that state.

$$L_t^{CLIP} = \mathbb{E}_t\left[min\left(\frac{\pi(a_t|s_t)}{\pi_{old}(a_t|s_t)}A, clip\left(\frac{\pi(a_t|s_t)}{\pi_{old}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)A\right)\right] \tag{2.20}$$

The PPO objective is augmented with two other objectives when trained on neural networks. The first one,

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{targ}\right)^2 \tag{2.21}$$

is the squared error loss of the regular and the target network of the value function. The second one,

$$S[\pi_\theta](s_t) \tag{2.22}$$

is an entropy bonus, which ensures that the agent will explore in its environment. The combined objective for the actor is

$$L_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t\left[L_t^{CLIP}(\theta) + cS[\pi_\theta](s_t)\right], \tag{2.23}$$

where c is a hyperparameter that regulates the importance of the entropy bonus.

Soft Actor Critic

Soft Actor Critic [21] (SAC) is a sample efficient, off-policy AC algorithm that incorporates an entropy measure of the policy into the reward to encourage exploration. It makes use of entropy maximization, which provides an improvement in exploration and is robust in the presence of model and estimation errors [62]. The algorithm learns three functions namely; the policy π_θ, the soft Q-value function Q_w and a soft state value function V_ϕ, and it maximizes

$$J(\theta) = \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi_\theta}}\left[r_t + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))\right]$$

which augments the objective with the expected entropy of the policy over $\rho_\pi(s_t)$. The temperature parameter α determines the relative importance of the entropy term against the reward.

2.4. summary

The first section formulated the RL framework and continued to apply this information especially on *Actor Critic* (AC) methods. The general notion of RL is to train a policy that seeks to maximize cumulative discounted reward called *discounted return*. These rewards are experienced by an agent in simulation. The agent takes actions in an environment which is formulated as a *Markov Decision Process* (MDP). The quality of states or actions is determined by value functions and action-value functions respectively. AC methods train two functions, namely a policy function known as the *actor*, and an (action) value function known as the *critic*. The critic is used to inform the actor on its performance and update its policy in favor of the information from the critic. These AC methods are well applicable to continuous control tasks.

The second section described what *deep neural networks* are and their workings. Deep neural networks are capable of approximating highly non-linear functions with minimal human intervention. Based on network structures, the deep neural network is able to adapt to multidimensional and/or recurrent data. The extreme generalizability of these deep learning methods makes it an excellent solution to our complex problem.

The last section applied *deep neural networks* as function approximators for RL, resulting in *Deep Reinforcement Learning* (DRL). DRL becomes compelling with increased system complexity. Generally, we seek to find an optimal policy in RL. Sparse rewards are favored over

dense rewards concerning the optimality of the policy, however, the more complex an RL objective becomes, the harder training becomes due to a lack of rewards obtained. We described different AC based DRL algorithms. We showed a family of off-policy DRL algorithms, namely DDPG, RDPG and TD3 with deterministic policies, which have been successful in solving various complex tasks. Furthermore, we have shown two other successful algorithms with stochastic policies, namely PPO and SAC.

3

Transfer From Simulation To Real World

RL-control has benefits over traditional control methods. RL controllers are useful when the objective of an agent is known, but it is unclear how this objective should be achieved. RL is applicable on a wide variety of tasks. It often finds solutions for given objectives, provided that the reward function is designed properly. Problems exist which are hard to design with traditional control engineering techniques but solvable through RL techniques, such as systems for manipulating objects using a humaniform robot hand [3]. RL-control, however, suffers from the often mentioned *reality gap* [26, 28], where an agent trained in simulation does not generalize well to the real-world environment. The transfer from simulation to real-world application is often referred to as *Sim-to-Real* in literature. There are examples of robots training on physical systems [33] which eliminates this reality gap, however, this method is not desirable. An agent trained in simulation is cheaper and less time consuming since we can exploit the capabilities of the computer to quickly generate simulation data on which the policy can be trained.

Several methods have been proposed which overcame this reality gap. One example is a method where the reality gap is solved by learning an additional inverse dynamics model [12]. Such methods, however, do not generalize well to changing dynamics (e.g. wear on robot joints) and assume there is data available from a physical system to train on.

RL-control on robots where the transfer from simulation to the real world was successful has been accomplished [3, 4, 40, 54]. This has been made possible mainly by two additions to the conventional RL training algorithms. These additions are; randomization, and the use of RNNs. How randomization and the implementation of RNNs for the sim-to-real transfer work, and why they work, will be explained in this chapter.

3.1. Randomization

Randomization of the simulation physics and domain is useful when a deployed policy should generalize to the dynamics of the real world [40, 45]. Due to discrepancies in the simulation dynamics and real-world dynamics, RL policies trained in simulation often fail to operate successfully in the real world. This occurs since the behavior of the agent in simulation tends to overfit to the simulation dynamics when the agent is naively trained in a single environment [5]. An example can be given from [40], where a robot manipulator could not account for the friction of a puck on a table when only trained in simulation. This policy did not work well in the real world scenario and thus was not able to push the puck over the table successfully. However, the same robot trained with dynamics randomization was able to push this puck. Training with different values for the friction between the puck and table, among with other dynamical parameters such as masses and damping coefficients, resulted in a policy that behaved desirably on the real robot arm.

In randomization, the environment and the dynamics are changed slightly during training episodes randomly. The agent needs to train a policy that works on a variety of conditions. The core idea of this principle is that the agent perceives the real environment, as a new randomized (simulated) environment. Since the agent is trained to adapt to randomization in the environment, it should also be able to adapt to the real environment.

3.1.1. Dynamics Randomization

In Dynamics Randomization, certain parameters of the model are changed randomly during simulation [40, 54]. These could be parameters such as; mass, friction, moments of inertia, observation noise, simulation timestep length, and the gravity constant. These randomly chosen parameters, i.e. the dynamics parameters μ, change the dynamics of the simulated model.

The dynamics parameters are known during training, but we do not have an exact knowledge of these parameters from the real world. For Actor Critic methods, the policy network (actor) is used as the controller when employed in the real world, i.e. the value function network (critic) is only used during training and simulation. Since during training and simulation all parameters are known, we can supply the value function network with this information. The authors of [40] refer to as an *omniscient critic*. This is to reduce the variance of the policy gradients. It allows the value function to provide more meaningful feedback to the policy as well. In Figure B.1 it is visible that our actor network does not include information about the dynamics parameters μ, while our critic network does.

Domain Randomization

Domain Randomization strongly resembles Dynamics Randomization, because it aims to generalize the policy to work in real-world applications. Domain Randomization, however, focuses on randomizing the domain rather than the dynamics, e.g. Domain Randomization randomizes the colors, textures and so on, when using computer-generated images as inputs [45, 56]. Our model does not rely on any image based input, so we do not use the technique of Domain Randomization.

3.1.2. Side Effects of Randomization

RL controllers trained with randomization techniques allow for robust control of systems where models are not accurate. While these techniques allow for stable control of systems with tolerable model errors, this also means that optimality is lost since these controllers need to cater for a variety of system dynamics.

3.2. Closed Loop Control using Recurrent Neural Networks

RNNs can effectively solve the problems for Partially Observable Markov Decision Process (POMDPs) in deep reinforcement learning [22]. POMDPS are MDPs where the agent cannot access the state directly. Problems introduced by POMDPs were solved by maintaining probability distributions over possible states based on observations in the environments and making decisions by combing information about these probability distributions and the observations [53]. They have proven to be essential when a simulation trained policy is deployed on a physical system. RNNs are able to interpret temporal sequences of data, which means that RNNs are capable of memorization. This allows the network to generalize the dynamics (similarly to system identification) on memorized data, and adapt the policy accordingly. This is essential when training networks with dynamics randomization.

3.3. Summary

This chapter examined the possibilities of transferring the policy from simulation to reality successfully without training on the real system. This is accomplished by randomizing features in simulation. The system is trained such that it can adapt to these discrepancies between features in simulation. Eventually, the policy should be robust to changes in the dynamics, hence work in the real world.

4

Improving upon Regular Exploration

In RL, the reward function is used to generate rewards. Reward functions are engineered in a manner such that it describes how an agent should behave. There are multiple strategies for how to engineer such reward functions. These different strategies often yield different outcomes for the trained model. We could train a model using dense reward functions, by giving feedback on the performance from every visited state during exploration. *Shaped reward functions* (a type of dense reward functions) *limit the applicability of RL in the real world. Shaped reward functions are not applicable in situations where we do not know what admissible behavior may look like.* [2] Sparse rewards are favored over dense rewards, due to the risk that the policy adopts less optimal behaviors due to bias in the reward function. [2] however, Training an agent with sparse rewards tends to be harder to solve than training on dense rewards. This is because the agent might never reach the goal position by random exploration, and thus would never receive reward to train on. Ideally, we train the agent on sparse rewards only. Without modifications to the RL algorithm, this might not be possible since the agent never explored the goal objective. This chapter will look into techniques to overcome the problems of exploration using traditional sparse rewards in complex tasks.

4.1. Hindsight Experience Replay

Hindsight Experience Replay [2] (HER) partially solves the problem of insufficient exploration possibilities in sparse reward function settings. In HER policies are trained on multi-goal problems[1], which follows the *Universal Value Function Approximators* [48] (UVFA) approach. This means that the policies and value functions trained take a goal $g \in \mathcal{G}$ as input alongside a state $s \in \mathcal{S}$.

Consider an episode with a state sequence $s_1, \ldots, s_T$ and a goal $g \neq s_1, \ldots, s_T$, and consider a reward function

$$r(s, g) = \begin{cases} r_g, & \text{if } g = s \\ 0, & \text{otherwise} \end{cases}$$

where r_g is the reward of reaching the goal. In regular exploration, the state sequence $s_1, \ldots, s_T$ would, most probably, not receive any reward, and thus would not be able to train on the reward signal properly. HER samples the state sequence $s_1, \ldots, s_T$, and samples a selection of states[2] from this state sequence as a virtual goal g'. This is what we refer to when we talk about the resampling of goals. When goals are resampled, the rewards of arriving in goal g' need to be recalculated, and a policy and value function update can be performed with this new information about g'. For the working principle of HER in pseudo-code format, see Algorithm 5. This method however only is able to train using an off-policy algorithm, since

[1] The objective to solve does not necessarily have to be a multi-goal problem, however, to apply HER techniques it should be modeled as one.

[2] The selection of states depends on the goal sampling strategy, in [2] the final state s_T is used as virtual goal sample. This, however, is not mandatory.

we sample from a policy with another goal state.

HER allows the creation of new training data where the agent reaches goals independently of the exploration strategy. This eliminates the problem of never exploring the goal states. However, depending on what the desired goal conditions are, HER still might not be able to solve this sparse reward problem. When tasks become more complex, the chances of arriving at these goal conditions becomes smaller. An example is given where a possible shortcoming of HER is demonstrated below.

The Shortcomings of HER; an example
A case where HER should have no problem with resampling goals is when, a quadcopter, for example, is only required to reach a position in 2D space independent of other states. An example that would be more difficult to achieve with HER is when a quadcopter should reach a point in space with zero velocity, but the velocity is not part of the reward function. If during exploration, there is no state when this agent has zero velocity, there is no opportunity to resample goals. This is because these goals can only be resampled where the goal condition of zero velocity is met. In short, when a goal condition is unlikely to be met during exploration, there is little opportunity for the resampling of goals. We defined our reward function such that it is able to sample from many states, which is shown in Chapter 6.1.3.

Algorithm 5 Hindsight Experience Replay [2]

Initialize $\mathbb{A}$ (the off-policy RL algorithm) {$\mathbb{A}$ is given}
Initialize replay buffer R
for episode= 1 **to** M **do**
  Sample a goal g and an initial state s_0 {Sampling strategy is given}
  for $t = 0$ **to** $T - 1$ **do**
    Sample an action a_t using the policy from $\mathbb{A}$:
    $a_t \leftarrow \pi_b(s_t|g)$
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0$ **to** $T - 1$ **do**
    $r_t = r(s_t, a_t, g)$ {Reward function is given}
    Store the transition $(s_t|g, a_t, r_t, s_{t+1}|g)$ in R
    Sample a set of additional goals for replay $G = \mathbb{S}(Currenepisode)$
    for $g' \in G$ **do**
      $r'_t = r(s_t, a_t, g')$
      Store the transition $(s_t|g', a_t, r'_t, s_{t+1}|g')$ in R
    end for
  end for
  for $t = 1$ **to** N **do**
    Sample minibatch B from R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

4.2. Learning from Demonstrations using Behavior Cloning

The training of an RL agent can be augmented with demonstration data. Using demonstrations in RL is an active research area [47, 58, 61]. A method that combines RL and demonstrations successfully on complicated multi-step, long-horizon tasks with generalization to varying goals has been published in [38]. This method makes use of four key concepts. The most notable of these concepts is the imitation loss with Q-filter [38].

4.2.1. Demonstration Buffer

Besides a replay buffer R, a second replay buffer R_D is maintained in which demonstration data is stored in the same format as in R. During training, from both R and R_D a minibatch

of transitions are drawn for the update of the actor and critic. This idea has been introduced in [58].

4.2.2. Behavior Cloning Loss

The behavior cloning loss L_{bc} can be used as an extra loss function in the RL framework with demonstrations [38]. This loss aims at minimizing the error between the demonstrator action and the action from the policy in a state presented by the demonstrator. The behavior cloning loss

$$L_{BC} = \sum_{i=1}^{N_D} \|\pi(s_i) - a_i\|^2 \tag{4.1}$$

is a standard loss in imitation learning, which also could be used as an auxiliary loss for RL to significantly improve learning. Where L_{BC} is the behavioural cloning loss, $\pi(s_i)$ is the policy in demonstration state s_i, a_i is the demonstration action.

The gradient applied to the actor parameters θ_π is:

$$\lambda_1 \nabla_{\theta_\pi} J - \lambda_2 \nabla_{\theta_\pi} L_{BC}\,, \tag{4.2}$$

where $\nabla_{\theta_\pi} J$ is the standard policy gradient, which is maximized, while the behavioral cloning loss L_{BC} is minimized. λ_1 and λ_2 are hyperparameters which influence the amount that the gradients of J and L_{BC} influence the policy update.

Q-filter

A problem with behavior cloning in RL is that it might be possible that these demonstrations are sub-optimal. This is solved through a *Q-filter*. Since the Q-value is an indicator of performance, it can be used to determine whether the demonstration action a_i is better than the action taken by the actor $\pi(s_i)$. Only if the demonstration action is better than the action from the policy, it is accounted for in the behavior cloning loss:

$$L_{BC} = \sum_{i=1}^{N_D} \|\pi(s_i) - a_i\|^2 \mathbb{1}_{Q(s_i,a_i) > Q(s_i,\pi(s_i))}\,, \tag{4.3}$$

where $Q(s_i, a_i)$ is the action value of the demonstration action a_i in demonstration state s_i and $Q(s_i, \pi(s_i))$ is the action value of policy action $\pi(s_i)$ in demonstration state s_i. The behavioral cloning loss is only taken into account for the demonstrations action that yield a higher action value than the policy action does.

4.2.3. Resets to demonstration states

Some training episodes are started from demonstration episodes using states and goals from R_D. Restarts from within demonstrations expose the agent to higher reward states during training. This is beneficial in problems with sparse rewards in long-horizon tasks.

4.3. Summary

This chapter introduced two methods to aid the agent in exploration. The first method is called Hindsight Experience Replay. This method requires the training objective to be based on multiple goals. Hindsight Experience Replay resamples visited states during a rollout as alternative goal states. Since receiving rewards in sparse reward tasks with regular exploration might be insufficient, Hindsight Experience Replay allows the agent to experience these rewards, by reformulating the goal objective after the trajectory rollout has been performed. This reduces the chance of not finding a solution due to the negative effects of sparse rewards drastically.

To aid exploration, demonstrations can be used as well. In many cases, demonstrations are available or can be generated with relative ease. Demonstration data can be used to give the agent a head start and to obtain a reasonable policy relatively quickly. From these demonstration data, goals can be resampled using Hindsight Experience Replay. Hindsight Experience Replay can resample goals beneficial to our policy.

5

Algorithm Implementations

In this chapter, we will introduce the algorithms that were implemented for the training of the controller and benchmark using the openAI LunarLander[1] [11]. In Section 5.1 we will discuss our own implementation of the off-policy TD3 [19] algorithm. In Section 2.3.5, we will give a description of our on-policy PPO algorithm [51] implementation.

5.1. Algorithm

Our TD3 [19] algorithm includes Hindsight Experience Replay [2] and exploration through demonstrations [38], and it seeks to overcome the reality gap using dynamics randomization and recurrent neural networks [40]. In the following sections, we will discuss why these methods are chosen for this task. The configuration of our Actor Critic networks can be found in Appendix B.1.

5.1.1. Twin Delayed Deep Deterministic Policy Gradient

A detailed description of the TD3 algorithm is given in Chapter 2.3.4. The use of TD3 is essential in our case and it is preferred to regular DDPG. The TD3 algorithm is able to learn more robust policies than DDPG, since TD3 compensates for the deficiencies in the Q-value approximation that DDPG can suffer from.

5.1.2. Sim-to-Real

To develop a controller to be employed in a real-world task, dynamics randomization [40] and LSTMs are used. This combination allows the system to generalize to a range of possible dynamics in the system. Since the controller generalizes over dynamics, the controller should theoretically be able to overcome the problems arising from the mismatch between simulation and real-world dynamics. The critic is modeled as an *omniscient critic* [40]. This means that the critic receives all available information from simulation, while the actor only receives information normally available in reality during training. In dynamics randomization, we have access to the dynamics parameters μ. In reality, these parameters might differ from the simulator dynamics. During training, the dynamics parameters μ are randomly selected for each rollout. In the case of an action-value function, the new value function is modeled $Q(s_t, a_t, \mu)$, where μ are the dynamics parameters that contain information about the dynamics present.

The LSTM units are used to infer the dynamics of the system from past observations [40].

5.1.3. Hindsight Experience Replay

In our simulation setup, we use Hindsight Experience Replay to be able to generalize to different goal positions and orientations. HER allows us to sample a state of the systems

[1] LunarLander is a simulation of a flying mechanism, aimed to land on a moon-like surface. We can perform tests with a custom reward function in this scenario, which we can copy to our quadcopter environment. Due to its flying dynamics, this readily available environment is a good comparison with our quadcopter environment.

as being in a feasible goal state. This significantly increases the chance of finding any goal states during training, since we can sample goal states from states of the system.

Probability of Reaching Goal State

In our objective, the goal state has a relatively high dimensionality. The goal state is described in Chapter 6.1.3.

With truly random exploration alone, these goals are unlikely to be reached frequently enough during exploration to be able to train the system on these goals, due to its high goal dimensionality. If we would use dense rewards (e.g. a reward function which is a function of distance to the goal and the difference in angle with respect to the desired angle), we could incentivize the agent to explore towards the goal. This, however, could lead to a biased policy, which we try to avoid. HER allows for the sampling of goals based on visited states during exploration, which can guarantee that the agent experiences goal states for training.

Generally, we desire that during the landing procedure of a quadcopter the objective is that speed $\dot{p}_g$ and rotational speed ω_g are approximately 0, since we define landing needs to occur at minimal velocity.

However, the frequency that this objective occurs during exploration is too low, and this would not allow for sufficient opportunities to resample goals for HER. To solve this we include the speed $\dot{p}_g$ and rotational speed ω_g into our goal g. This allows for sampling generalized goals where $\dot{p}_g$ and $\omega_g \neq 0$. Although this means that we increase our goal state g with additional goal states that are fixed when deployed on a real system (i.e. $\dot{p}_g$ and $\omega_g \approx 0$ in the goal state g), we are able to resample generalized goals where $\dot{p}_g$ and $\omega_g \neq 0$ which allows us to resample goals frequently. The reward function for our quadcopter environment is defined in Chapter 6.1.3.

5.2. Learning from Demonstrations

Since quadcopters are controlled stably, either human or computer-controlled, we can exploit this stability for learning from demonstrations. Learning from demonstrations can be done from human demonstrations, but since we can use conventional controllers in simulation, we generate off-policy exploratory training data using these controllers in simulation. We do not focus on optimizing our non-RL-controller and imitating its behavior. The demonstration data is used for learning a stable policy of our quadcopter, especially in the early stages of training. This stable policy protects the agent of exploring towards undesirable (i.e. unstable) states. This method increases learning speed in RL, and the RL trained policies can outperform the demonstrator policy, as shown in [38].

This method is different from [61], where the goal is to mimic a conventional MPC controller, with the benefits of having a policy network that can generalize its policy with its on-board sensor data, while trained on the full-state MPC controller using supervised learning.

5.3. Algorithm Implementation

For the design of the RL training algorithm, a sample efficient off-policy training algorithm has been implemented, together with measures to overcome insufficient exploration and reducing errors arising in sim-to-real by randomization in the simulator dynamics. Since we are using recurrent neural networks, dynamics randomization, and HER, we need to account for this in the algorithm and neural network structure. How and why this is done will be shown below.

Firstly, we need to take historic states into account in the actor and critic networks. The actor-network is as follows,

$$a_i = \pi_\phi(s_i, h_T(s_i, a_i)), \tag{5.1}$$

where a_i is the action as calculated by the policy network π_ϕ with inputs s_i at time i and $h_T(s_i, a_i)$ is a function of the historic states of s_i and a_i respectively, with $T > 0$ historical sequential samples. $h_T(s_i, a_i)$ returns a vector of length $T - 1$ sequential historical samples. The function is as follows

$$h_T(s_i, a_i) = [(s_{i-1}, a_{i-1}), (s_{i-2}, a_{i-2}), \dots, (s_{i-T+1}, a_{i-T+1})] \tag{5.2}$$

where the function extracts the states for the recurrent neural network from a buffer. For the TD3 implementation these buffers are R and R_D.

The critic network is as follows,

$$Q_\theta(s_i, a_i, h_T(s_i, a_i)), \tag{5.3}$$

where s_i and a_i are the state and the action at time i and $h_T(s_i, a_i)$ are the historic states of s_i and a_i respectively, with $T > 0$ historical sequential samples.

Secondly, we need to include the dynamics parameters μ into the critic network. The critic network is as follows

$$Q_\theta(s_i|\mu, a_i), \tag{5.4}$$

where s_i and a_i are the state and the action at time i, and $\mu \in M$ are the dynamics parameters. These dynamics parameters are given only to the critic since information about the dynamics of the environment is only available in simulation. The actor-network eventually will be employed in a non-simulated environment, and therefore it will not have access to these dynamics.

Lastly, we need to implement a generalized policy [48]

$$\pi_\phi(s_i|g) \tag{5.5}$$

where the goal $g \in G$ is provided along with state s_i. In simulation, we will sample a random goal at the start of the episodes which does not change during the course of the episode. During training, we will sample goals from states the agent has encountered during exploration, following the *future, episode* and *final* sampling strategies [2] which are further elaborated in Chapter 6.2.1.

We need to include the dynamics parameters μ into the critic network as well. The critic network is defined as follows

$$Q_\theta(s_i|\{g, \mu\}, a_i), \tag{5.6}$$

where s_i and a_i are the state and the action at time i, and $g \in G$ is the goal.

When combining these components for recurrent neural networks, dynamics randomization, and HER, we get for the actor and critic networks

$$\pi_\phi(s_i|g, h_T(s_i|g, a_i)) \tag{5.7}$$

and

$$Q_\theta(s_i|\{g, \mu\}, a_i, h_T(s_i|\{g, \mu\}, a_i)), \tag{5.8}$$

respectively.

5.3.1. Training

During training, we maintain six neural networks and two replay buffers. Two critic networks θ_1 and θ_2, two target networks θ_1' and θ_2' for the critic, one policy network ϕ, and one target network ϕ' for the policy. We maintain a replay buffer R for regular training data, and a demonstration replay buffer R_D for demonstration data. Before sampling from the policy, we obtain demonstration data. We train our network episodically. At the start of the episode,

the initial state generally starts from a random state. But if possible, the initial state is one of the demonstration states, periodically. During the rollout with the original policy and the demonstration rollouts, goal states are sampled using HER. Rollouts are generated by sampling from the policy with parameter space noise [41] (see Appendix B.2), and samples are stored in R. Each episode ends when the maximum sampling steps are exceeded, or when the system is in a termination condition. Data from both R and R_D are sampled randomly for training the critic, and policy (with policy delay) using the TD3 method. Demonstration data from R_D is also used for imitation learning loss with Q-mask. For an in-depth explanation of the mechanics of the TD3 algorithm, please refer to Algorithm 4. For the hyperparameters used during training please refer to Appendix A.1.

5.4. PPO

For the mechanics of the Proximal Policy Optimization method, please refer to Chapter 2.3.5. The desirability of a state-action pair is expressed in the advantage $A(s,a)$, which we calculate using the Generalized Advantage Estimation (GAE) [50]. The generalized advantage estimator for $0 < \lambda < 1$ makes a compromise between bias and variance in the estimation of $A(s,a)$. This is a similar approach as TD(λ).

Since we compare PPO to our TD3 algorithm, we neglected dynamics randomization in PPO for increased computation time. PPO has been successful with dynamics randomization [3].

5.4.1. Training

We maintain two neural networks (see Appendix B.1), namely one policy network and one value network. They have similar structures, but we do not use the current action a_t, since for PPO the value function $V(s)$ is used, and for TD3 the action-value function $Q(s,a)$ is used. We train our network episodically. At the start of the episode, the initial state generally starts from a random state. But if possible, the initial state is one of the demonstration states, periodically. Rollouts are generated by sampling from the stochastic policy. Each episode ends when the maximum sampling steps are exceeded, or when the system is in a termination condition. Minibatches from the batch of rollouts are used to update both the policy and value networks. Our hyperparameters can be found in Appendix A.2.

6

Simulation Setup and Results

This chapter introduces our simulation setup, in which the goal-oriented sparse reward function is defined. We define our goal-oriented quadcopter environment, and how the actions from the policy influence our environment. This chapter also includes the results of our environments. The LunarLander [11] environment is used to test the efficiency of HER and Learning from Demonstrations[1]. Lastly, we show the results of our quadcopter on our algorithm and compare it to our environment on PPO.

6.1. Simulation Setup

In this section, we will discuss how we validated the appropriate reward function structure for our quadcopter problem. We use a popular benchmark in RL, which has a resembles our RL problem to a degree[2] and we modify its reward function such that it resembles our quadcopter objective.

6.1.1. Reward Function

Our reward functions for both environments are similar in nature, that is, both use exclusively sparse reward functions. For both environments, we aim to reach a specified goal. This goal is reached when the agent reaches any given coordinates and orientation at a minimal velocity.

Generally we divide the reward function up in ”sub”-reward functions;

$$R_{sub}(g_k) = \begin{cases} 1 & \text{if } |k_g - k| < \epsilon_k \\ 0 & \text{otherwise} \end{cases}, \quad (6.1)$$

where the total reward is defined by

$$R(s, g) = \prod_{k=x,y,\dots}^{N} R_{sub}\left(k_g, k\right). \quad (6.2)$$

Here k indicates the measure of the desired goal states k_g. For example, if we would specify a goal for an arbitrary agent to reach a position in x, y and z , our goal would consist of three desired goal states $g = [x_g, y_g, z_g]$ where $k = [x, y, z]$. The tolerance for each measure of k is defined by ϵ_k.

[1] The inclusion of such a benchmark was chosen to confirm whether or not our implementations of the RL-algorithm worked.

[2] Among popular benchmarks for RL, the LunarLander benchmark resembles ours the best. While both environments are described similar in their states s, they do differ significantly with regards to the actions a. However, we suspected that if our RL-algorithm worked on the benchmark, it probably should work on our quadcopter environment.

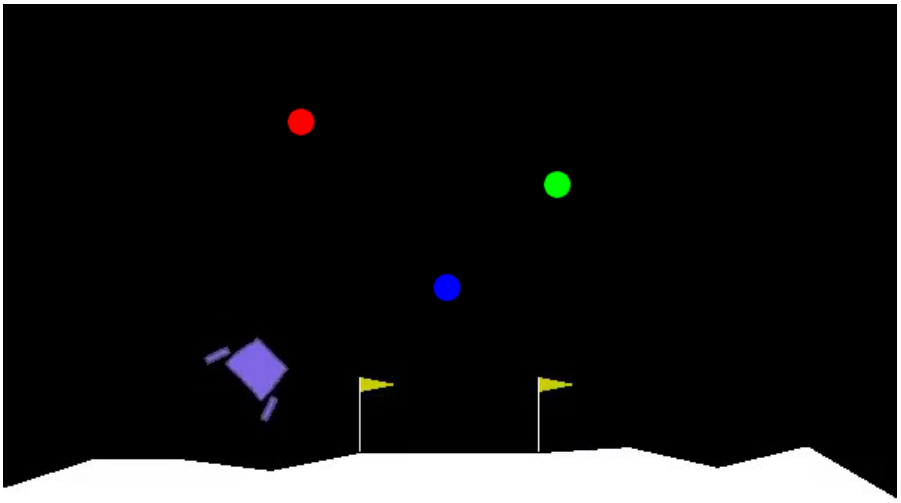

Figure 6.1: This is the LunarLander environment [11], with modified goal positions. The agent controls the spaceship (the polygon in purple). Normally, the goal is specified as landing between the flags on the ground. We position a goal randomly above the ground. The colored dots represent randomly generated goals.

6.1.2. LunarLander Environment

For our modified version of the LunarLander environment[3], shown in Figure 6.1, we aim to approach a specified goal position $p_g = [x_g, y_g]$ in the global coordinate frame, goal velocity $\dot{p}_g = [\dot{x}_g, \dot{y}_g]$ in the global coordinate frame, goal orientation θ_g in the global coordinate frame, and a goal angular velocity ω_g in the local coordinate frame[4]. We specify these criteria into our goal state since we need to be able to generalize towards all possible states when sampling goals with HER. If we would eliminate any of these states from the goal, we limit ourselves in what goals we can sample. For example, if we would eliminate $\dot{p}_g$, the goal velocity would be fixed and thus we would only be able to sample a goal when the velocity is within the tolerance as described by Equation 6.1. This limits the amount of possible resampled goals. This holds for our quadcopter environment as well, which is described below. The LunarLander environment is controlled with 2 actions; the thrusts on the sides for balance, and a thrust on the bottom of the lander for height control. The thrusts for balancing the device cannot occur on both sides simultaneously.

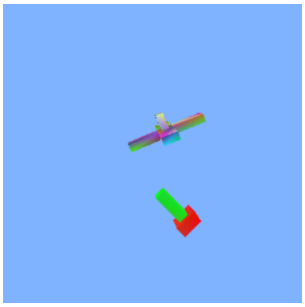

Figure 6.2: Our custom quadcopter environment. The upper object is the agent. The other object is the landing position in the desired landing orientation. The green rod indicates the desired z-axis of our quadcopter for landing.

[3] Originally, the LunarLander environment has a fixed goal position, we deviated from this fixed goal.

[4] goal measures k are $[x, y, \dot{x}, \dot{y}, \theta, \omega]$

6.1.3. Quadcopter Environment

For our quadcopter environment (Figure 6.2), we aim to approach a specified goal position $p_g = [x_g, y_g, z_g]$ in the global coordinate frame, goal velocity $\dot{p}_g = [\dot{x}_g, \dot{y}_g, \dot{z}_g]$ in the global coordinate frame, goal orientation vector $z_g = [z_{x,g}, z_{y,g}, z_{z,g}]$ in the global coordinate frame, and a goal angular velocity $\omega_g = [\dot{\theta}_g, \dot{\phi}_g, \dot{\gamma}_g]$ in the local coordinate frame[5]. The goal state has these many criteria for the same reasons as described in the previous section.

The actions of our policy control the error signal $\vec{e}_p$ from (1.14) in only global y and z position, since we can specify a global reference frame such that the direction of the landing vector lies in the y,z-plane of this global reference frame. This transforms our problem into a 2D type situation, however, we do not restrict the drone to fly in the x direction. We do, however, keep the error between the desired x position ($x = 0$) and the x position of the quadcopter minimal.

Our new error signal e_p is as follows:

$$e_p = [x, a_0, a_1],$$

here x is the x position in the global reference frame of the drone, a_0 and a_1 are the actions of the policy. We modified our error signal e_v (1.15), where we assume that $\dot{r}_{des} = 0$, thus our new error signal is $e_v = \dot{r}$. Our RL algorithm should be able to adapt its policy to overcome possible negative effects of this assumption. By not including e_v for our policy actions, we simplify the agent, and thus simplify training.

We limit our agent to enter a region below the landing position defined by a sphere with a radius of $R = 0.5\,m$. This is an artificial boundary where in reality the obstacle to land on would be present. An illustration of this is given in Figure 6.3. If the agent enters this area or flies 20 meters away from its starting position, then the simulation will terminate and no reward will be given. Doing this allows us to use HER, since HER can not sample goals when the dynamics of the system are dependent on this goal [2]. If we consider a successful landing when there is contact with a surface under safe landing conditions, the contact dynamics would not allow us to sample goals from the trajectory where these criteria have been met. This is not beneficial. Firstly because there is an infinitesimal chance that these landing criteria are satisfied during random exploration. Secondly, it limits us in how much of the state space can be used for the resampling of goals.

Figure 6.3: An illustration of how contact is restricted during simulation. The red sphere represents a forbidden area the quadcopter is not allowed to enter in simulation, and the simulation will be terminated if it does. The camera (the landing platform) is programmed to have contact dynamics with the quadcopter. This simplifies our environment while maintaining restrictions for flying dangerously near an object.

6.2. Results

In this section, we discuss the results of our algorithm on the LunarLander benchmark, and our quadcopter environment. The LunarLander environment is used to evaluate our modified TD3 algorithm. Both HER and Learning from Demonstrations are examined in this environment since training on this environment should be less expensive due to a smaller state space than the 3D quadcopter environment.

6.2.1. HER and Learning from Demonstrations

We conducted experiments for different modes of the sampling of goals for HER. The sampling strategies are, in accordance with [2]; *future* — replay with k random states which come from the same episode as the transition being replayed and were observed after it, *episode* — replay with k random states coming from the same episode as the transition being replayed and *end* - replay with 1 state which comes from the end of the episode. In [2], experiments

[5]goal measures k are $[x, y, z, \dot{x}, \dot{y}, \dot{z}, z_x, z_y, z_z, \dot{\theta}, \dot{\phi}, \dot{\gamma}]$

have shown that the number of replays per transition k should preferably be 4 or higher, but not higher than 8. We adopted $k = 4$ in our experiments. In Figure 6.4 the performance of these techniques is plotted for these sampling strategies.

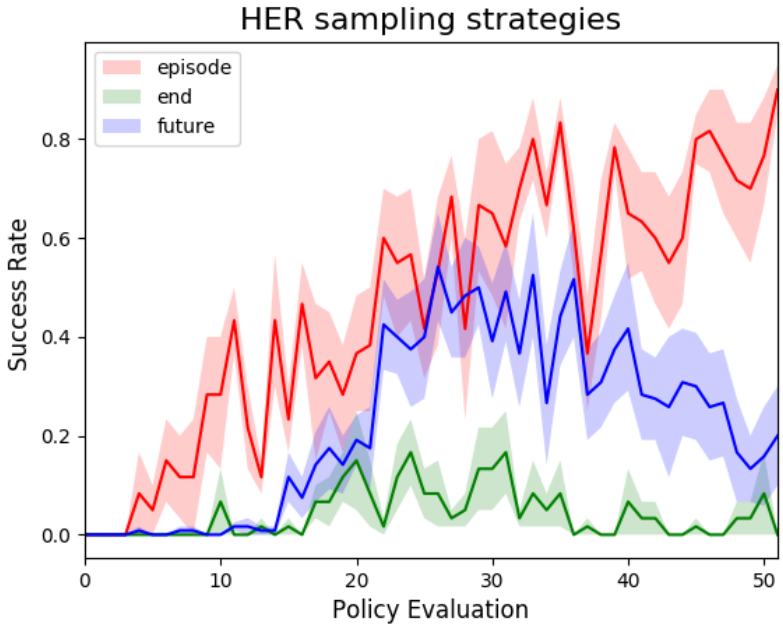

Figure 6.4: Evaluation of HER strategies on the LunarLander environment. For every strategy, we trained a neural network 5 times each with 51,200 Q-function updates. The results are averaged across 5 random seeds and the shaded areas represent one standard deviation. The goal orientation is held at $0\,rad$ during evaluation.

The *episode* strategy performs best on average, the *future* strategy performs second best, and the *end* strategy performs worst. The *future* sampling strategy was found to be the overall best performing sampling method by [2]. Our findings do not match with [2]. Since the LunarLander environment simulates contact dynamics, these dynamics do influence the performance of these sampling strategies. The *end* strategy would most probably sample a goal that is in a crash condition (unsafe landing of the LunarLander) since crashes occur regularly when the policy is ill-defined. The crash terminates the simulation and hence will be the last state, which the *end* sampling strategy uses as the resampled goal. The *future* sampling strategy performs worse on the LunarLander benchmark than the *episode* sampling strategy. We suspect that this is due to the bias of sampling the goals based on future states. These future states have a higher chance of including unfavorable states when contact occurs, which results in similar goals as sampled by the *end* sampling strategy.

It should be noted that the *future* sampling strategy does not always fail. But HER does not discriminate on the quality of the states it samples as its goal, and thus resamples goals from policy rollouts that result in states which are not favorable to resample using HER.

We chose to use the *future* sampling strategy. Our environment is designed such that contact dynamics do not occur. Furthermore, our agent is self-stabilizing due to the PD control loop within our environment. This is not the case for the LunarLander environment. The PD controlled system and the lack of contact dynamics reduce the risk of sampling non-ideal states as goal states. We also have access to valuable demonstration data. This data will most likely contain state-action pairs with low risk of a catastrophic outcome.

We investigated what influence demonstrations had on exploration efficiency, and we compared it to the performance of HER. It is visible from Figure 6.5 that training the LunarLander

in a sparse reward setting is unfeasible purely on random exploration alone. Training the sparse reward environment with HER alone takes a relatively large amount of optimizations compared to demonstrations alone. Demonstrations alone can help to overcome this problem of sparse rewards more efficiently that HER does. This, however, does not mean that HER should not be implemented, since HER helps with generalizing towards different goals. The demonstrations are useful since it ensures that during training there always is data available from successful stable episodes. HER, on the other hand, can generate more training data from these successful rollouts, by sampling additional goals from the demonstration data. This is similar to curriculum learning [10, 15], where the agent and the desired goal states are initialized in such a way that the probability of reaching the desired state is more probable than with random initialization alone.

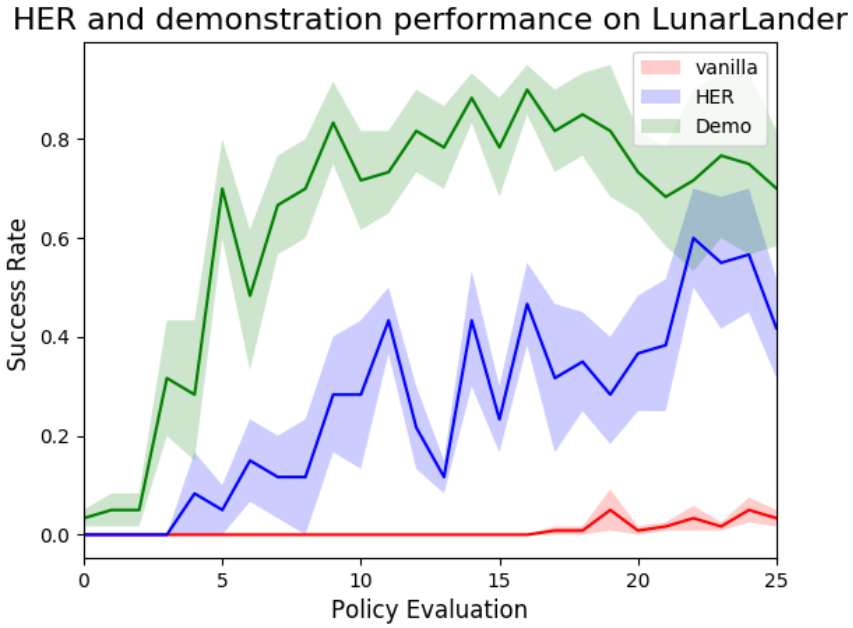

Figure 6.5: Evaluation of Demonstrations, HER (*episode strategy*), and the TD3 algorithm without any additions. For every strategy, we trained a neural network 5 times, each with 25,600 Q-function updates. The results are averaged across 5 random seeds and the shaded areas represent one standard deviation. The goal orientation is held at $0\,rad$ during evaluation.

The demonstration replay buffer has increased benefits since this buffer does not get replaced with new data. A regular replay buffer, on the other hand, does renew its data and deletes old data. This has the disadvantage that potentially valuable data is lost, and less valuable data is added to the buffer. In [16], data from successful rollouts are kept in a replay buffer similarly to our demonstration data. This ensures that valuable data is not removed, and learning can occur from this data. When demonstration data are not available, the methods from [16] and [10] are good alternatives. Quadcopters traditionally are already controlled by humans, so collecting demonstration data is not a limitation in our case.

When both HER and Learning from Demonstrations are absent, no successful policy can be found. The absence of an incentive for exploration makes our LunarLander problem unsolvable.

6.2.2. Results on Quadcopter Environment

The results show that the agent is generally able to reach the goal under a small landing angle. For angles larger than $\frac{\pi}{8}rad$, the success-rate goes down dramatically. For the landing action on inclined surfaces, we see that the policy generally yields in a swinging motion near the goal position to achieve the landing under an angle. The policy exploits the dynamics of the

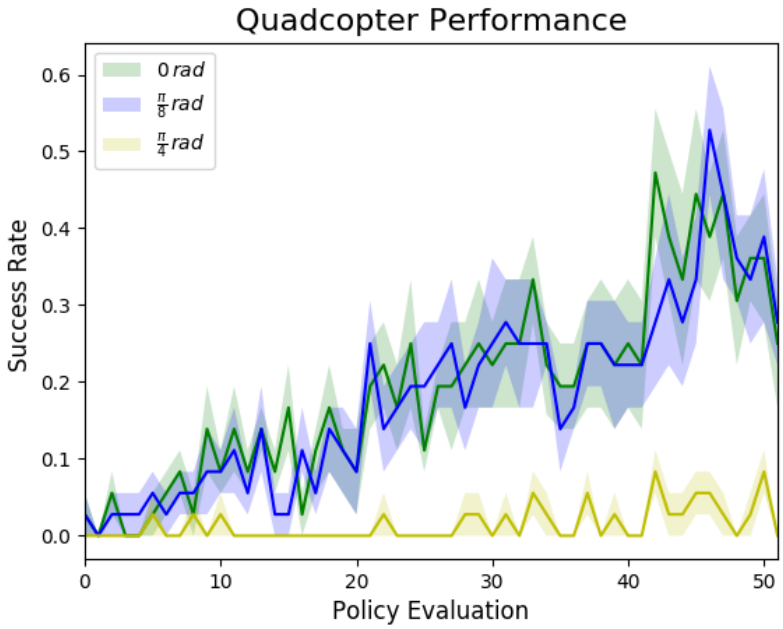

Figure 6.6: Evaluation of our performance for our quadcopter environment with dynamics randomization. The policy is evaluated with different goal orientations. We trained a neural network 10 times each with 51,200 Q-function updates. The results are averaged across 10 random seeds and the shaded areas represent one standard deviation.

PD controlled quadcopter. Here it uses the fact that the quadcopter tilts when it changes direction, and the amount it tilts is dependent on the angle of landing. Figure 6.7 shows that the trained policy results in a slow motion towards the goal combined with a swinging motion. This swinging motion results in the quadcopter pivoting to the angle it should land in, while this also allows for a near-zero velocity at the points where it reverses direction. Figure 6.6 shows the performance of policies trained with dynamics randomization, where the mass m and mass moment of inertia J for all axes are 90% of these original values. This indicates that the dynamics randomization does not fail. However, due to limited computing capabilities, we were not able to train the system on more dynamics parameters.

In Appendix C more figures are shown of our quadcopter performance. These figures show similar results as described above. A collection of trajectories of our policy evaluations for our quadcopter objective is plotted. These plots in Figure **??** show similar swinging behavior for the policies as described above.

Results on PPO

Our PPO algorithm is a combination of RL baselines [13] and SpinningUp OpenAI [1], which is adapted to be able to handle with LSTM based neural networks. Our results show that PPO is not able to find a successful policy with our complex reward function. During training the advantages $A(s, a)$ become zero for every state-action pair. Rewards are extremely sparse, so there is not enough information available to train the policy and value function properly. Since PPO is an on-policy method, our implementations of HER and Learning from Demonstrations are not applicable to PPO.

These results were to be expected, similar to our results on the LunarLander. It is essential to aid the regular exploration in DRL methods when working with complex systems with sparse rewards.

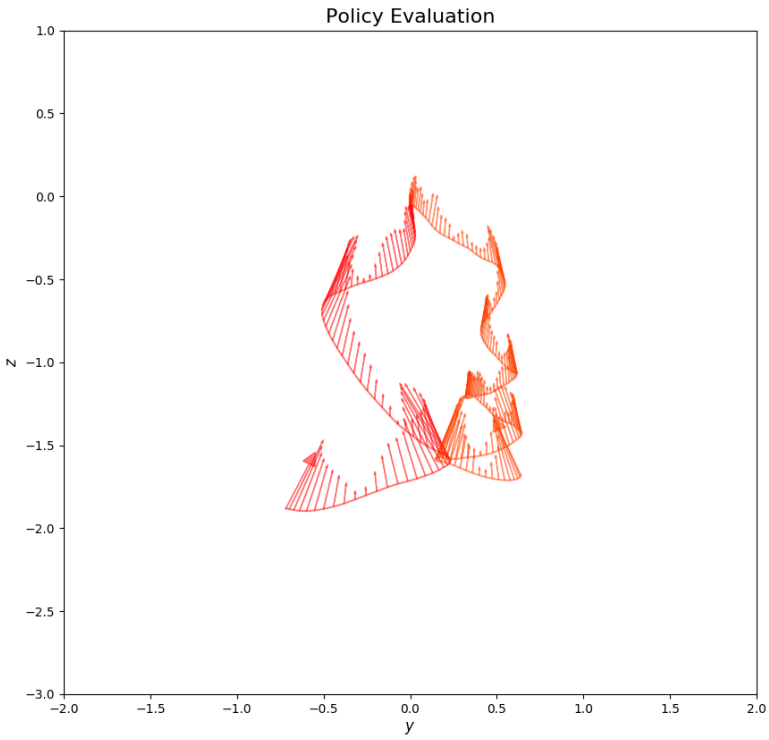

Figure 6.7: Policy of our quadcopter environment with dynamics randomization. The arrows and their length indicate the orientation and the magnitude of the orientation of our quadcopter. The agent starts at position $(0, 0)$ in under an angle of $\theta = 0\,rad$. This policy clearly shows a swinging motion to their goals, while arriving under an angle. The goal position z and y are respectively ± 1, -2 and the landing is defined between $0\,rad$ and $\pm\frac{\pi}{8}\,rad$

7

Conclusion

7.1. Conclusion

Deep Reinforcement Learning (DRL) has great potential to control complex robotic systems. It is already able to achieve superior performance in games, and performance on robotic systems has progressed in the recent past. We investigated if we could land a quadcopter on inclined surfaces with Deep Reinforcement Learning.

From our literature study, we found that to achieve a non-biased policy, the reward function should be a sparse binary reward. Finding sparse rewards in our environment, however, is unfeasible with regular exploration. To aid the regular exploration, we adopted Hindsight Experience Replay (HER), and training from stable demonstration data. Demonstration data are off-policy data and HER creates off-policy data as well. Therefore we are restricted to use off-policy algorithms. Twin Delayed Deep Deterministic Policy Gradient (TD3) is a state-of-the-art algorithm, which compensates for the overestimation of the Q-values, and exploitation of Q-value errors, regularly found in regular Deep Deterministic Policy Gradient (DDPG).

The sampling of goals using HER allowed the agent to explore successfully towards goals which the agent otherwise would have an infinitesimal chance of reaching. Learning from demonstrations allowed us to decrease the number of iterations before a stable policy was found. It also tackled the problem of forgetting in the policy network, since it stored valuable data about stability during the entire training time in a demonstration buffer. The regular replay buffer was not sufficient for our policy network to maintain a stable policy during the entirety of training, showing behavior of forgetting.

Dynamics randomization is used to overcome the reality gap where the RL policy is not able to adapt to real-world dynamics, since the dynamics in simulation are imperfect. This also means that we adopted an LSTM based network structure for our DRL method.

We found that naively implementing our complex system with sparse rewards on RL algorithms, is unfeasible if the RL algorithm does not take measures to counter insufficient exploration. Our method aimed to maximize the number of times rewards were present, and we think that off-policy methods have an advantage over on-policy methods with regards to sparse rewards in complex environments. Our algorithm is relatively simple since our method does not need complicated reward functions which are hard to optimize, are prone to imperfect proxies, and have a chance to add a bias in the policy.

The policy of our quadcopter environment generally does not succeed to meet our expectations with regards to consistency, and performance in landing under larger angles. We created a policy that utilized the dynamics of a quadcopter to achieve a task which, to our knowledge, was not tackled before. Controllers for quadcopters generally are designed to

operate around its marginally stable hovering point, without purposely deviating from this position. We wanted to achieve a maneuver with a quadcopter that does not follow its traditional operation. A controller trained with our DRL method is promising since we can utilize our current knowledge on controlling quadcopters and can design a controller that can reach objectives significantly different from the objectives of the demonstrations it is trained on. Our method, in theory, also allows the training of a quadcopter controller with a different objective, with relative ease. Since we have a sparse reward function, we would only have to assign a set of new goal requirements to our reward function, and an optional modification to the simulator.

Summary

In conclusion, this research has shown that reinforcement learning has the capability to train policies for unconventional control tasks. Our research showed that hindsight experience replay and learning from demonstrations allowed us to train policies for complex tasks in a complex environment with sparse rewards. The resampling of goals of HER allowed us to reach goals that were beyond the scope of the demonstration policy. This method is promising due to its capabilities to generalize towards arbitrary goals. This allows the human to design a controller for robotic systems without needing a deeper understanding of the dynamics of these robotic systems.

7.2. Discussion and Future Work

For consistent performance, our model-free, off-policy RL method with demonstrations and HER might not be sufficient. However, there is a chance that the performance will increase if we would continue training our policy. Figure 6.6 indicates that the policy is not converged, due to the rising performance, without stagnating at 51,200 Q-function updates. Since DDPG methods are sensitive to hyper-parameter tuning, it is also possible that performance will increase with different hyperparameters.

Model-based methods might have an advantage for applying RL policies on flying robotic systems [59], however, this does mean we can only train a policy when the model of our system is accurate. This is often not the case. Guided Policy Search (GPS) [31, 32] is a state-of-the-art machine learning method that has been successfully applied in robotic tasks and is a mixture of both model-based and model-free RL methods. GPS is an extremely sample efficient algorithm, which learns an optimal distribution over optimized trajectories, and uses these trajectories to learn a policy for neural networks through supervised learning. However, our model-free methods with dynamics randomization should be able to adapt to differing real-life dynamics [3, 40]. GPS, on the other hand, has not been shown to work nor not to work when the dynamics of the system are significantly altered. Therefore it is interesting to investigate GPS as an alternative for our method, as well as further investigating our current approach.

Another approach would be to adapt the method used in [25]. This article introduces a model-free and on-policy RL algorithm with deterministic policies implemented on a quadcopter. For our problem, we could extend this method with dynamics randomization. It is mentioned that their quadcopter showed significantly different behavior in the real world compared to simulation. While the behavior in the real world was deemed more stable compared to the simulation, which in many cases is positive, in our case we want to ensure that the policy can handle these discrepancies in the real world. Dynamics randomization would be an option to handle this issue. Furthermore, we would have to tackle the problem of a low probability of retrieving reward in our complex environment. If we would explore our environment with an on-policy method, there is an infinitesimal chance that the agent observes any positive rewards. We could make use of curriculum learning [10], this increases the probability of observing positive rewards. To increase the probability of sufficient exploration for on-policy methods on quadcopters, we could also utilize *exploration rewards* [8]. These exploration rewards are rewards based on dense reward functions, and these rewards are

slowly decayed during training. The last portion of training a policy is trained with purely sparse rewards. Training with an exploration reward, a useful policy is trained to be able to explore successfully in sparse reward settings. This is similar to our demonstration data, where we exploit a sub-optimal policy for the training of a sparse reward setting. To use demonstration data with on-policy methods, a reward function can be designed such that it encourages to match a reference trajectory based on some demonstration data [39]. Combining exploration rewards and the latter, we could create a similar RL approach for on-policy methods compared to our current off-policy approach. However, Hindsight Experience Replay is not as trivial to mimic in on-policy methods.

HER sampling strategies are imperfect. We have seen that depending on the environment, different sampling strategies work best. Further research could be done to choose sampling strategies dynamically during training, based on the quality of the policy rollouts. Measures like the Q-function $Q(s,a)$ could be used to inform a custom sampling strategy. The state action pairs with higher Q-values than others will have a higher chance of being sampled. This is probably why the *future* strategy in [2] worked so well. Many state-action pairs in the neighborhood of the goal will have higher Q-values than ones far away from this goal. At the beginning of training, the Q-values are ill-defined. Slowly the Q-values are better defined, and the policy is trained to approach these high-value states. In [2] the *future* sampling strategy samples goals from higher valued states more often. This simulates the effect as we proposed before. Our LunarLander system on the other hand, has due to its dynamics, less valuable states near the end of the trajectory rollouts than it would have at the beginning of the rollout. An appropriate name of this sampling strategy could be *prioritized* sampling, since the inclusion of information about the value of state transitions for informed sampling has already been done in [49].

Bibliography

[1] Josh Achiam and Peter Abbeel. Spinning up in deep rl. `https://github.com/openai/spinningup`, 2019.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems 30*, pages 5048–5058. Curran Associates, Inc., 2017.

[3] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018.

[4] Rika Antonova, Silvia Cruciani, Christian Smith, and Danica Kragic. Reinforcement learning for pivoting task. *arXiv preprint arXiv:1703.00472*, 2017.

[5] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20. Citeseer, 1997.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[7] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4945–4949. IEEE, 2016.

[8] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[9] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[10] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, page 41–48. Association for Computing Machinery, 2009.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[12] Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv preprint arXiv:1610.03518*, 2016.

[13] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[14] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

[15] Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

[16] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300*, 2017.

[17] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[18] Emil Fresk and George Nikolakopoulos. Full quaternion based attitude control for a quadrotor. In *2013 European Control Conference (ECC)*, pages 3864–3869. IEEE, 2013.

[19] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, volume 15, pages 315–323, 2011.

[21] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[22] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015.

[23] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, 2017.

[26] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life*, pages 704–720. Springer, 1995.

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436, 2015.

[31] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems 27*, pages 1071–1079. Curran Associates, Inc., 2014.

[32] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, volume 28, pages 1–9.

[33] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[34] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[35] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525. IEEE, 2011.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[37] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[38] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6292–6299. IEEE, 2018.

[39] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):143, 2018.

[40] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[41] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[42] Caitlin Powers, Daniel Mellinger, and Vijay Kumar. Quadrotor kinematics and dynamics. *Handbook of Unmanned Aerial Vehicles*, pages 307–328, 2015.

[43] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016.

[44] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[45] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[46] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[47] Stefan Schaal. Learning from demonstration. In *Advances in Neural Information Processing Systems 9*, pages 1040–1046. MIT Press, 1997.

[48] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis Bach and David Blei, editors, *International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1312–1320. PMLR, 07–09 Jul 2015.

[49] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[50] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[51] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[52] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[53] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction(2nd Edition, in preparation)*, volume 2. MIT press Cambridge, 2017.

[54] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[55] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 2012.

[56] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 23–30. IEEE, 2017.

[57] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, volume 2, page 5. Phoenix, AZ, 2016.

[58] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[59] Steven Lake Waslander, Gabriel M Hoffmann, Jung Soon Jang, and Claire J Tomlin. Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3712–3717. IEEE, 2005.

[60] Ian H Witten. An adaptive optimal controller for discrete-time markov environments. *Information and control*, 34(4):286–295, 1977.

[61] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 528–535. IEEE, 2016.

[62] Brian D Ziebart. *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. PhD thesis, Carnegie Mellon University, 12 2010.

Appendices

A

Hyperparameters

This shows our hyperparameters used.

Hyper-Parameter	LunarLanderContinuous-v2	Quadcopter
Critic Learning Rate	10^{-3}	10^{-3}
Actor Learning Rate	10^{-3}	10^{-3}
Actor Delay	2	2
Discount Factor	0.99	0.99
Optimizer	ADAM	ADAM
Target Update Rate	$5 \cdot 10^{-3}$	$5 \cdot 10^{-3}$
Buffer Size	10^6	10^6
Batch Size	256	256
Demo Batch Size	128	128
Reward Scaling	1	1
Recurrent Steps	8	8
Exploration Noise	0.1	0.2

Table A.1: Hyper Parameters of our TD3

Hyper-Parameter	
Steps Per Epoch	4000
Epochs	200
Clip Ratio ϵ	0.2
Optimizer	ADAM
Critic Learning Rate	10^{-3}
Actor Learning Rate	$3 \cdot 10^{-4}$
GAE λ	0.95
Microbatch Size	250

Table A.2: Hyper Parameters of our PPO

B

Neural Networks

B.1. Configuration

This neural network is used for all actor-critic applications for this research [40]. The input for the actor and critic networks contains G the goal requirements, current state s_t and previous action a_{t-1}. The critic also needs information about current state a_t and has information available about the dynamics μ, since we utilize the critic only in simulation.

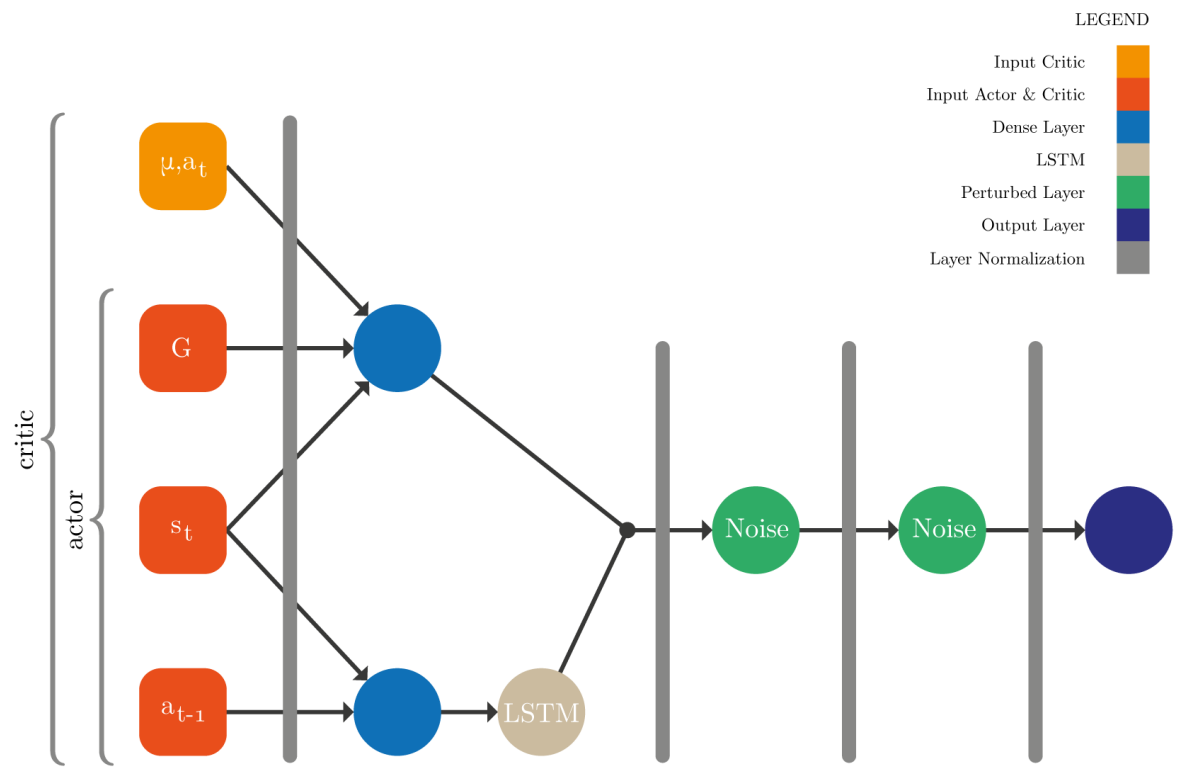

Figure B.1: Our Neural network configurations. Every layer hidden layer has 128 nodes, with ReLU activation functions (except the LSTM unit). The output layer for the critic-network is unbounded, and the output layer for the actor-network is bounded between -1 and 1 using the $TanH$ activation function.

B.2. Parameter Space Exploration

We use a policy network wheres noise is added to a part of the parameter space, for exploration [17, 41, 46]. This is different from the conventional exploration noise added to the action of the policy. The scaling of the parameter noise σ_k is adjusted according to the method of [41]:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d(\pi, \tilde{\pi}) \leq \delta \\ \frac{1}{\alpha}\sigma_k & \text{otherwise.} \end{cases} \tag{B.1}$$

Here d is a distance measure between original policy π and perturbed policy $\tilde{pi}$. The threshold δ is used to determine whether to increase or decrease the perturbations scale σ_k by multiplying or dividing by the scaling factor α for the to be updated perturbed network. For our TD3 algorithm, we calculate $d(\pi, \tilde{\pi})$ as follows [41]:

$$d(\pi, \tilde{\pi}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left[(\pi(s)_i - \tilde{\pi}(s)_i)^2\right]}, \tag{B.2}$$

where N is the dimensionality of the actions, the expectation is estimated over a batch of samples from the replay buffers. The threshold δ is chosen as σ, which results in a normally distributed noise on the action space with standard deviation σ.

Due to this normalizing across activations within a layer, the same perturbation scale can be used across all layers, even though different layers may exhibit different sensitivities to noise.

Since layers may exhibit different sensitivities to noise perturbations, layer normalization [6] ensures that the same perturbation scale can be used across all perturbed layers [41].

C

Supplementary Results

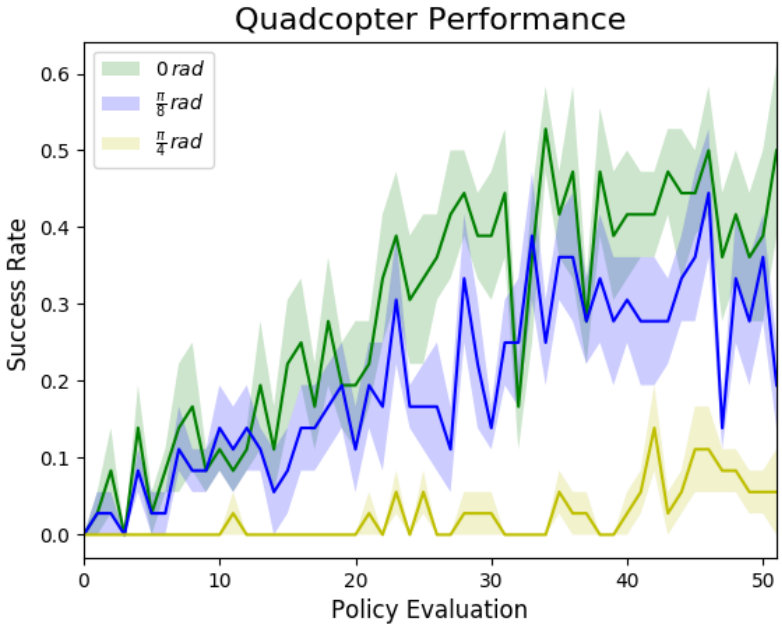

Figure C.1: Evaluation of our performance for our quadcopter environment without dynamics randomization. The policy is evaluated with different goal orientations. We trained a neural network 10 times each with 51,200 Q-function updates. The results are averaged across 10 random seeds and the shaded areas represent one standard deviation.

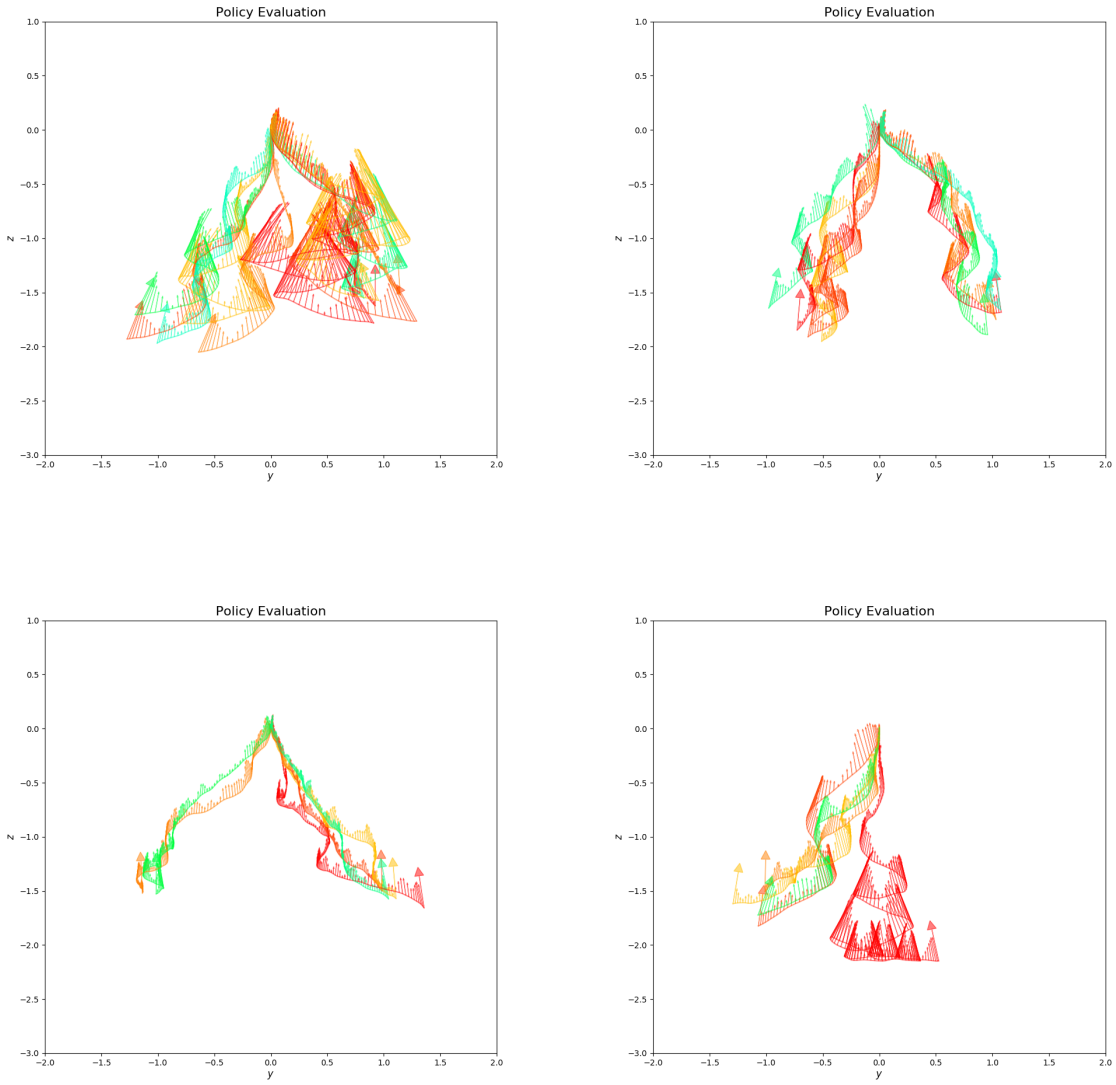

Figure C.2: 4 Policies of our quadcopter environment with dynamics randomization. The arrows and their length indicate the orientation and the magnitude of the orientation of our quadcopter. The agent starts at position $(0, 0)$ in under an angle of $\theta = o\,rad$. The policies clearly show a swinging motion to their goals, while arriving under an angle. The goal position z and y are respectively ± 1, -2 and the landing is defined between $0\,rad$ and $\pm\frac{\pi}{8}\,rad$