Master's thesis

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

Daniel Tavakoli

# Autonomous Drone Landing using Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics

Supervisor: Anastasios Lekkas

June 2020

NTNU
Norwegian University of Science and Technology

Daniel Tavakoli

# Autonomous Drone Landing using Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

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

NTNU
Norwegian University of
Science and Technology

# Abstract

The landing problem has been the topic of research for quite some time in the control engineering community. An abundance of the systems developed in the community have relied heavily on extensive modeling of the plant, before employing advanced control methods whose aim is to impose control laws yielding desired behavior with respect to a specified objective. Emanating from the renaissance machine learning (ML) is experiencing due to exponential growth in computational power in recent years, extensive research has been conducted in the community for exploring the potential of ML. Especially reinforcement learning (RL), a branch within ML, has benefited notably from these advancements. Together with artificial neural networks (ANNs), classical RL disciplines were augmented, giving rise to the emergence of deep reinforcement learning (DRL). Since these DRL methods generally do not need modeling of the plant, it was deemed interesting to explore the potential of such techniques in robotic tasks where both plant and environment is particularly challenging to model and predict. Control of a quadrotor, although extensively studied in the research communities, remains a challenging task for a number of reasons. Modeling alternating and time-dependent aerodynamic forces, restrictions on GPS, limited options for state estimation and the fact that a quadcopter often is underactuated are among these.

As such, this thesis proposes a general framework for adopting a DRL method for optimizing control for autonomous landing of a quadrotor. A quadrotor as the plant was chosen mainly due to its applicability and manoeuvrability, allowing several areas of application. By applying a novel DRL algorithm for control accompanied by a rudimentary planning system for solving the quadrotor landing problem, this thesis investigates the promise of DRL methods for drone control tasks. Two DRL agents with the specific objectives of hovering and descending were designed, trained and tested for safe, efficient and satisfactory landing on a helipad in simulated environments. Based on the results obtained in this thesis, DRL showcases its ability to perform control tasks of complex, nonlinear robotic systems without needing prerequisite knowledge on the plant, nor in-depth descriptions on how the environment is affecting the plant. The findings in the thesis exhibit sufficiently accurate control in unknown environments, and plants a seed suggesting that DRL may become the rule rather than the exception as this particular field of study advances.

# Sammendrag

Landingsproblemet har lenge vært et omdiskutert forskningstema i reguleringsteknikkmiljøet. Et flertall av utviklede systemer har vært sterkt avhengig av omfattende modellering av den kontrollerte prosessen før man utnytter etablerte reguleringsmetoder som har som mål å påtrykke kontrollover som gir ønsket atferd relativt til et spesifisert mål. Som følge av den fornyede tilliten maskinlæring (eng. machine learning, ML) opplever takket være eksponentiell vekst i datamaskiners regnekraft og -kapasitet i senere år, har omfangsrik forskning blitt gjort for å utforske potensialet til ML. Særlig forsterkende læring (eng. reinforcement learning, RL), en gren innen ML, har dratt nytte fra denne fremgangen. Sammen med kunstige nevrale nettverk (eng. artificial neural network, ANN) har klassisk RL blitt utvidet, som har gitt opphav til fremveksten av dyp forsterkende læring (eng. deep reinforcement learning, DRL). I og med at slike DRL-metoder på det generelle plan ikke behøver en modell av prosessen, ble det dermed ansett som interessant å undersøke potensialet til slike teknikker i robotikkapplikasjoner hvor både prosessen og omgivelsene er nevneverdig utfordrende å modellere og predikere. Til tross for at forskningsmiljøer utført omfattende studier rundt temaet, forblir regulering av en drone med fire rotorer en utfordrende oppgave som følge av en rekke grunner. Modellering av tidsavhengige og fluktuerende aerodynamiske krefter, posisjonering i områder hvor man opplever restriksjoner på GPS-signaler, begrensede alternativer for tilstandsestimering og faktum at en drone generelt sett er underaktuert er noen av disse.

Følgelig foreslår denne masteroppgaven et generelt rammeverk for å bruke en DRL-metode for å optimere regulering av en drone for autonom landing. En drone bestående av fire rotorer som prosess ble valgt, hovedsakelig grunnet dens anvendbarhet og bevegelighet, som tillater et bredt spekter av bruksområder. Ved å anvende en DRL-metode som tilbyr ny tilnærming til å løse dronelandingsproblemet, samt et rudimentært planleggingssystem, undersøker oppgaven potensialet DRL har relativt til regulering av droner. To DRL-agenter med spesifikke mål, henholdsvis hovere og synke, ble utformet, trent og testet for trygg, effektiv og kravoppfyllende landing på en landingsplattform i et simulert miljø. Basert på resultatene denne masteroppgaven har kommet frem til viser DRL dens evne til å regulere komplekse, høyt ulineære robotikksystemer uten forhåndskunnskaper om pros-

essen, ei heller informasjon om det rundtomliggende miljøet. Oppgavens funn fremstiller i tilstrekkelig grad nøyaktig regulering i ukjente omgivelser og bygger under påstanden om at dyp forsterkende læring kan utvikle seg til å bli et rammeverk som erstatter mer tradisjonelle reguleringsmetoder når dette forskningsområdet utvikler seg ytterligere.

# Preface

This master's thesis constitutes the culmination of my work completed at Norwegian University of Science and Technology (NTNU) through the spring of 2020. Supervised by Anastasios Lekkas, the work summarizes methodologies used and presents the corresponding findings this resulted in. Historically, the proposed methods have seldom been used due to limitations in data resources and computational performance. However, several major advances has been done in recent years to rekindle the interest for such methods. Inspired by the advancements in the field of machine learning due to exponentially increasing computational power, it was compelling to see how techniques based on a conjunction of machine learning and reinforcement learning, so-called deep reinforcement learning, could be applied in control tasks of robotic plants. As such, this thesis seeks to contribute by applying deep reinforcement learning aimed to solve the autonomous landing problem for a quadcopter.

Although offering an extensive presentation of the theory adopted, this thesis was designed under the assumption that the reader inhabits prerequisite knowledge with respect to advanced control theory, modeling, simulation and optimization. Albeit not strictly necessary, rudimentary understanding of machine learning and how this theory can be applied to other tasks is advantageous.

There are several contributions supplementing and assisting the development of this thesis. First and foremost, this master's thesis serves as an augmentation to the project thesis written in the fall of 2019, where basic understanding of the theory and more primitive algorithms were developed and initially presented. As such, segments in this thesis stems from this preceding work [1]. Secondly, NTNU's Faculty of Information Technology and Electrical Engineering provided a Dell Optiplex 7040 i7-6700 computer with Ubuntu 16.04 and ROS Kinetic for development. Further, Udacity's tutorials in DRL implementations from their nanodegree program [2], OpenAI's `gym` toolkit [3] and `Spinning Up` framework were fundamental sources of inspiration for setting up and developing the DRL methods. The programming language Python in addition to open-source software including ROS, Gazebo, NumPy and Tensorflow were vital for the development of this thesis. Several open-source ROS packages also formed the basis for the simulatory environment

for the used drone, among these being `ardrone_autonomy` [4], `tum_simulator` [5] and `hector_gazebo` [6]. Supervisor Anastasios Lekkas and colleague Thomas Sundvoll have also aided with various subjects, both for the setup for the ROS and Gazebo frameworks and the design and implementation of the DRL methods. Although developing independent work, Thomas and I have developed interconnected theses, where the ultimate goal was to merge our efforts to create a stand-alone drone system able to land on a platform using perception. As such, the sub-missions and overall objectives have been discussed extensively, as well as the mission structure and how to consolidate our two systems together to one, complete solution. Note that figures in this thesis that do not specifically state a source of reference are created independently. Also note that ”drone”, ”quadrotor” and ”quadcopter” are interchanged throughout this thesis, where all terms symbolize a multirotor helicopter consisting of four rotors.

With the support and guidance of Tom Arne Pedersen, the work in this thesis was initially aimed to be applied with DNV GL's Revolt vessel. Preferably, the developed system would be subject to testing in a laboratory, an open field, and finally the ocean with the Revolt in order to test its robustness in both controlled environments and real-life settings. Additionally, it would be advantageous to benchmark the derived solution relative to more traditional methods for comparative purposes. However, due to COVID-19, the scope of the thesis and, resultantly, the final product had to be scaled down.

# Acknowledgments

To my mom and dad, thank you for everything.

# Contents

**Abstract** i

**Sammendrag** iii

**Preface** v

**Acknowledgments** vii

**Contents** ix

**List of Tables** xi

**List of Figures** xiii

**Symbols** xix

**1 Introduction** 1
- 1.1 Background and motivation . . . 1
- 1.2 Objective and method . . . 3
- 1.3 Outline . . . 4

**2 Theoretical background** 5
- 2.1 Supervised learning . . . 5
  - 2.1.1 Artificial neural networks . . . 5
  - 2.1.2 Deep learning . . . 8
  - 2.1.3 Training DL models . . . 9
- 2.2 Reinforcement learning . . . 11
  - 2.2.1 Markov decision processes . . . 12
  - 2.2.2 Rewards . . . 14
  - 2.2.3 The Bellman equation . . . 15
  - 2.2.4 Temporal difference learning . . . 17

2.3 Deep reinforcement learning . . . 19
2.3.1 Policy gradient methods . . . 20
2.3.2 Actor-critic methods . . . 23
2.3.3 Deep Deterministic Policy Gradient . . . 26
2.4 Quadrotor dynamics . . . 29

**3 Experimental setup** **33**
3.1 Software frameworks . . . 33
3.1.1 Gazebo . . . 33
3.1.2 ROS . . . 34
3.1.3 Tensorflow . . . 34
3.2 Quadrotor platform . . . 35
3.2.1 Hardware and sensors . . . 35
3.2.2 Built-in velocity controller . . . 36

**4 Methodology and system design** **41**
4.1 System architecture and setup . . . 41
4.2 Quadrotor control using DDPG . . . 44
4.2.1 State and action representation . . . 49
4.2.2 Reward function . . . 53
4.2.3 Case study using perception estimate . . . 57

**5 Results** **61**
5.1 Training framework . . . 61
5.2 Ground truth results . . . 65
5.2.1 Hover . . . 66
5.2.2 Descend . . . 73
5.3 Case study results . . . 79

**6 Future work** **85**

**7 Conclusion** **87**

**Bibliography** **89**

# List of Tables

5.1 Table encapsulating the parameters of the DDPG solution for both the hover and descend agent. . . . . 61

# List of Figures


2.1 Illustration of a perceptron consisting of an input vector of dimension 3 and a bias term. $\boldsymbol{x}$ is inputted to the perceptron and multiplied with $\boldsymbol{w}$. The bias term is subsequently added to the product. This sum progresses through a step function and generates the output $y$. . . . . 6
2.2 Illustration of a neuron with nonlinear activation consisting of an input vector of dimension 3 and a bias term. . . . . 7
2.3 An example of a fully connected artificial neural network with one hidden layers, with all their weights and biases stated. . . . . 8
2.4 Chart conceptualizing the difference between classical ML methods and DL methods. The light blue processes indicate the location of where the agents learn, i.e. the position of the neural networks in the systems. . . . . 9
2.5 Computational graph illustrating the forward pass and the resulting back-propagation steps for computing the loss gradient. . . . . 11
2.6 The feedback dynamics of the reinforcement learning problem. . . . . 12
2.7 The decision network of an MDP with transition function $\mathcal{P}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')$ and reward function $\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')$. . . . . 13
2.8 The standard architecture of an actor-critic method. . . . . 25
2.9 A visual representation of the relationship between the world frame $\mathcal{W}$, where the axes are denoted with $w$, and the body frame of the drone $\mathcal{B}$, denoted with $b$. The rotation directions for the body reference frame are also illustrated, where the right-hand rule is the convention applied. These coordinate frames form the basis for the extraction of the equations of motion stated in (2.47). . . . . 31

3.1 The controller realized in the `tu-darmstadt-ros-pkg` ROS package, in which `tum_simulator` bases their solution on. The block diagram is taken from [67]. . . . . 37

3.2 Exemplification of how the velocity controller's design introduces the coupled behavior between the horizontal velocities and the roll and pitch angle. At $t = 4$ the drone receives a velocity command purely in $x$. This induces a response in the pitch angle such that the drone tilts and generates a horizontal force, pushing the drone in positive $x$ direction. At $t = 8$ the same incident occurs, although in the $y$ direction. This induces a *negative* roll angle to generate velocity in the positive $y$ direction. Figure courtesy of [37]. . . . . 38
3.3 Simulation highlighting the drifting conditions of the drone's pose, namely its position and orientation. The simulation was conducted over a 60 second interval, where the drone was initialized at $[x, y, z, \phi, \theta, \psi] = [0.0, 0.0, 2.0, 0.0, 0.0, 0.0]$. There were no applied velocity commands during the simulation. Figure courtesy of [37]. . . . . 39

4.1 Flow chart of the different steps to be conducted for completing the objective. Using conventional flow chart notation, the activities are in rectangles and the decisions in diamonds. Each decision has a set of specific requirement to output either yes or no. . . . . 42
4.2 The simple simulation environment modeled in Gazebo during training. . 43
4.3 A figure highlighting the components and the signal flow of the system developed in this thesis. The DRL controller is highlighted in blue and the environment encapsulating the quadrotor platform in green. . . . . 45
4.4 The network architecture of the actor and critic in the implemented DDPG solution. The state and action dimensions are denoted $n$ and $m$, respectively. 47
4.5 A more detailed block diagram portraying the architecture of the system developed in this thesis. The figure augments Figure 4.3 and illustrates the controller, highlighted in blue, and the environment, including the quadrotor platform, in green. Further, the figure introduces the replay buffer in addition to the modules that the quadrotor platform consists of, namely the state estimator and built-in velocity controller. The two latter entities are boxed in red. . . . . 48
4.6 A bird's-eye view illustration of the relationship between $\mathcal{W}$ and $\mathcal{B}$ in the quadrotor environment. The frame $\mathcal{B}'$ symbolizes the body frame translated to the world frame origin. The figure bases its axes upon the right-hand rule, such that the $z$ axes of all three frames point outwards from the paper. . . . . 50
4.7 Comparison between a boundary function with $a = 1.0$ and $b = 1.0$, and a Gaussian distribution with $a = 0.0$, $\mu = 0.0$ and $\sigma = 1.0$. The figure illustrates that a smooth function such as the Gaussian allows the agent to experience small increments in rewards even when the error is high. This detail may greatly benefit the agent's convergence towards desired behavior. 55

4.8 The landing platform, also referred to as helipad, seen from a bird's-eye view. The helipad was designed such that the perception module would have the required amount of features to work with for the computer vision modules. The H is of size 0.20 m $\times$ 0.28 m. The distance from the center of the helipad to the arrow tip is 0.30 m, while the radii of the orange and green circle are 0.26 m and 0.40 m, respectively. Figure courtesy of [39]. . 59
4.9 The augmented simulation environment modeled in Gazebo when using the perception estimates. Figure courtesy of [39]. . . . . 59

5.1 The reward and number of steps for each episode in the hovering agent's training phase. . . . . 69
5.2 Five tests of how the hovering agent approaches the setpoint, in dotted lines, with varying initial positions. The horizontal axes illustrate time steps, while the vertical axes constitute the drone's position in the three dimensions. . . . . 70
5.3 A bird's-eye view of the tests illustrated in Figure 5.2. The green and red crosses constitute the starting and finishing positions, respectively. The unit of all axes are given in meters. . . . . 71
5.4 Means and standard deviations of the drone's linear position and positional error relative to the hovering setpoint. The errors are given in meters and are calculated under the assumption that the quadrotor is within close proximity of its setpoint. The results portrayed were subject to the same initial positions as in Figure 5.2, where the agent was spawned at each initial position 20 times. This resulted in 100 independent runs. . . . . 72
5.5 The reward and number of steps for each episode in the descending agent's training phase. . . . . 75
5.6 Eight tests of how the descending agent approaches the setpoint, in dotted lines, with varying initial positions. The horizontal axes illustrate time steps, while the vertical axes constitute the drone's position in the three dimensions together with the yaw angle. $\psi$ is converted to radians in the figures for visual purposes. . . . . 77
5.7 Means and standard deviations of the drone's error in linear position $\tilde{\boldsymbol{x}}$, yaw angle $\tilde{\psi}$ given in radians and the pseudo-Euclidean distance $||\tilde{\boldsymbol{x}}_a||$, where latter is also computed with the yaw angle being given in radians. Further, the positional error $||\tilde{\boldsymbol{x}}||$ is illustrated for comparative purposes with respect to the hover agent given in Figure 5.4. . . . . 78
5.8 Initial position at $\boldsymbol{x}_a = [-0.55, 1.53, 2.50, 9.34]^\intercal$ relative to the helipad. The system switches from hover to descend at step 34, and lands after 70 steps, meaning it completed hovering after 11 seconds and initiated landing after 24 seconds. Footage of the test. . . . . 81
5.9 Initial position at approximately $\boldsymbol{x}_a = [-1.14, -1.81, 3.02, -0.47]^\intercal$ relative to the helipad. The system switches from hover to descend at step 45, and lands after 81 steps, meaning it completed hovering after 15 seconds and initiated landing after 27 seconds. Footage of the test. . . . . 82

5.10 Initial position at approximately $\boldsymbol{x}_a = [-1.02, -0.23, 3.56, -4.96]^\intercal$ relative to the helipad. The system switches from hover to descend at step 39, and lands after 76 steps, meaning it completed hovering after 13 seconds and initiated landing after 26 seconds. Footage of the test. . . . . 82
5.11 Initial position at approximately $\boldsymbol{x}_a = [-1.26, -1.12, 4.65, 2.93]^\intercal$ relative to the helipad. The system switches from hover to descend at step 29, and lands after 61 steps, meaning it completed hovering after 10 seconds and initiated landing after 21 seconds. Footage of the test. . . . . 83
5.12 Initial position at approximately $\boldsymbol{x}_a = [-0.29, -2.59, 4.01, -0.71]^\intercal$ relative to the helipad. The system switches from hover to descend at step 69, and lands after 100 steps, meaning it completed hovering after 23 seconds and initiated landing after 34 seconds. Footage of the test. . . . . 83

# Nomenclature

| | |
|---|---|
| AI | Artificial intelligence |
| ANN | Artificial neural network |
| DDPG | Deep Deterministic Policy Gradient |
| DL | Deep learning |
| DNN | Deep neural network |
| DP | Dynamic programming |
| DRL | Deep reinforcement learning |
| MDP | Markov decision process |
| ML | Machine learning |
| MLP | Multilayer perceptron |
| MPC | Model predictive control |
| PGM | Policy gradient method |
| PID | Proportional-integral-derivative |
| RL | Reinforcement learning |
| ROS | Robot Operating System |
| TD | Temporal difference |
| UAV | Unmanned aerial vehicle |

# Symbols

$\boldsymbol{W}^l$ Weighting matrix for layer $l$.

$\boldsymbol{b}^l$ Bias vector for layer $l$.

$f^l$ Activation function for layer $l$.

$\boldsymbol{\theta}$ Network parameters of an arbitrary ANN.

$J$ Loss function.

$\boldsymbol{h}^l$ Hidden layer operation of layer $l$.

$\mathcal{S}$ Set of states for an agent.

$\mathcal{A}$ Set of actions for an agent.

$\mathcal{P}$ Transition function for an agent.

$\mathcal{R}$ Reward function for an agent.

$\gamma$ Discount factor.

$\boldsymbol{s}$ Vector describing the state of the agent.

$V(\boldsymbol{s})$ Value for a given state.

$\pi(\boldsymbol{s})$ The action policy of an agent for a given state.

$\boldsymbol{a}$ Vector describing the action chosen by the agent.

$Q(\boldsymbol{s}, \boldsymbol{a})$ Q-value for a given state-action pair.

$\alpha$ Learning rate of an agent.

$\mu(\boldsymbol{s}|\boldsymbol{\theta}^\mu)$ Actor network returning the optimal action of a state, given the actor network parameters.

$Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^Q)$ Critic network returning the Q-value of a state-action pair, given the critic network parameters.

Chapter 1

# Introduction

## 1.1 Background and motivation

The autonomous landing problem for unmanned aerial vehicles (UAVs) has been a widely studied topic in recent years as communities within control theory, aerial robotics and artificial intelligence realize the potential of such systems. Their ability of remote operation in addition to the capability to maneuver in tight spaces render UAVs highly useful with respect to search-and-rescue operations [7], inspection missions [8], surveillance [9], border patrol [10], mapping [11], transportation [12] and identification of agriculture [13].

Accordingly, extensive research has been dedicated to this topic and has resulted in rich literature, covering the vast possibilities for solving these missions as robustly and efficiently as possible. Comprehensive work has been conducted in order to develop aerial systems able to autonomously land based on available situational awareness. Many solutions adopt both low-level control accompanied by a planning system responsible for the decision-making, which any rudimentary autonomous system must include.

With respect to the low-level control aspect of the problem, systems deploy different techniques within the field of robotics in order to realize the landing objective. Among these is model predictive control (MPC), a vastly researched and well-established control principle which has shown promising results in previous work related to UAV landing [14, 15, 16, 17]. The ability to enforce time-dependent constraints and account for additive disturbances during the modeling phase are some of the main attributes of MPC. Additionally, the algorithm offers great flexibility for the control engineer, as adjusting the objective function weights will directly influence the transient response of the system. These aspects separate MPC from more traditional controller schemes, such as the proportional-integral-derivative (PID) controller [18, 19]. Moreover, they are deemed essential when dealing with problems that often require aggressive maneuvering, such as small UAVs experiencing wind gusts.

Though exhibiting promising results when successfully deployed, such model-based approaches generally require a highly accurate model of the plant in order to derive controllers yielding satisfactory performance. Obtaining such accuracy during modeling is nothing short of challenging, especially for an in-flight UAV that can experience dynamic and aerodynamic nonlinearities depending on its position in the environment and even the slightest misalignment between rotor blades [20, 21]. Given inaccurate model dynamics, the error will propagate and render the state predictions imprecise, yielding insufficient performance [22, 23]. Furthermore, a predefined model may even be rendered inaccurate *after* being exposed to the environment due to outdated modeling of the environment or even the plant. Such cases might include substantial turbulence, change of propellers on a drone or a significant amount of rain that was not accounted for during modeling.

Given that both plant and environment can be challenging to predict during operation, the idea of untying the controller design from modeling naturally follows. Using approaches that are not as heavily dependent on model accuracy could lead to more flexible systems robust to changes in both the plant and environment. As machine learning (ML) is experiencing a renaissance due to the exponential growth in computational power and availability of large datasets, ML based methods were deemed promising candidates for solving the aforementioned predicaments. For many years the Artificial Intelligence (AI) communities have developed rich and robust literature for optimal system performance under uncertainty. Methods based on this framework go under the term reinforcement learning (RL).

Based on evaluative feedback [24] rather than instructive feedback, RL based methods are able to train a system to act satisfactorily without knowing the model of the plant nor its environment beforehand. The agent, analogous with the controller in traditional schemes, interacts with the environment and receives a scalar reward based on a predefined reward function. Since it directly relates to the agent's objective, the reward function is often described as the core of each RL algorithm. The agent will process the actions, the control signals, it received rewards for and the actions it received penalties for. Based on its experiences while exploring the environment the agent will derive a policy, a control law, that achieves the control objective in the most optimal way.

Albeit RL methods in general offer several advantages given that they are model-free, it is worth noting that they pose challenges that need to be taken into account from an engineering standpoint. Since the agent's learning process is based on exploring the environment, such methods may have to perform exhaustive searches over the entirety of both state and action space. This increases the time consumption before an adequate policy is derived. Furthermore, complexity and required memory escalate as the dimensionality of the problem increases. Lastly, RL algorithms seldom yield convergence guarantees due to the nature of how policies are calculated in these methods, which may directly affect the resulting policy extracted from the agent.

As a means of mitigating these challenges, the work presented by DeepMind [25] combined multilayered artificial neural networks (ANNs), coined Deep Neural Networks (DNNs), with RL principles in order to approximate values that previously had to be calculated through computationally expensive operations. This was seen as a breakthrough in the

RL community. RL methods using DNNs as function approximators were fittingly coined Deep Reinforcement Learning (DRL), with DRL giving rise to some very interesting results in multiple application areas. In addition to DeepMind playing several Atari games, the world has also seen agents beating the world champion in Go [26] and landing a UAV [27]. The authors in both [28] and [29] train DRL agents for controlling a quadrotor. The results show that the employed techniques are able to outperform MPC with respect to accuracy while also being significantly less computationally expensive.

Lillicrap et al. [30] augmented DeepMind's scheme in order to handle continuous action spaces as well as continuous state spaces, which fitted control tasks in particular. In addition to the results presented in the paper, the community witnessed ravishing success in various complex problems characterized by continuous action spaces, for instance robot manipulation [30], gameplay [31], robotic locomotion [32] and curling [33]. The authors in [34] present an overview of the principles for utilizing DRL for control. Path-following and control of marine vessels is demonstrated in [35], where the DRL motivated solutions outperform more traditional marine vessel control schemes. Implementation of a velocity controller for quadrotor control is done in [36]. The paper shows that the agent is able to outperform a well-tuned proportional-integral-derivative controller (PID) with respect to tracking accuracy and robustness. The work showcased in [37] solves the quadrotor hovering problem using an end-to-end DRL solution for control. This was done with relatively short training time while still maintaining high precision and general performance. The authors in [38] also use DRL to train a UAV system to land on a moving platform by learning reference velocities.

## 1.2 Objective and method

This master's thesis aims to investigate how to apply DRL as a control scheme to a quadrotor system in an unknown environment in order to successfully land on a fixed platform. The thesis encapsulates research on how to solve the autonomous landing problem on a fixed platform, in addition to deploying the system to a simulatory setting. The work presented serves as an augmentation of the project thesis completed in the preceding semester [1]. Additionally, it is interconnected with the work done in [39], where a perception system is developed for estimating the pose of the drone through imagery. The long-term goal is to create a robust planning and controlling system able to land on a platform that is detected using the aforementioned perception system.

The proposed solution consists of two components:

- The controlled system, namely the UAV. This unit contains the information required for carrying out the task of control through only observing the environment state.
- The DRL agent, subject to training for optimizing performance. The agent comprises a control policy whose objective is to map the environment's state to a control signal, in addition to a value function returning a measure of how beneficial the chosen control was in the given state.

In the proposed architecture both the control policy and value function are represented by

ANNs, and are learned through trial-and-error. The DRL agent deployed is based on the Deep Deterministic Policy Gradient (DDPG) algorithm [30], which is discussed in detail in Chapter 2.

The reasoning for putting such strong emphasis on DRL based methods mainly stems from the fact that modeling of both UAV dynamics in addition to its surroundings is often a challenging, if not infeasible, task to execute. Recent results in the community, discussed in Section 1.1, suggest that these methods have the potential to outperform the more traditional control methods. Model-based approaches such as MPC are notoriously known for yielding satisfactory results when the model is accurate and disturbances are precisely accounted for during the modeling process. For UAV control, this is not always possible to do in real-life applications due to wind gusts, sudden changes in aerodynamics due to external effects and other unforeseen events that may affect the system during operation [21, 40]. Considering that DRL based methods are characteristically model-free approaches and can resultantly *learn* how to tackle the aforementioned challenges, it is interesting to explore the possibilities of DRL as an alternative to classic UAV control for the drone landing problem.

## 1.3 Outline

This thesis is partitioned into 7 chapters, where Chapter 2 introduces basic notation along with fundamental theory and methods. The goal of this chapter is to establish fundamental concepts, followed by the introduction of methods to be used in the later implementations. The chapter opens with an overview of ANNs and how to train such models, before a comprehensive section revolving around RL is presented. Finally, the chapter is concluded with a model of a general quadrotor with 6 degrees of freedom. Chapter 3 describes the experimental setup, where the drone specifications and how it is controlled is presented, in addition to displaying the software frameworks used in the thesis. Chapter 4 describes the system design of the planner and control methodologies developed, and Chapter 5 reports the corresponding results obtained. Chapter 6 suggests possible improvements and additions that can be conducted in future work. Finally, the thesis is summarized and concluded in Chapter 7.

Chapter 2

# Theoretical background

This chapter aims to present the fundamental theory for developing a DRL agent capable of completing the landing problem under environmental uncertainties. As the control principles used to accomplish this are based on several disciplines within machine learning, the subsequent sections aim to present two of the main pillars within ML, namely supervised learning and reinforcement learning. Despite our proposed solution being based on RL, the principles within supervised learning, particularly ANNs, play a vital role in the recent progress in DRL.

## 2.1 Supervised learning

Machine learning represents a field of study in computer science where machines have the ability to learn tasks without being explicitly programmed. Supervised learning is one of the mainstays in ML and encapsulates, in layman's terms, learning through labeled data.

The agent is given examples from a labeled dataset, i.e. $(x, y) \in \mathcal{D}$, where the relationship between the known input $x$ and the known output $y$ is defined by the unknown function $g$, namely $y = g(x)$. The overall goal of is to obtain an approximation for this function, $\hat{g}$, given the examples. There are various ways to build such approximators depending on the complexity of the input, output and the relationship between them.

### 2.1.1 Artificial neural networks

Artificial neural networks (ANNs) prove to be well suited for continuous inputs and outputs, as well as being able to approximate highly nonlinear functions. As discussed further in Section 2.3, traditional reinforcement learning algorithms suffer from learning becoming intractably slow with a growing state space [41]. Luckily, ANNs came to the rescue and provided a framework for adopting parameterized function approximation to mitigate

the so-called representation problem. As such, these networks proved to be crucial building blocks for the emergence of deep reinforcement learning methods.

Inspired by the structure of neurons found in animal brains, the ANN is a learning algorithm constituted by a set of interconnected nodes, also called artificial neurons. The simplest form of these artificial neurons is the *perceptron*, presented by Frank Rosenblatt in the 1950s [42].

The perceptron is assembled by a weighting vector and a bias. Given an input vector, $\boldsymbol{x} = [x_1,\ x_2,\ \dots]^\intercal$, the weighting vector describing the importance of each element in the input vector, $\boldsymbol{w} = [w_1,\ w_2,\ \dots]^\intercal$, and a bias term $b$, the output of the perceptron $y$ can be described by the following equation:

$$y = \begin{cases} 1, & \text{if } \boldsymbol{w}^\intercal \boldsymbol{x} + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

(2.1) is depicted in Figure 2.1 where the dimension of the input is 3. The bias serves as an adjustment term that illustrates how easily a perceptron is *activated*. Note that the bias can be interpreted as a weight itself, where the corresponding input is a constant set to 1.

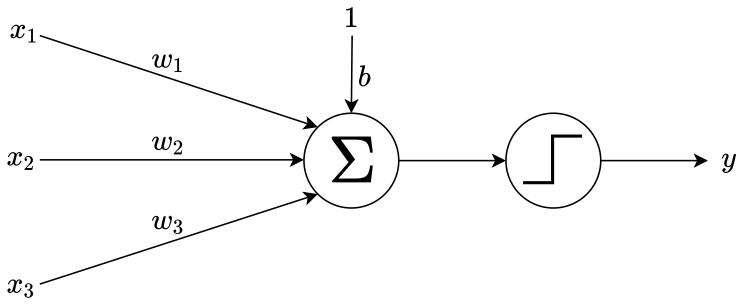



**Figure 2.1:** Illustration of a perceptron consisting of an input vector of dimension 3 and a bias term. $\boldsymbol{x}$ is inputted to the perceptron and multiplied with $\boldsymbol{w}$. The bias term is subsequently added to the product. This sum progresses through a step function and generates the output $y$.

Perceptrons are mainly used for binary classification, i.e. tasks where one wishes to decide which class a data point belongs to. A *linear* decision boundary $\boldsymbol{w}^\intercal \boldsymbol{x} + b = 0$ decides if $\boldsymbol{x}$ is in class 0 or 1. This subsequently renders a perceptron inadequate for problems that do not conform to binary classes nor linear classifications.

It was recognized by the community that building a *network* consisting of *layers* of perceptrons would mitigate these shortcomings. This type of network was coined multilayer perceptron (MLP), and is known as the first adaptation of artificial neural networks. The idea of such internal representations was first presented in [43]. The additional perceptrons enabled MLPs to extract higher level features from the input and could thus solve more challenging tasks.

In order to further increase flexibility in addition to limit the required number of neurons in MLPs, activation functions were introduced. Instead of a standard Heaviside step function,

the MLPs were now allowed to have continuous functions allowing continuous output in a range. Evolving from (2.1), the general neuron with activation $f$ was given as

$$y = f(\boldsymbol{w}^\intercal \boldsymbol{x} + b), \tag{2.2}$$

and is illustrated in Figure 2.2.

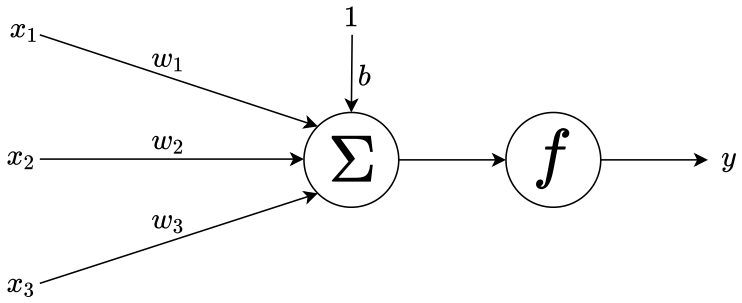



**Figure 2.2:** Illustration of a neuron with nonlinear activation consisting of an input vector of dimension 3 and a bias term.

In an ANN neurons are aggregated into layers. A layer $l$ is represented by a weighting matrix $\boldsymbol{W}^l$ and a bias vector $\boldsymbol{b}^l$. Further, each neuron in a layer is assumed to have the same activation function $f^l$. Given that the input to layer $l$ is the output of the previous layer $l-1$, the output of $l$ is given as

$$\boldsymbol{y}^l = f^l(\boldsymbol{W}^l \boldsymbol{y}^{l-1} + \boldsymbol{b}^l), \tag{2.3}$$

where $\boldsymbol{y}^{l-1}$ is the output from layer $l-1$. $\boldsymbol{W}^l$ constitutes weights between neurons in $l-1$ and neurons in $l$. $w^l_{jk}$ expresses the weight from neuron $j$ in layer $l-1$ to neuron $k$ in layer $l$, while $b^l_k$ is the bias of the $k^{\text{th}}$ neuron in layer $l$. An example of a network illustrating this notation is depicted in Figure 2.3. The network consists of 2 hidden layers and all layers are fully connected, i.e. every neuron in one layer is connected to every neuron in the successive layer.

By studying the figure and using the notation presented in (2.3), the weighting matrix and bias vector for the hidden layer can be expressed as

$$\boldsymbol{W}^1 = \begin{bmatrix} w^1_{11} & w^1_{21} \\ w^1_{12} & w^1_{22} \\ w^1_{13} & w^1_{23} \end{bmatrix} \text{ and } \boldsymbol{b}^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}. \tag{2.4}$$

Given the input $\boldsymbol{x}$, one can now compute the output of the first hidden layer, $\boldsymbol{y}^1$, which is the input to the output layer. This scheme illustrates the principle of *forward pass*, which render networks of this structure *feed-forward ANNs*.

Feed-forward ANNs possess great potential, as described by the *universal approximation theorem* [44]. It states that a feed-forward network with a single hidden layer consisting

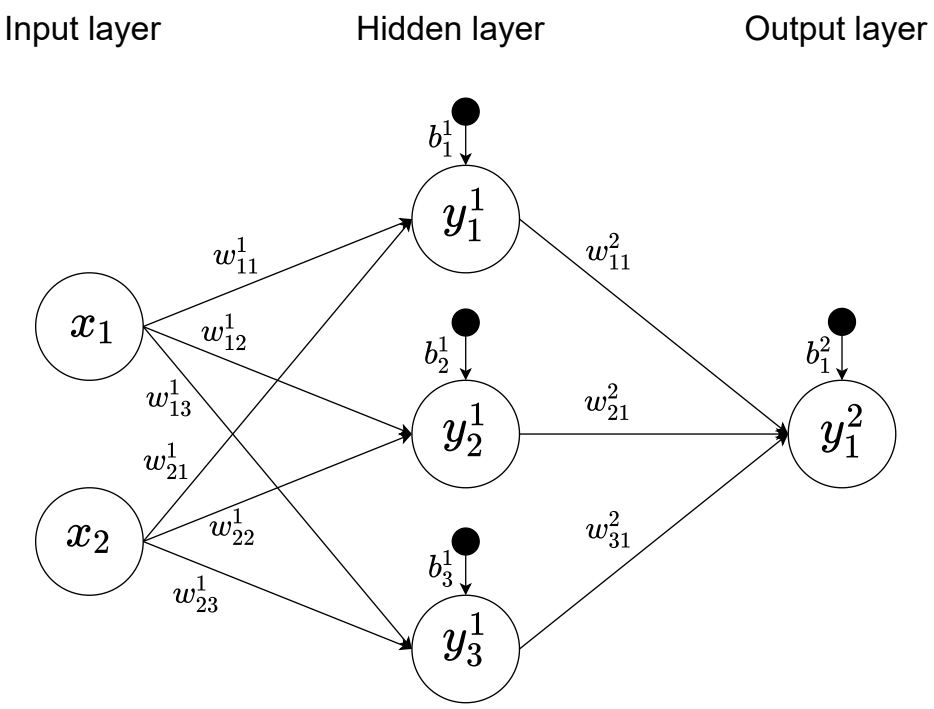



**Figure 2.3:** An example of a fully connected artificial neural network with one hidden layers, with all their weights and biases stated.

of a finite number of neurons has the ability to approximate any continuous function on a compact subset of $\mathbb{R}^n$, under the assumption that the activation function $f$ is non-constant, bounded and continuous. In other words, this theorem establishes that ANNs can, in theory, solve any problem that reduces to function approximation.

### 2.1.2 Deep learning

Depending on the application, various structures to ANNs are utilized. As computational power has increased drastically in recent years, most applications no longer limit themselves to only one layer in the network architecture. By increasing the number of hidden layers, networks acquire the ability of progressively extracting higher level features from the raw input, which is vastly useful in e.g. computer vision. From a mathematical perspective, these networks allow more complex and nonlinear parameterizations, which again allows modeling of more complex approximations. Since the network becomes *deeper* when the number of hidden layers increases, methods adopting this architecture are coined *deep learning* (DL) methods.

Deep learning is essentially an extension to the field of ML and encapsulates ANNs that have multiple hidden layers. DL facilitate *feature learning*, meaning techniques that are able to identify representations from raw input data. Since DL methods are able to extract

such representations without manual feature engineering, these methods enable training with minimal human intervention. This concept is emphasized in Figure 2.4.

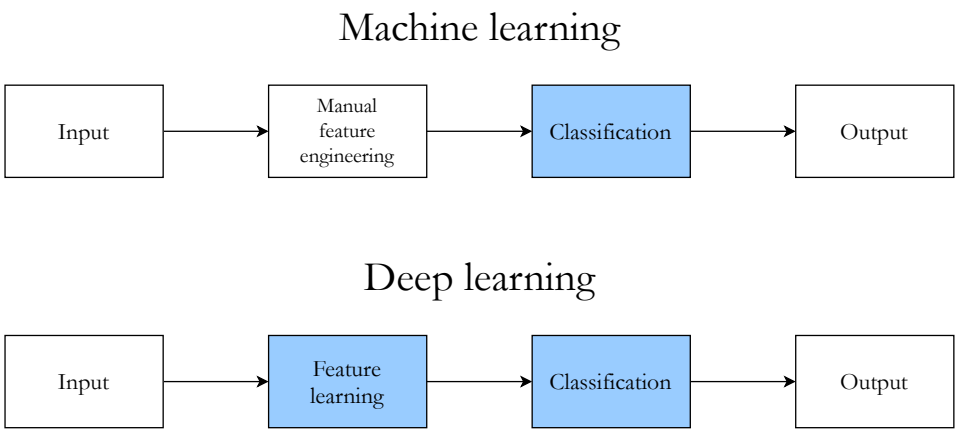



**Figure 2.4:** Chart conceptualizing the difference between classical ML methods and DL methods. The light blue processes indicate the location of where the agents learn, i.e. the position of the neural networks in the systems.

The main difference between ML and DL is therefore the process of extracting features. In classic ML methods this has to be done manually through human feature engineering. DL has, however, demonstrated that this can be executed automatically through the adaptation of additional layers, yielding feature learning. The drawback is that DL methods would require a notable increase in data to succeed.

The most straightforward DL network is the deep neural network (DNN) which is simply a feed-forward ANN with multiple hidden layers. DL also encapsulates methods tailor-made for specific tasks, one of which being convolutional neural networks (CNNs). CNNs are the common strategy for analyzing visual imagery. Using convolutional layers as hidden layers, these networks exploit hierarchical features in the data and construct complex patterns using simpler patterns, such as lines or circles. CNNs were first used as a means to classify handwritten numbers, separating a number into several simpler components to obtain the correct classification [45]. Recurrent neural networks (RNNs) branch towards analysis of temporal data, where the hidden layers are used as memory buffers to store internal states in order to interpret sequences of inputs. RNNs are vastly used in applications where the data is sequential, such as speech recognition [46].

### 2.1.3 Training DL models

From the theory presented thus far it is clear that neural networks possess great potential, concretized by the universal approximation theorem and displayed through recent results in the community. The main challenge of DL methods, and specifically DNNs, is to obtain the correct weights and biases, jointly referred to as the network parameters, in order to obtain a sufficiently accurate approximator for a given problem.

Given a dataset $\mathcal{D}$ consisting of $N$ input vectors with known target outputs,

$$\mathcal{D} = \{(\boldsymbol{x}_1,\ \boldsymbol{t}_1), (\boldsymbol{x}_2,\ \boldsymbol{t}_2),\ \dots\ , (\boldsymbol{x}_N,\ \boldsymbol{t}_N)\},$$

the error of input-output pair $i$ can be computed as the difference between the target and the computed, predicted output of the network, $\boldsymbol{e}_i = \boldsymbol{t}_i - \boldsymbol{y}_i(\boldsymbol{x}, \boldsymbol{\theta})$. Here, $\boldsymbol{\theta}$ expresses the the network parameters $\boldsymbol{\theta} = [w_{11}^1, b_1^1, w_{12}^1, \dots]^\intercal$.

The error is normally inputted into a loss function, $J(\boldsymbol{\theta})$, which serves as a metric expressing how far off the prediction $\boldsymbol{y}(\boldsymbol{x}_i, \boldsymbol{\theta})$[1] was to the target value $\boldsymbol{t}_i$. Intuitively, one wishes to minimize the loss by means of pushing the prediction towards the target. This, in turn, renders the search for the optimal network parameters an optimization problem, a field with rich literature.

One of the most popular computational optimization techniques is the *gradient descent* algorithm. By iteratively computing the gradient of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, one can calculate the contributing loss affiliated with each network parameter. Using this method the magnitude and direction one needs to change each parameter in order to minimize the loss can be found, yielding an update rule for the weights and biases that minimize the loss. Assuming that $J$ is differentiable with respect to $\boldsymbol{\theta}$, gradient descent states the update rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{2.5}$$

where $\alpha$ is the specified learning rate of the network. It is worth noting that there exist a manifold of optimizers for minimizing the loss, such as *stochastic gradient descent* and *Adam* [47]. Still, the majority of these stem from the concept of utilizing the loss gradient for updating parameters.

Since the output layer in a DNN is the only layer that has a desired target value, the loss can only be calculate at the output layer. After being computed, the loss must therefore be passed backwards in the network, from the output layer to the hidden layers, in order to convey the loss to these layers and provide the necessary information for calculating parameter changes for loss minimization. Due to the flow of the loss being *propagated* backwards in the network, this technique was coined *backpropagation* and was first introduced in [43, 46].

The backpropagation algorithm computes the gradient of the loss function with respect to each network parameter by utilizing the chain rule. This scheme computes the gradient layer by layer, iterating backward from the output layer. This ensures no redundant calculations of intermediate terms in the chain rule, which yields reduced run-time.

Using (2.3) recursively, one can express the output of the network, $\boldsymbol{y}$, with respect to the input $\boldsymbol{x}$. The result is a nested function with respect to the activations of each layer. Consider an arbitrary hidden layer operation $\boldsymbol{h}^l$ that encapsulates the weights, biases and activation as fol lows:

$$\boldsymbol{h}^l(\boldsymbol{a}) = f^l(\boldsymbol{W}^l \boldsymbol{a} + \boldsymbol{b}^l). \tag{2.6}$$

[1] Though $\boldsymbol{y}$ depends upon $\boldsymbol{x}$ and $\boldsymbol{\theta}$ this is omitted in the rest of the theory for simplicity.

Given a DNN with $L$ hidden layers the forward pass becomes:

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{h}^L(\boldsymbol{y}^{L-1}) \\ &= \boldsymbol{h}^L(\boldsymbol{h}^{L-1}(\dots(\boldsymbol{h}^2(\boldsymbol{h}^1(\boldsymbol{x}))))). \end{aligned} \tag{2.7}$$

Figure 2.5 illustrates that the forward pass results in computing the output and subsequently obtaining the loss. After the forward pass, the network will propagate the loss backwards in order to calculate $\nabla_{\boldsymbol{\theta}} J$ as the product of the gradients of each layer with respect to the layer's input.

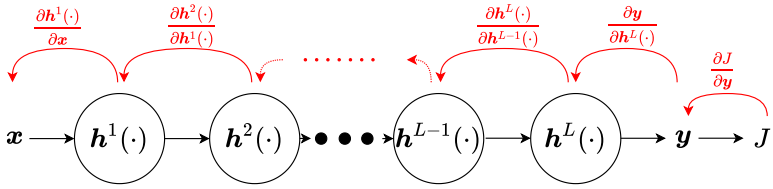



**Figure 2.5:** Computational graph illustrating the forward pass and the resulting backpropagation steps for computing the loss gradient.

Ultimately, training a DNN translates to computing

$$\nabla_{\boldsymbol{\theta}} J = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots\right]^\intercal \tag{2.8}$$

using the chain rule on the nested function described in (2.7) and perform a gradient descent update based on (2.5).

Training DNNs often pose several challenges due to the nature of supervised learning. Since the network is only fed with training data during the learning period, there is no real guarantee that the agent will perform well for test data that it has not previously experienced. This concept is referred to as *overfitting*; the network learns how to respond to examples it has seen before but refrains from learning the underlying concepts that the training data is based on. On the other hand, DNNs can also experience *underfitting*, meaning that the network struggles to capture the underlying structure of the examples. Furthermore, these networks may also be difficult to train with respect to stability, where small perturbations in input can result in substantial changes in output. This causes oscillatory behavior during learning and may result in a network not converging. Luckily, there are several proposed methods aiming to mitigate these challenges, such as dropout [48], weight initialization and regularization [49], early stopping [50], parametric noise injection [51], data augmentation [52] and batch normalization [53].

## 2.2 Reinforcement learning

As opposed to the classical ML approaches presented so far where the target value is given beforehand, reinforcement learning (RL) aims its attention at making an agent learn how

to behave in an environment where the only feedback it receives for its actions is a scalar reward signal. Generally, the agent's long-term goal is to execute actions at each discrete time step that maximize the sum of the rewards acquired throughout its lifetime. The RL problem therefore consists of the agent continuously interacting with the environment and gaining knowledge about how to optimize its behavior, solely based on the evaluative feedback it receives. This dynamics system is presented in the widely-accepted textbook of Sutton and Barto [54].

The RL problem is illustrated in Figure 2.6. At time step $t$ the agent receives a state representation of the environment $\boldsymbol{s}_t$ and chooses an action, $\boldsymbol{a}_t$. The agent receives a scalar reward $r_{t+1}$, and the environment evolves to state $\boldsymbol{s}_{t+1}$. For generality, the subsequent theoretical sections assume multidimensional states and actions.

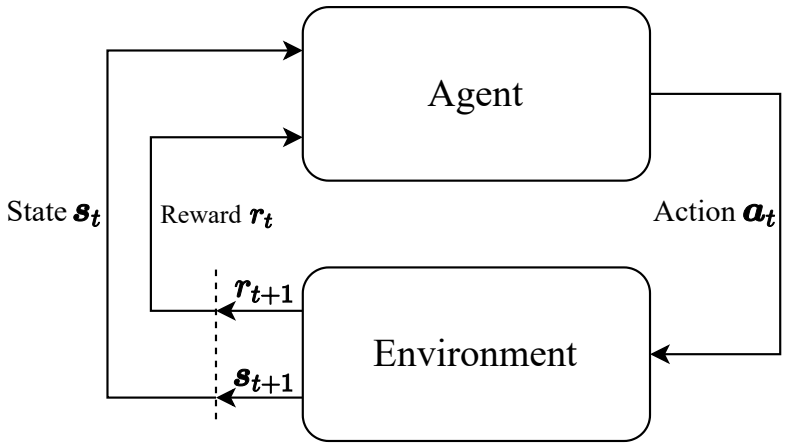



**Figure 2.6:** The feedback dynamics of the reinforcement learning problem.

The following sections, inspired by the work of Sutton and Barto [54], describe the integral theory behind the fundamental agent and lays the foundation for the more advanced RL methods presented later.

### 2.2.1 Markov decision processes

Dynamic programming (DP) constitute solving complex problems by dividing them into subproblems. The general RL problem is commonly modeled such that it fits the DP framework and it has therefore been found advantageous to formulate an RL process as a *Markov decision process* (MDP). An MDP is formally defined as a discrete time stochastic control framework, dedicated to address sequential decision-making problems in the face of uncertainty. MDPs facilitate the 4-tuple consisting of ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$), where each component serves its own purpose, explained as follows:

- $\mathcal{S}$ is a finite set of unique states available in the environment.
- $\mathcal{A}$ is a finite set of unique actions. The available actions for an agent often relies upon the state $\boldsymbol{s}$ the agent finds itself in, i.e. $\mathcal{A}(\boldsymbol{s})$.

- $\mathcal{P}$ is the transition function constructed by the transition probability $\mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) = \Pr(\boldsymbol{s}_{t+1} = \boldsymbol{s}'|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a})$. Here, $\boldsymbol{s}$ is the current state, $\boldsymbol{a}$ is the action chosen while in $\boldsymbol{s}$ and $\boldsymbol{s}'$ is the next state of the agent.
- $\mathcal{R}$ is the reward function containing the immediate rewards the agent receives after transitioning from state $\boldsymbol{s}$ to state $\boldsymbol{s}'$ due to action $\boldsymbol{a}$: $\mathcal{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') = \mathbb{E}(r_t \mid \boldsymbol{s}_{t-1} = \boldsymbol{s}, \boldsymbol{a}_{t-1} = \boldsymbol{a}, \boldsymbol{s}_t = \boldsymbol{s}')$.

The transition function and reward function collectively compose the *model* of the MDP. What makes MDPs so powerful in terms of modeling is the predication assumptions made regarding the state of the agent. A process satisfies the *Markov property* if one can predict the next state of the process solely based on its present state, without impairing the prediction quality. Mathematically, the Markovian dynamics is described as

$$\mathcal{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{s}_{t-1}, \dots, \boldsymbol{s}_0, \boldsymbol{a}_t, \boldsymbol{a}_{t-1}, \dots, \boldsymbol{a}_0) = \mathcal{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t). \tag{2.9}$$

The Markov property holds great value as this "memorylessness" characteristic heavily reduces the number of parameters required to construct the transition model of an arbitrary MDP. A general process with the Markov property can subsequently be described through the decision network shown in Figure 2.7, where the probability of evolving to $\boldsymbol{s}_{t+1}$ only depends on $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$, as (2.9) states.

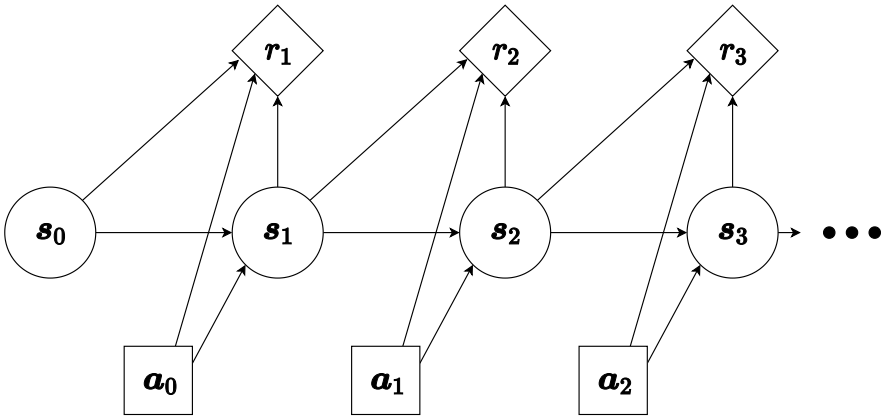



**Figure 2.7:** The decision network of an MDP with transition function $\mathcal{P}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$ and reward function $\mathcal{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$.

Though Figure 2.7 assumes that the transition function is on the form $\mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a})$ and the reward function is on the form $\mathcal{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}')$, the structure of the transition and reward function may vary, depending on the process. The reward function can for instance be dependent on the state the agent arrives at and the action that took it there or only of the state, i.e. $\mathcal{R}(\boldsymbol{s}',\boldsymbol{a})$ and $\mathcal{R}(\boldsymbol{s}')$ respectively. Given the model $(\mathcal{P},\mathcal{R})$, the dynamics of the problem and hence Figure 2.7 will change accordingly.

In certain settings the agent is not capable of observing its full state, which can impose challenges for a model. *Partially observable MDPs* (POMDPs) handle such cases, where

the agent experiences restrictions to state observations. POMDPs are defined as processes that only observe evidence that can yield useful information concerning the state, without being able to observe the state itself. The lack of full knowledge will understandably contribute to a more complex model of the process, albeit painting a more realistic picture of real world situations. Fortunately, POMDPs may be converted into MDPs where the inaccessibility of the full state is still taken into account. Through the introduction of a belief state that is deducted from the observations, the agent will have the ability to act optimally in a partially observable environment, purely based on the belief states.

### 2.2.2 Rewards

The reward function was introduced as an evaluative feedback in Section 2.2.1 which can be based on several parameters, depending on how the model is defined. The agent receives a reward at each time step based on a reward function designed to fit the agent's objective. The reward function can be considered as a tool to convey what the desired behavior of the agent is. This goal can be related to a wide selection of objectives. The reward function may for example praise an automotive agent for following a specific path or a marine vessel agent for holding its dynamic positioning setpoint.

The reward may also be negative, which is often considered in the literature as *cost* or a *penalty*. The disbursement of cost has the same logical approach as handing rewards, though the effect on the agent is negated. A marine vessel can be given a penalty for being far away from a target value, a self-driving car can be penalized for being offset from its desired path and a drone may experience cost for being far away from the landing platform. Regardless of considering rewards as a carrot or a stick, it can be agreed upon that the agent's performance greatly relies upon the reward function, and it is hence vital to know how to design it to obtain desired behavior. For convenience's sake, the theory beyond this point treats the reward as a positive feedback.

The agent seeks to maximize the total future reward over an episode, which is a subsequence restricted to a finite time horizon. This total reward, denoted $G_t$, is defined as some variety of combination of the reward sequence. In its simplest form, the total reward is just the sum of all registered rewards beyond $t$:

$$G_t := r_{t+1} + r_{t+2} + r_{t+3} + \ \dots \ + r_T, \tag{2.10}$$

where $r_t$ is the reward at time step $t$ and $T$ is the final time step, concluding the episode. Note that the notation is simplified by omitting the reward's dependence on states and actions introduced in Section 2.2.1. Though simplistic, the sum of future rewards is seldom used, and is replaced by the *discounted* total reward, namely

$$G_t := r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ \dots \ + \gamma^{N-1} r_{t+N} = \sum_{n=0}^{N} \gamma^n r_{t+n+1}, \tag{2.11}$$

where $0 \leq \gamma < 1$ is the *discount factor*. This value determines the current significance of future rewards: a reward received $k$ time steps in the future is worth $\gamma^{k-1}$ times what

it would be worth if it was received immediately. Given a discount factor of 0, the agent would only be concerned with maximizing the reward it receives at time $t$, rendering the agent short-sighted or *myopic*. As $\gamma$ approaches 1 the agent becomes more future oriented, due to the future rewards becoming less discounted. Although it can be positive for an agent to consider future rewards, choosing $\gamma$ is a choice of design to fit with respect to the agent's application. If immediate reward is of highest importance and future rewards are not, $\gamma$ can be set to a lower value to suppress the importance of future rewards. On the flip side, if an agent wants to achieve a long-term goal, the future rewards are decisive and, accordingly, $\gamma$ must be set higher.

From an analytic standpoint one can prove a relationship between returns in succeeding time steps. Given that $t < T$ and $G_T = 0$ the following relationship is derived:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \cdots) \\ &= r_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.12}$$

This relationship results in substantial simplifications from an algorithmic point of view. Computing returns can now be done efficiently from the reward sequences, fitting the DP framework well.

### 2.2.3 The Bellman equation

The total reward from an arbitrary state to the terminal state is closely related to the *value* of being in a specific state $\boldsymbol{s}$, $V(\boldsymbol{s})$. The subtle, though crucial, difference between $G$ and $V$ is that the former is a measured amount of the *actual* total reward received in a specific sequence, while the latter is the total reward the agent *expects* to receive when following an action sequence from $\boldsymbol{s}$ onward. Formulated in another way, given an action sequence policy $\pi$ that maps a state $\boldsymbol{s}$ onto an action $\boldsymbol{a}$, $V^\pi(\boldsymbol{s})$ is the *expected* total reward, or the value, when the agent starts in $\boldsymbol{s}$ and follows $\pi$ thereafter. This can be described mathematically as

$$V^\pi(\boldsymbol{s}) = \mathbb{E}_\pi \left\{ G_t \mid \boldsymbol{s}_t = \boldsymbol{s} \right\}. \tag{2.13}$$

In essence, a value function serves as a metric of how beneficial it is for the agent to be in a certain state. The reason for using value functions over total rewards is that the former form convenient tools for linking an optimality criteria to deriving action sequence policies. Many learning algorithms for MDPs compute optimal policies through learning value functions, for example *value iteration* [54].

Following a similar train of thought, the value of an action $\boldsymbol{a}$ in state $\boldsymbol{s}$ and following policy $\pi$ thereafter is named the state-action value function, often referred to as the Q-value or the Q-function:

$$Q^\pi(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}_\pi \left\{ G_t \middle| \, \boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{a}_t = \boldsymbol{a} \right\}. \tag{2.14}$$

The utility of value and state-action value functions is that their values can be estimated without the agent having a model of the environment. These quantities can be learned

from experience, normally through trial-and-error approaches. A fundamental property these functions satisfy is the recursive relationships between successive states, similar to what was discussed in Section 2.2.2. With the base in (2.13) one can derive the following relationship with a policy $\pi$:

$$\begin{aligned} V^{\pi}(\boldsymbol{s}) &= \mathbb{E}_{\pi}\{G_t \,|\, \boldsymbol{s}_t = \boldsymbol{s}\} \\ &= \mathbb{E}_{\pi}\{r_{t+1} + \gamma G_{t+1} \,|\, \boldsymbol{s}_t = \boldsymbol{s}\} \qquad \text{(by (2.12))} \\ &= \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \big[\mathcal{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') + \gamma \mathbb{E}_{\pi}\{G_{t+1}|\boldsymbol{s}_{t+1} = \boldsymbol{s}'\}\big] \\ &= \sum_{\boldsymbol{a}} \pi(\boldsymbol{a}|\boldsymbol{s}) \sum_{\boldsymbol{s}'} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) \big[\mathcal{R}(\boldsymbol{s},\boldsymbol{a},\boldsymbol{s}') + \gamma V^{\pi}(\boldsymbol{s}')\big], \quad \forall\, \boldsymbol{s} \in \mathcal{S}. \end{aligned} \tag{2.15}$$

This is the *Bellman equation* for $V^{\pi}$. It declares a relationship between the value between successive states and can be viewed as a way to forecast the value recursively. Equation (2.15) states that the value of $\boldsymbol{s}$ must be equal to the discounted value of the expected next state $\boldsymbol{s}'$ plus the reward expected for going from $\boldsymbol{s}$ to $\boldsymbol{s}'$ due to action $\boldsymbol{a}$. This can be described as a one-stage look-ahead into all possible future states, where this operation will transfer information about the successive states $\boldsymbol{s}'$ back to the current state $\boldsymbol{s}$. Equation (2.15) will perform a one-stage look-ahead in order to backpropagate information about successive state-action pairs to preceding state-action pairs. The equation states that an agent is able to learn from every state transition that the agent experiences, and not only from a full episode.

Equation (2.15) can be considered a powerful result, granting several RL methods the ability to compute, approximate and learn $V^{\pi}$. In fact, the result can be enhanced further in virtue of *Bellman's principle of optimality*, defined by Richard Bellman himself [55]:

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

This principle suggests that an optimal policy can be assembled in a sequential fashion, where the decision problem can be divided into subproblems and the optimal action in any subproblem will also be the optimal action for the full-scale problem. As this result is one of the cornerstones of DP, the Bellman equation often is referred to as the dynamic programming equation.

Bellman's principle of optimality is considered the foundation for deriving a solution for maximizing rewards in sequential decision problems, which leads to Bellman's optimality equations. RL problems generally aim to maximize the reward received over time. This reward depends on the policy the agent follows. In the sense of returned reward, a policy $\pi$ can be deemed better than or equally good as a different policy $\pi'$ if the expected return of the former is greater than or equal to the latter. An optimal policy, denoted $\pi^*$, will by definition have greater or equal state value than any other policy $\pi$ regardless of the state, namely

$$V^{\pi^*}(\boldsymbol{s}) \geq V^{\pi}(\boldsymbol{s}),\ \forall \boldsymbol{s} \in \mathcal{S}.$$

An optimal policy holds the following property:

$$V^{\pi^*}(\boldsymbol{s}) := \max_{\pi} V^{\pi}(\boldsymbol{s}).$$

Further, it can be shown that the optimal solution $V^* = V^{\pi^*}$ satisfies

$$V^*(\boldsymbol{s}) = \max_{\boldsymbol{a} \in \mathcal{A}(\boldsymbol{s})} \sum_{\boldsymbol{s}' \in \mathcal{S}} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma V^*(\boldsymbol{s}')),$$

constituting the Bellman optimality equation for the state value. Extracting optimal actions given the optimal state value function $V^*$ can be done by the following relationship:

$$\pi^*(\boldsymbol{s}) = \operatorname*{argmax}_{\boldsymbol{a} \in \mathcal{A}(\boldsymbol{s})} \sum_{\boldsymbol{s}' \in \mathcal{S}} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma V^*(\boldsymbol{s}')). \tag{2.16}$$

Equation (2.16) states the equation for a *greedy policy*, meaning a policy where the agent always chooses the action that maximizes expected return. An analogous result to the Bellman optimality equation for the state value can be derived for the Q-value:

$$Q^*(\boldsymbol{s}, \boldsymbol{a}) = \sum_{\boldsymbol{s}' \in \mathcal{S}} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma \max_{\boldsymbol{a}'} Q^*(\boldsymbol{s}', \boldsymbol{a}')).$$

The relationship between $V^*$ and $Q^*$ is given as:

$$\begin{aligned} Q^*(\boldsymbol{s}, \boldsymbol{a}) &= \sum_{\boldsymbol{s}' \in \mathcal{S}} \mathcal{P}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma V^*(\boldsymbol{s}')), \\ V^*(\boldsymbol{s}) &= \max_{\boldsymbol{a}} Q^*(\boldsymbol{s}, \boldsymbol{a}). \end{aligned}$$

Thus, (2.16) can be rewritten using the state-action value:

$$\pi^*(\boldsymbol{s}) = \operatorname*{argmax}_{\boldsymbol{a}} Q^*(\boldsymbol{s}, \boldsymbol{a}). \tag{2.17}$$

Equation (2.17) identifies the optimal action for a state, $\pi^*(\boldsymbol{s})$, as the action that yields the highest expected long-term reward resulting from executing action $\boldsymbol{a}$ in state $\boldsymbol{s}$.

Q-functions are useful when implementing algorithms in RL due to the fact that they make the weighted summation over different actions. These functions do not require forward-reasoning steps in order to compute optimal actions in states, which is the leading reason as to why Q-functions are preferred over value functions in model-free RL approaches.

### 2.2.4 Temporal difference learning

As discussed in Section 2.2.3, the Bellman equation forms the basis for solving sequential decision problems through DP. Though the theory discussed is well-defined and robust, it is not always feasible to implement these methods in real-life applications. DP problems require knowledge of the model of the system, $\mathcal{P}$ and $\mathcal{R}$, which is unlikely more often than not. One can argue that RL methods excel for obtaining optimal policies under uncertainties, for instance when the model of the system is not known. These methods focus

on estimating various quantities in the system. This may include estimating the model and subsequently use DP methods, or directly estimate $V$ or $Q$ without the intermediate step of model estimation. These two approaches are called indirect and direct RL, respectively.

A general underlying mechanism for model-free methods is *temporal difference learning* (TD learning), a technique utilizing Bellman's principle of optimality. TD learning allows the agent to adjust its estimate of a value every time it obtains information about in-between steps. The agent employs *bootstrapping*, meaning it estimates values based on other estimates. Each step the agent takes generates a learning example which can be utilized to bring additional information for the estimation of $V$ or $Q$. This improves the agent's learning efficiency as it no longer has to wait until the end of an episode to update its estimates, as opposed to e.g. Monte Carlo methods [54]. Furthermore, only the states that are visited will be updated, implying no redundant update computations. The simplest TD method, coined $TD(0)$, executes an update of the form

$$V(\boldsymbol{s}) \leftarrow V(\boldsymbol{s}) + \alpha \underbrace{(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma V(\boldsymbol{s}') - V(\boldsymbol{s}))}_{\text{TD error}}, \tag{2.18}$$

where $0 < \alpha < 1$ is the *learning rate* of the agent, indicating how much of the newly measured information is absorbed to the agent's new value estimate. The *TD error* highlighted in (2.18) is a measure of the estimate error at the time of the update. Evidently, the backup operation is performed directly after experiencing the transition from $\boldsymbol{s}$ to $\boldsymbol{s}'$ based on action $\boldsymbol{a}$, while receiving reward $\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')$. Analogously, one can derive the $TD(0)$ update for a state-action value, given that one has the additional information of the next action $\boldsymbol{a}'$ in the next state:

$$Q(\boldsymbol{s}, \boldsymbol{a}) \leftarrow Q(\boldsymbol{s}, \boldsymbol{a}) + \alpha \underbrace{(\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma Q(\boldsymbol{s}', \boldsymbol{a}') - Q(\boldsymbol{s}, \boldsymbol{a}))}_{\text{TD error}}. \tag{2.19}$$

The underlying calculations conclude that if the TD error is positive, the agent's tendency to select $a$ should be strengthened for the future. Likewise, if the TD error is negative, it suggests that the tendency should be weakened. $TD(0)$ portrays a scheme where the agent bases its new estimate on a weighted sum of old estimate and the new measurements.

The general $TD(0)$ algorithm has shown that $V$ is assured convergence to $V^*$ and the policy $\pi$ converges towards $\pi^*$ when the following constraints are met [56]:

- The values of $V$ are stored in a lookup table.
- The learning rates satisfy:
  - $\alpha_t(s_t) \in [0, 1]$.
  - $\sum_t \alpha_t(s_t) = \infty$.
  - $\sum_t (\alpha_t(s_t))^2 < \infty$.
  - $\alpha_t(s) = 0$ unless $s = s_t$.
- $\forall \boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}'$: $\text{Var}\{\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}')\} < \infty$.

There have been multiple algorithms that have sprung from the TD learning principles, among these being *Q-learning* and *SARSA* [54]. For their designated purpose these methods, as most TD methods, will be sufficient for problems that do not have a very high level of complexity, including a small state-action space. However, for more involved problems that require handling continuous behavior, such as control in robotics, these solutions yield suboptimal behavior due to the discretization of the state-action space. When this space increases the memory and computational power required to compute the values for all combinations inevitably becomes infeasible.

## 2.3 Deep reinforcement learning

Algorithms such as Q-learning and SARSA are so-called *tabular* methods, meaning they represent value functions, Q-functions and policies using tables with states or state-action pairs as entries. But, as briefly mentioned in Section 2.2.4, this scheme is only feasible when the state-action space is sufficiently small. As problems develop in complexity and the state-action space increases, tabular techniques will require immoderate amounts of computational power to extract solutions. This problem was recognized by Bellman as the *curse of dimensionality*.

To counteract this the question arose whether or not it was possible for the agent to generalize its experiences given a limited subset of the state-action space, and hence extract a wholesome solution for the entire space based on this. In the wake of this, function approximators were extensively studied to attempt to accomplish exactly this.

Using the methodologies presented in Section 2.1, the birth of *deep reinforcement learning* (DRL) was a fact. In general terms, DRL methods combine the methods of deep learning with reinforcement learning to create efficient algorithms capable of scaling previously unsolvable problems. As computational power has increased in recent years the use of ANNs as function approximators have proven to yield several DRL systems the ability to generalize and extract approximations of value functions and policies successfully. These schemes have resulted in a manifold of algorithms, one of which being *Deep Deterministic Policy Gradient* (DDPG). DDPG utilize ANNs to approximate $Q$ and $\pi$, effectively eliminating the need to store all state-action pairs in tables as this space increases. DDPG is presented more thoroughly in Section 2.3.3.

DRL methods are commonly divided into three subcategories: actor-only methods, critic-only methods and actor-critic methods. The division is motivated by the the architecture of the solutions, and what they aim to approximate. Actor-only methods aim to extract the policy $\pi(\boldsymbol{s})$, where an *actor* controls the actions to be executed. The second consists of learning only the value function $V(\boldsymbol{s})$ or Q-function $Q(\boldsymbol{s}, \boldsymbol{a})$, where the critic measures how good the chosen action is in a given state. Actor-critic methods aim at combining the strong points of actor-only and critic-only methods by learning both the policy and the value function.

Since critic-only methods exclusively approximate the value function, one will need to iterate over every action available in a state in order to find the optimal action. This renders the critic-only method inept in environments with continuous action spaces. On the other

hand, actor-only methods and actor-critic methods both have the ability to learn policies with continuous action spaces, which generally is an advantageous trait when designing control systems.

### 2.3.1 Policy gradient methods

The methods discussed so far have been based on first learning a value function and subsequently derive the policy based on the result. This means that the policies would not even exist without the value estimates. Policy gradient methods (PGMs) are different in this regard. PGMs mainly seek to conquer the aforementioned issues that arise for tabular method in relation to required memory and computational power. The simplest PGM aims to learn a *parameterized policy* able to directly approximate the optimal policy without the intermediate step of estimating value functions. This effectively eliminates the need to estimate the value function for action selection. Additionally, earlier restriction to discrete action spaces are discarded since this framework allow learning stochastic policies, which can output probability distributions over actions instead of deterministic actions.

These methods are based on optimizing a parameterized policy function with respect to the expected cumulative reward, and find the optimal policy function $\pi^*(\boldsymbol{s})$ based on this. This is achieved by utilizing gradient ascent to iteratively improve the policy with respect to maximizing the expected return. PGMs are generally on-policy, though many policy gradient based algorithms have made adjustments to make them off-policy.

Parameterized by parameter vector $\boldsymbol{\theta} = [\theta_1, \theta_2, \dots]^\intercal$, the policy is considered a probability of executing an action $\boldsymbol{a}$ at time $t$, assuming that the environment is in state $\boldsymbol{s}$ at time $t$. This can be expressed mathematically as

$$\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}) = \pi(\boldsymbol{a}|\boldsymbol{s}, \boldsymbol{\theta}) = \Pr(\boldsymbol{a}_t = \boldsymbol{a}|\boldsymbol{s}_t = \boldsymbol{s}, \boldsymbol{\theta}_t = \boldsymbol{\theta}). \tag{2.20}$$

The target is to learn the policy parameters based on the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter vector. As stated previously, the objective in RL is generally to maximize the cumulative reward, which means that the performance measure $J$ is often stated as

$$J(\boldsymbol{\theta}) = \mathbb{E}\left\{\sum_t \gamma^t \mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t)\right\}\Bigg|_{\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \tag{2.21}$$

where $\boldsymbol{a}_t$ is the action at $t$ sampled from the policy $\pi_{\boldsymbol{\theta}}$ and $\boldsymbol{s}_t$ is the state at $t$. Equation (2.21) assumes that the reward function does not consider the state following $\boldsymbol{s}_t$. Note that, in contrast to previous derivations, we now wish to *maximize* the performance measure $J$. As this performance measure is sought to be maximized with respect to $\boldsymbol{\theta}$ the optimization problem may be stated as

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathbb{E}\left\{\sum_t \gamma^t \mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t)\right\}\Bigg|_{\boldsymbol{a}_t \sim \pi_{\boldsymbol{\theta}}} \tag{2.22}$$

where $\boldsymbol{\theta}^*$ is the optimal policy parameter vector. Since the performance measure is to be *maximized*, one can obtain $\boldsymbol{\theta}^*$ by performing gradient *ascent* in $J$, which takes the update

form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{2.23}$$

Here, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is a stochastic estimate whose expectation approximates the gradient of $J$ with respect to $\boldsymbol{\theta}$.

The policy gradient scheme offers a big improvement considering it allows not only continuous state space, but also continuous action space. Comparing to critic-only methods where one would have to iterate through all possible state-action values in order to find the optimal action in a state, this exhaustive search is not necessary for PGMs. This is a desirable attribute, especially for continuous control applications.

It is worth mentioning that all schemes where the learning phase of the policy is gradient-based are classified as PGMs, regardless if they are actor-only or actor-critic methods.

There are various methods that can be adopted when calculating the gradient of the objective function $J$, two of which being the method of *finite difference approximation* and *direct policy differentiation*.

#### 2.3.1.1 Finite difference gradient approximation

One of the more established approaches for calculating the policy gradient is given by using a finite difference. By perturbing the network parameter $i$ by a small value $\varepsilon$, an approximation of the gradient in the $i$th dimension can be found. The forward difference method is expressed as:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{J(\boldsymbol{\theta} + \varepsilon \vec{e}_i) - J(\boldsymbol{\theta})}{\varepsilon}, \tag{2.24}$$

where $\vec{e}_i$ is a unit vector directed along the positive $i$ axis with the same dimension as $\boldsymbol{\theta}$.

Similarly, one can use the backward difference,

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{J(\boldsymbol{\theta}) - J(\boldsymbol{\theta} - \varepsilon \vec{e}_i)}{\varepsilon}, \tag{2.25}$$

or the central difference,

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{J(\boldsymbol{\theta} + \varepsilon \vec{e}_i) - J(\boldsymbol{\theta} - \varepsilon \vec{e}_i)}{2\varepsilon}. \tag{2.26}$$

From these approaches, the gradient of $J$ with respect to an $n$-dimensional parameter vector $\boldsymbol{\theta}$, can be calculated

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \Big[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_2}, \cdots, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n}\Big]^{\intercal}, \tag{2.27}$$

as presented in Section 2.1.3.

The two former methods use $n$ evaluation steps to acquire the gradient of $J$, while the latter use $2n$. Despite the fact that the finite difference approaches are able to compute the gradients of non-differentiable functions, they are ultimately are rendered inefficient and easily affected by noise.

#### 2.3.1.2 Direct policy differentiation

A more modern approach with higher accuracy than the finite difference method is the direct policy differentiation. Most neural nets in today's APIs are designed by constructing underlying computational graphs symbolizing the signal flow in the network. By using the chain rule and automatic differentiation on these graphs, gradients can be found easier and more efficiently.

Assuming that the policy gradient $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a|s)$ is known, the following identity holds:

$$\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}) = \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})} = \pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s})\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\boldsymbol{a}|\boldsymbol{s}). \tag{2.28}$$

Further calculations, including rewriting the terms for $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ and $\pi_{\boldsymbol{\theta}}(a|s)$, allow (2.21) to be augmented to

$$\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) = \mathbb{E}\left\{\sum_{t=1}^{T}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t)\left(\sum_{t=1}^{T}\gamma^t\mathcal{R}(\boldsymbol{s}_t,\boldsymbol{a}_t)\right)\right\}, \tag{2.29}$$

where $T$ is the number of steps in the episode.

Further, the expected value of the gradients can be approximated as the average of all gradients over an entire simulation with $N$ episodes. Resultantly, the policy gradient is calculated as:

$$\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{i,t}\,|\,\boldsymbol{s}_{i,t})\left(\sum_{t=1}^{T}\gamma^t\mathcal{R}(\boldsymbol{s}_{i,t},\boldsymbol{a}_{i,t})\right). \tag{2.30}$$

Analyzing (2.30), one can see that the gradient of $J$ is given as a product of the reward and the gradient of the probability of taking the action leading to that specific reward. Consequently, during gradient ascent, an action resulting in a higher reward will cause the network parameters to be nudged in the direction that will render that action more probable. Note that the nudge is proportional to the reward, meaning a higher reward will tweak the parameters more aggressively than a small reward. In this way the policy is altered in the direction advocating higher rewards, which is the ultimate goal of an RL agent.

By nature, the actions chosen by an agent in a policy gradient scheme will introduce high variance, which is known to hurt deep learning optimization. Mitigating high variances often results in faster and more stable policy learning. Fortunately, this can be carried out through *baselines*.

To reduce the variance caused by the actions, the variance for the sampled rewards are subject to reduction. A constant baseline will be unbiased in expectation and can hence be subtracted from the accumulated rewards. This yields reduced variance in the estimator's calculations without changing the expectation itself. When including a baseline to the

computations, the loss gradient is altered to the following form:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \left( \sum_{t=1}^{T} \gamma^t \mathcal{R}(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) - b \right), \tag{2.31}$$

where $b$ is the baseline.

There are several options when choosing a baseline for variance reduction. There is typically a trade-off between complexity and optimality. The simplest baseline is the

We also wish for our computations to inhabit *causality*, meaning that future actions do not alter previous rewards. In other words, the policy at $t'$ cannot affect the reward at $t$ if $t < t'$. The lower bound of the last sum in (2.31) is therefore adopted to inhabit causality:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \left( \sum_{t'=t}^{T} \gamma^{t'-t} \mathcal{R}(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'}) - b \right), \tag{2.32}$$

One of the simpler baseline is given as the average reward, namely

$$b = \sum_{i=1}^{N} \sum_{t=1}^{T} \gamma^t \mathcal{R}(\boldsymbol{s}_{i,t'}, \boldsymbol{a}_{i,t'}). \tag{2.33}$$

Although it is straightforward to implement, it has higher variance than its alternatives. The optimal baseline seen from a mathematical standpoint is

$$b = \frac{\sum_{i=1}^{N} \left( \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \right)^2 \left( \sum_{t=1}^{T} \gamma^t \mathcal{R}(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \right)}{\left( \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \right)^2}, \tag{2.34}$$

which is the expected reward weighted by gradient magnitudes.

The most common choice, however, is the on-policy value function $V^{\pi}(\boldsymbol{s}_t)$, i.e.

$$b(\boldsymbol{s}_t) = V^{\pi}(\boldsymbol{s}_t). \tag{2.35}$$

This value constitutes an agent's average return when starting in $\boldsymbol{s}_t$ and acts according to $\pi$ thereafter. It has been proven empirically that this specific baseline reduces variance while encoding faster and more stable policy learning. Additionally, it serves as an intuitive baseline in the sense that if the agent receives the return it expects to receive, the agent should perceive the result of the episode as a "neutral" score.

The scheme presented lays the foundation for the REINFORCE algorithm, whose pseudo code is given in Algorithm 1 [54].

### 2.3.2 Actor-critic methods

The preceding section discussed the simplest form of a PGM, aiming to approximate an agent's policy without the use of a value function, deeming it an actor-only method.

**Algorithm 1** REINFORCE

Initialize $\boldsymbol{\theta}$ arbitrarily and choose a baseline.
**for** each episode $= 1 : M$ **do**
    Generate trajectory using the current policy:
    $\{\boldsymbol{s}_1, \boldsymbol{a}_1, r_2, \boldsymbol{s}_2, \boldsymbol{a}_2, r_3, \ldots, \boldsymbol{s}_{T-1}, \boldsymbol{a}_{T-1}, r_T\} \sim \pi_{\boldsymbol{\theta}}$
    **for** $t = 1 : T$ **do**
        Compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by using (2.32)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
return $\boldsymbol{\theta}$

Though an agent seldom requires anything more than a policy in order to act, actor-critic methods exploit a clever trick. By incorporating both policy and value functions in the learning process, the estimate of the former can help improve the estimate of the latter, and vice versa. Since actor-critic schemes have the ability to learn both which actions to take and analyze how good these actions were, these methods are often superior to actor-only methods when designing agents able to handle continuous problems. Additionally, actor-critic methods hold the promise of delivering faster convergence compared to actor-only methods, due to variance reduction.

As the name may suggest, actor-critic methods are based on two components: the actor and the critic. These methods adopt many of the traits from policy gradients while also including additional functionality. The actor takes the state as input and outputs a probability distribution over all actions, effectively controlling how the agent behaves by learning the optimal policy. Correspondingly, the actor follows a policy based model. The critic, on the other hand, conducts value-based operations by evaluating the agent's state and action by computing the value function or Q-function. The actor and critic participate in a back-and-forth motion where one entity receives feedback from the other in order to improve its own performance. The dynamics between the actor, critic and the environment is illustrated in Figure 2.8.

Both the actor and the critic are normally based on using policy gradients and value function approximation through ANNs. The methodology shown in Section 2.3.1 is adopted, where the actor policy function $\pi_{\boldsymbol{\theta}_a}$, which often is stochastic in these methods, is parameterized by the parameter vector $\boldsymbol{\theta}_a$. In a similar fashion the critic value function $V_{\boldsymbol{\theta}_c}$ or $Q_{\boldsymbol{\theta}_c}$ is parameterized by the parameter vector $\boldsymbol{\theta}_c$. Though $V_{\boldsymbol{\theta}_c}$ is used in the coming derivations, note that the critic can support both $V_{\boldsymbol{\theta}_c}$ and $Q_{\boldsymbol{\theta}_c}$.

The training of the two networks is performed in parallel. Through the use of gradient ascent, a local maximum can be found in order to update the parameters of each network. As time passes, the actor learns to produce better and better actions, i.e. it is starting to learn a desired policy, and the critic is improving at evaluating the actions that the actor is taking. Opposed to policy gradient methods, the update of the weights does not happen at the end of the episode but rather at each step, inspired by TD learning discussed in Section 2.2.4.

Since actor-critic methods rely on the actor achieving the optimal policy by learning from

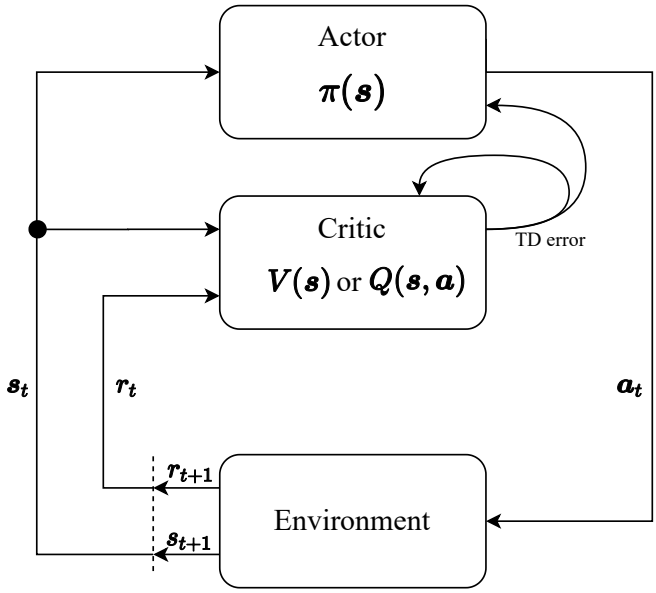



**Figure 2.8:** The standard architecture of an actor-critic method.

the critic, the latter entity calls for some evaluation measure for the policy. The TD error, introduced in (2.19), namely

$$\delta = \mathcal{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}') + \gamma V_{\boldsymbol{\theta}_c}(\boldsymbol{s}') - V_{\boldsymbol{\theta}_c}(\boldsymbol{s}), \tag{2.36}$$

proves to be a sensible choice for updating the critic, as the error should converge to 0 since the estimated value function approximates the true value function. Defining a critic loss function of the form

$$J_c(\boldsymbol{\theta}_c) = \frac{1}{2}\delta^2 \tag{2.37}$$

allows an update rule for the critic network that is adopted by the gradient descent scheme given in (2.5). Assuming that the value function estimate of the next state in the TD error is fixed and independent of $\boldsymbol{\theta}_c$, computing the gradient of (2.37) with respect to the parameter vector yields

$$\nabla_{\boldsymbol{\theta}_c} J_c(\boldsymbol{\theta}_c) = \delta \nabla_{\boldsymbol{\theta}_c} \delta = -\delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(\boldsymbol{s}). \tag{2.38}$$

Inserting this into the gradient descent formula yields the update rule for the critic, stated as

$$\boldsymbol{\theta}_c \leftarrow \boldsymbol{\theta}_c + \alpha_c \delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(\boldsymbol{s}), \tag{2.39}$$

where $\alpha_c$ is the learning rate of the critic.

For the case of the actor we wish to maximize the cumulative reward of the policy, see (2.21). Given an episodic update, i.e. an update when the batch size is 1, it can be shown that the policy gradient can be approximated in the following fashion:

$$\nabla_{\boldsymbol{\theta}_a} J(\boldsymbol{\theta}_a) \approx \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(\boldsymbol{s}). \tag{2.40}$$

Now, we use the gradient ascent scheme to find that the actor's update rule

$$\boldsymbol{\theta}_a \leftarrow \boldsymbol{\theta}_a + \alpha_a \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(\boldsymbol{s}), \tag{2.41}$$

where $\alpha_a$ is the actor's learning rate. As stated in Section 2.2.4 positive TD errors indicate that an action taken being a good one with respect to the value function, and the tendency to select this action again should be strengthened. Analogously, if the TD error is negative the tendency should be weakened. That is to say if $\delta$ is positive at time $t$, the action taken at $t$ performs better than the policy and the policy should adapt accordingly. Hence, by multiplying the gradient with $\delta$, the descent of $\boldsymbol{\theta}_a$ is *guided*, meaning that $\boldsymbol{\theta}_a$ is tweaked towards actions which give positive TD errors.

The pseudo code for the episodic actor-critic algorithm is given in Algorithm 2 [54].

**Algorithm 2** Episodic actor-critic

Initialize learning rates $\alpha_a$ and $\alpha_c$.
Randomly initialize the actor and critic with weights $\boldsymbol{\theta}_a$ and $\boldsymbol{\theta}_c$, respectively.
**while** training not finished **do**
    Receive initial observation state $\boldsymbol{s}$
    **while** $s$ not terminal **do**
        Sample action $\boldsymbol{a} \sim \pi(\boldsymbol{a}|\boldsymbol{s})$
        Execute $\boldsymbol{a}$ and observe reward $r$ and $\boldsymbol{s}'$
        Compute $\delta \leftarrow r + \gamma V_{\boldsymbol{\theta}_c}(\boldsymbol{s}') - V_{\boldsymbol{\theta}_c}(\boldsymbol{s})$ (if $\boldsymbol{s}'$ is terminal, then $V_{\boldsymbol{\theta}_c}(\boldsymbol{s}') = 0$)
        Update critic: $\boldsymbol{\theta}_c \leftarrow \boldsymbol{\theta}_c + \alpha_c \delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(\boldsymbol{s})$
        Update actor: $\boldsymbol{\theta}_a \leftarrow \boldsymbol{\theta}_a + \alpha_a \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(\boldsymbol{s})$
        Propagate state: $\boldsymbol{s} \leftarrow \boldsymbol{s}'$

### 2.3.3 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient, denoted DDPG, is a relatively novel DRL approach for continuous action control [30]. The theory behind DDPG emerges from deterministic policy gradients, a paper published by Silver et al. in 2014 [57]. This algorithm concurrently learns a policy in addition to a Q-function rather than a value function, meaning DDPG adopts principles from both critic-only and actor-critic frameworks. DDPG does this as a means of facilitating generalized solutions for tasks that extend to both continuous state and action space. This is done by learning a policy that *deterministically* maps states to specific actions using DNNs, hence the algorithm's name.

DDPG uses the Bellman equation on off-policy data to learn both the state-action function and the policy. By learning the state-action function instead of the value function the

critic is allowed to learn the value of every action in all states. Since the algorithm learns the optimal policy with off-policy data, DDPG is categorized as an off-policy actor-critic method.

As a DRL method, DDPG aims to mitigate the problem of requiring discrete and finite action spaces. In more simplistic algorithms such as Q-learning, the optimal action is found by iterating over all the Q-values and choosing the action that maximizes this quantity. When the action space is continuous, this search would be exhaustive, and solving the optimization problem would be highly non-trivial. Calculating $\max_a Q^*(\boldsymbol{s}, \boldsymbol{a})$ would therefore be a computationally expensive routine and given that this must be done every time the agent wants to execute an action, this scheme is rendered infeasible. Since the action space is continuous one can assume that the state-action function $Q^*(\boldsymbol{s}, \boldsymbol{a})$ is differentiable with respect to $\boldsymbol{a}$. This opens up the possibility to use gradient-based theory to obtain a policy $\mu(\boldsymbol{s})$ exploiting this trait. As a result, instead of executing the expensive routine of iterating through $Q^*(\boldsymbol{s}, \boldsymbol{a})$ to find the optimal action, DDPG approximates the optimal action as

$$\max_{\boldsymbol{a}} Q^*(\boldsymbol{s}, \boldsymbol{a}) \approx Q(\boldsymbol{s}, \mu(\boldsymbol{s})). \tag{2.42}$$

It is noted that though the published paper of DDPG includes several flexibility measures for modeling, such as modeling transitions through a stochastic process $\beta$ and considering the visitation of states for a policy $\pi$ as a distribution $\rho^\pi$, these elements are not included in the following derivations, which simplifies the notation.

Given an actor $\mu(\boldsymbol{s}|\boldsymbol{\theta}^\mu)$ and a critic $Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^Q)$, respectively parameterized by $\boldsymbol{\theta}^\mu$ and $\boldsymbol{\theta}^Q$ the loss function can defined as the squared error between the target and the prediction, i.e.

$$J(\boldsymbol{\theta}^Q) = \Big(y_t - Q(\boldsymbol{s}_t, \boldsymbol{a}_t|\boldsymbol{\theta}^Q)\Big)^2, \tag{2.43}$$

where the target is

$$y_t = r_t + \gamma Q(\boldsymbol{s}_{t+1}, \mu(\boldsymbol{s}_{t+1}|\boldsymbol{\theta}^Q)). \tag{2.44}$$

Note that the resulting error to be minimized in (2.43) is the TD error for a state-action function, similar to the one presented in Section 2.3.2.

The critic is learned through the Bellman equation, similar to e.g. Q-learning. The actor, on the other hand, must be updated by applying the chain rule to the expected return with respect to the actor parameters:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}^\mu} J &\approx \nabla_{\boldsymbol{\theta}^\mu} Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^Q), \\ &= \nabla_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^Q) \nabla_{\boldsymbol{\theta}^\mu} \mu(\boldsymbol{s}|\boldsymbol{\theta}^\mu). \end{aligned}$$

In [57] this result is denoted the deterministic policy gradient, which can be viewed as the gradient of the policy's performance.

DDPG adopts several techniques as a means of improving learning stability and allow more efficient learning. One of such improvements was the replay buffer, allowing independent and identically distributed (i.i.d.) inputs to the networks during training. Further,

target networks for both the critic and the actor were added. As opposed to popular techniques such as *Deep Q-Networks* [25], where target network parameters hold their values constant for a fixed amount of intervals before being updated, DDPG employs a subtle difference. *Soft* target updates to the target networks through Polyak averaging [58] has proven to improve learning stability.

The critic's target network $Q'$ is parameterized by $\boldsymbol{\theta}^{Q'}$ while the target network of the actor $\mu'$ is parameterized by $\boldsymbol{\theta}^{\mu'}$. Consequently, the update of these networks can be expressed in the following manner:

$$\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^{Q} + (1-\tau)\boldsymbol{\theta}^{Q'}$$
$$\boldsymbol{\theta}^{\mu'} \leftarrow \tau\boldsymbol{\theta}^{\mu} + (1-\tau)\boldsymbol{\theta}^{\mu'}$$

where $\tau \ll 1$ is the update rate constraining the target networks to change slowly which shows to greatly improve the stability of learning.

Since the policy is deterministic, following it would lead to no exploration of the environment. In order to incorporate this within the DDPG framework, an exploration policy $\boldsymbol{a}_t$ is derived by adding a noise component, $\mathcal{N}$, sampled from an Ornstein-Uhlenbeck (OU) process. The OU process $x_t$ is defined by the stochastic differential equation

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t, \tag{2.45}$$

where $\mu$, $\theta > 0$ and $\sigma > 0$ are constants, and $W_t$ denotes the Wiener process [59]. With this noise component, the exploration policy becomes

$$\boldsymbol{a}_t = \mu(\boldsymbol{s}_t|\boldsymbol{\theta}_t^{\mu}) + \mathcal{N}_t. \tag{2.46}$$

More recent results have suggested that uncorrelated, zero-mean Gaussian noise works well for providing the noise component for exploration[2], which could simplify implementations. Integrating all these advancements together, the DDPG algorithm takes the form shown in Algorithm 3 [30].

It should be noted that there are no guarantees that satisfying Bellman's equations will lead to obtaining an optimal policy. Empirically, one can derive an agent exhibiting great behavior and high performance, but the absence of guarantees renders such DRL algorithms potentially unstable and brittle. Descendants of DDPG such as TD3 [60] and SAC [61] are more evolved DRL algorithms where varying techniques are employed, aiming to mitigate the aforementioned challenges.

[2] https://spinningup.openai.com/en/latest/algorithms/ddpg.html

**Algorithm 3** DDPG algorithm

Randomly initialize the critic network $Q(s, a|\boldsymbol{\theta}^Q)$ and actor $\mu(s|\boldsymbol{\theta}^\mu)$ with weights $\boldsymbol{\theta}^Q$ and $\boldsymbol{\theta}^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^Q$, $\boldsymbol{\theta}^{\mu'} \leftarrow \boldsymbol{\theta}^\mu$
Initialize replay buffer $R$
**for** episode $= 1 : M$ **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $\boldsymbol{s}_1$
    **for** $t = 1 : T$ **do** (for each step in episode)
        Select action $\boldsymbol{a}_t = \mu(\boldsymbol{s}_t|\boldsymbol{\theta}^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $\boldsymbol{a}_t$ and observe reward $r_t$ and $\boldsymbol{s}_{t+1}$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in $R$
        Sample a random minibatch of N transitions $(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(\boldsymbol{s}_{i+1}, \mu'(\boldsymbol{s}_{i+1}|\boldsymbol{\theta}^{\mu'})|\boldsymbol{\theta}^{Q'})$
        Update the critic by minimizing the loss: $J = \frac{1}{N}\sum_i (y_i - Q(\boldsymbol{s}_i, \boldsymbol{a}_i|\boldsymbol{\theta}^Q))^2$
        Update the actor policy using the sampled policy gradient:

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

        Update the target networks:

$$\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$$
$$\boldsymbol{\theta}^{\mu'} \leftarrow \tau\boldsymbol{\theta}^\mu + (1-\tau)\boldsymbol{\theta}^{\mu'}$$

## 2.4 Quadrotor dynamics

In most control approaches the kinematics of the plant builds the foundation for developing the controller, and how accurate the model is often determines how well the controller performs. As suggested in earlier chapters, DRL methods do not directly depend on a model of the environment and control schemes based on these principles can therefore be designed without having previous knowledge of the plant model.

Despite the fact that RL methods being independent from a predefined model, it is still of high value for developers to understand the underlying physical properties of the plant for several reasons. Prerequisite knowledge of model kinematics and dynamics often proves integral when defining the reward function and can also be highly advantageous during analysis of an agent's behavior.

Correspondingly, the following section presents the equations of motion of a general quadrotor model. This yields fundamental understanding of a quadrotor's translational and rotational behavior when it is not affected by external forces. Furthermore, these equations describe how the Gazebo framework, presented in Section 3.1.1, simulates the

drone's behavior. Resultantly, this theory could be of valuable use when analyzing the performance of an agent in addition to Gazebo's ability to accurately simulate the quadrotor's behavior.

Through the derivations in [62], the nonlinear and coupled equations of motion for a general quadrotor with 6 degrees of freedom are expressed as:

$$\begin{aligned} m\ddot{x} &= (\sin\psi \sin\phi \ + \cos\psi \cos\phi \sin\theta)u_1 \\ m\ddot{y} &= (-\cos\psi \sin\phi \ + \sin\theta \sin\psi \cos\phi)u_1 \\ m(\ddot{z} + g) &= \cos\theta \cos\phi \ u_1 \\ I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z &= u_2 \\ I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_z\omega_x &= u_3 \\ I_{zz}\dot{\omega}_z &= u_4 \end{aligned} \tag{2.47}$$

In (2.47), $x$, $y$, and $z$ symbolize the linear positions of the drone with respect to the world reference frame $\mathcal{W}$ (also known as the earth-fixed reference frame), while $\phi$, $\theta$ and $\psi$ are the roll, pitch and yaw angles in $\mathcal{W}$, respectively. $\omega_x$, $\omega_y$, and $\omega_z$ represent the roll, pitch, and yaw rates of the drone in the body reference frame $\mathcal{B}$, where the relationship between $\mathcal{W}$ and $\mathcal{B}$ is illustrated in Figure 2.9. $\boldsymbol{u} = [u_1,\ u_2,\ u_3,\ u_4]^\intercal$ encapsulate the control parameters composed by the forces and moments generated by the drone propellers. $m$ is the mass of the quadrotor, while $I_{xx}$, $I_{yy}$ and $I_{zz}$ are the principal moments of inertia. It is worth noting that (2.47) assumes that the drone is axisymmetric.

Assuming that the quadrotor is in hovering mode, several simplifications can be conducted. Firstly, (2.47) shows that all linear states are subordinated to control parameter $u_1$, meaning only one of these states are controllable. Hence, the rest of the states are subject to the other controlled linear and angular motions. Considering only linear motion in $z$ and in hover mode, namely $\phi \approx 0$ and $\theta \approx 0$, the dynamics in (2.47) are simplified significantly:

$$\begin{aligned} m(\ddot{z} + g) &= u_1 \\ I_{xx}\ddot{\phi} &= u_2 - (I_{zz} - I_{yy})\dot{\theta}\dot{\psi} \\ I_{yy}\ddot{\theta} &= u_3 - (I_{xx} - I_{zz})\dot{\psi}\dot{\phi} \\ I_{zz}\ddot{\psi} &= u_4 \end{aligned} \tag{2.48}$$

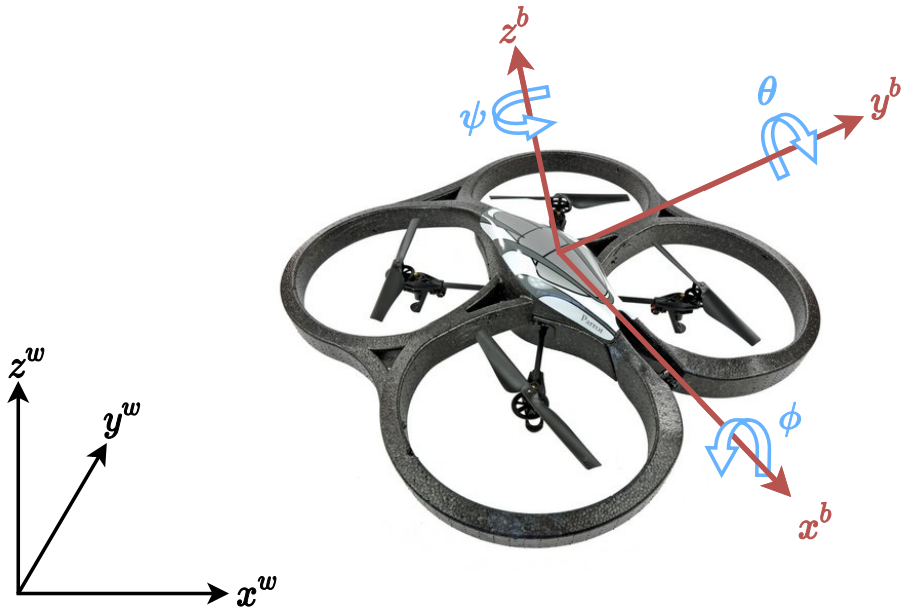



**Figure 2.9:** A visual representation of the relationship between the world frame $\mathcal{W}$, where the axes are denoted with $w$, and the body frame of the drone $\mathcal{B}$, denoted with $b$. The rotation directions for the body reference frame are also illustrated, where the right-hand rule is the convention applied. These coordinate frames form the basis for the extraction of the equations of motion stated in (2.47).

Chapter 3

# Experimental setup

## 3.1 Software frameworks

The following section presents the various software frameworks used in this thesis. In order to allow the scope of the work to be within feasible range, several frameworks were exploited in order to execute underlying tasks, such as communication, simulation and training DNNs. All software utilized are open-source.

### 3.1.1 Gazebo

Gazebo is a 3-dimensional dynamic robotics simulator that supports a range of physical dynamics in order to generate realistic models and environments. Gazebo has the ability to simulate complex indoor and outdoor environments and is hence able to accurately and efficiently model the behavior of a wide range of robotic plants in various settings. The simulator also supports sensor simulation and actuator control.

In order to fit the simulated robotic entities as closely to the real plants as possible, Gazebo considers many vital components of a rigid object, such as mass, friction and inertia. Gazebo offers physics simulation at a high degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Gazebo builds upon the open source library Open Dynamics Engine (ODE) [63] for calculating the dynamics and kinematics for all rigid bodies inside the simulated environment. Gazebo represents sensors, environmental scenes and the links and joints of robot models through Universal Robot Description Format (URDF) files. These URDF files are represented in XML format. A URDF file describing a drone, for instance, will encapsulate a detailed description of all elements and physical properties of the robot, its link to the world coordinate frame and detailed 3-dimensional geometrical meshes for visualization and realistic collision handling.

Gazebo is a key component in the work presented, as it allowed visualization and deeper interpretations of the modeled system at hand. It also yielded feasible and efficient testing by allowing running code under development without having to risk a real-life drone. Resetting and spawning the agent in specific locations in the environment during the training was straightforward, which was quite practical in terms of streamlining the learning process of the DRL agent. Further, availability of additional features such as sensor and pose measurements directly from the simulator allowed analysis and debugging of the developed software. This was done while still behaving very similarly to the real world plant with respect to accuracy of the simulated model dynamics compared to the real version of the drone. This, in turn, would allow more seamless transfer of logic from simulator environments to real-life applications with only minor adjustments.

### 3.1.2 ROS

Robot Operating System (ROS) [64] is a collection of software frameworks tailored for developing robot applications. It provides tools and libraries that has contributed significantly to many advances in the field of robotics. Being open source, ROS is a key building block for numerous advances within the research and development communities and lays the foundation for many implementations in today's technology.

ROS offers message-passing and package management, contributing to seamless and straightforward communication between processes. So-called ROS topics are used for most of the communication. Low-level control of robotic devices is also facilitated thanks to the device drivers and hardware abstractions that ROS provides. Additionally, visualizers, logging and sensor modeling for creating realistic scenarios is also made possible using this framework. One of ROS's strengths is its flexibility in terms of the programming languages supported, among these being Python and C++. Resultantly, one can quite easily set up modular designs for robotic applications. A ripple effect of supporting multiple languages is that the online community advances and provides plentiful documentation.

Conveniently, ROS is embedded into Gazebo, making the communication between them smooth. Gazebo provides services that allow ROS to pause and unpause simulations and fetch states during training. This was found to be highly advantageous, especially when implementing DRL based solutions that often require a controllable environment in addition to analysis during run-time.

Due to the fact that ROS interfaces well with Gazebo, has the ability to facilitate modular solutions and provides a cooperative online community and comprehensive documentation, ROS is deemed a well-suited framework to utilize for developing applications in this master thesis. The work presented is based on the ROS Kinetic distribution with Ubuntu 16.04 LTS as the operating system.

### 3.1.3 Tensorflow

Tensorflow [65] is an end-to-end open source library tailored for building, training and deploying ML models, and is on the forefront of most applications that employ deep learning approaches. Tensorflow permits automatic differentiation of neural networks by construct-

ing *computational graphs*. These graphs are high-level abstraction structures that define the data flow in a neural network, which allow Tensorflow to develop and deploy DNN models with relative ease.

High-level APIs are also included, such as `tf.keras`, which is TensorFlow's implementation of the vastly used Keras API [66]. Keras offers several implementations of vastly used DNN building blocks, such as hidden layers, optimizers, objective functions, activation functions and regularizers as well as dropout, batch normalization, and pooling. Additional functionality such as eager execution, data pipelines and estimators allow immediate model iteration and straightforward debugging. Tensorflow offers great frameworks together with a Python API and comply with most necessities required for building state-of-the-art DNN architectures while still accounting for agile prototyping, which suits this project well.

## 3.2 Quadrotor platform

The quadrotor used in this thesis was the Parrot AR.Drone 2.0 Elite Edition[1]. The reasoning behind this included both price, specifications and availability for extra parts. The drone itself costs approximately NOK 1300, deeming it one of the cheapest on the market with similar specifications. Spare parts such as propellers and batteries were easily attainable and the quadrotor also included a HD 720p frontal camera and a QVGA vertical camera. Since the work consisted of developing systems that would require experimentation where the drone may be damaged during runs or can run out of power during longer testing sessions, it was laid extra emphasis on the availability of extra parts. A camera with relatively high resolution was also an important criteria since this would directly benefit the accuracy of the drone's state estimation for the interconnected perception estimation project [39]. Since the AR.Drone 2.0 ticked all these boxes, it was deemed a suitable candidate for the scope of this thesis.

Despite the previously mentioned specifications, the main factors for choosing the AR.Drone 2.0 was that it was open source and included an already-implemented simulator for the drone in Gazebo, facilitating unambiguous development of the methods in this thesis. It also featured compatibility for both the real drone and the simulated drone with ROS. Using ROS as the communication protocol would deem straightforward development, since the same code and models could be used for both the real and the simulated drone with minor adjustments. Access to the simulator, given by the ROS package `tum_simulator` [5], was integral for this project in terms of training the agent as realistically and efficiently as possible while developing the methods presented in Chapter 4. By this we mainly mean that the behavior of the simulated drone was transferable to the real plant, such that real-life testing would mirror simulations as accurately as possible.

### 3.2.1 Hardware and sensors

The AR.Drone 2.0 includes computationally capable but still lightweight hardware. The drone is equipped with a 32-bit ARM Cortex A8 processor that runs at 1 GHz with Linux

[1] https://www.parrot.com/global/drones/parrot-ardrone-20-elite-edition

2.6.32 as its operating system. This allows the open source aspect of this specific drone. It also includes a 1 GB RAM unit and a high-speed USB 2.0 for extensions, such as flight recorders. With internal frame only the drone's mass is 380 grams, and with the external frame it increases to 420 grams. With the latter configuration the drone measures 53 cm $\times$ 52 cm horizontally.

The quadrotor also includes a wide range of sensors whose purpose is to stabilize the plant and yield user-friendly and accurate motion. The AR.Drone 2.0 is supplied with an inertial measurement unit (IMU), consisting of a gyroscope, accelerometer and magnetometer. These sensors measure in three axes and provide translational and rotational information of the drone. The AR.Drone 2.0 also includes a pressure sensor and altitude ultrasound sensor for altitude estimation. This assortment of sensors, accompanied by the hardware, aims to accurately estimate the linear and angular motion of the drone such that the velocity controller presented in Section 3.2.2 can function as wanted.

In addition to the hardware and sensors presented, the drone features one frontal and one vertical camera. The former is a frontal camera with 720p video quality. It has a 93 degree lens and can record up to 30 frames per second. This camera has no stabilizing function to the drone, but is rather used for observation purposes. The latter is a QVGA camera and aims to measure the ground speed of the drone, helping it stabilize the horizontal dynamics. It has a 64 degree lens and has a record rate of 60 frames per second. The resolution of each frame is 320 $\times$ 240 but are scaled up to 640 $\times$ 360 automatically.

In addition to the USB port, this drone also has a Wi-Fi interface. Users can connect to the drone via this Wi-Fi network and control the drone by sending velocity commands, as well as retrieving sensor data. The former is fundamental for applying the developed algorithms on the real plant, while the latter allows analysis of performance in addition to debugging. All sensor measurements and estimated horizontal velocities are available to a computer through this Wi-Fi interface at a frequency of up to 200 Hz.

The complete specifications of the Parrot AR.Drone 2.0 Elite Edition can be found at https://www.parrot.com/global/drones/parrot-ardrone-20-elite-edition under "Technical specifications".

### 3.2.2 Built-in velocity controller

The physical drone does not allow manipulation of the low-level voltage signals for the propellers directly. Rather, AR.Drone 2.0 features a built-in velocity controller. The velocity controller Ar.Drone 2.0 utilizes is based on the structure depicted in Figure 3.1 and consists of a set of cascaded PI, PD and PID controllers. The purpose of the inner loops is to control the attitude, yaw rate and vertical velocity, while the outer loops control the horizontal velocity, heading and altitude. This velocity controller adopts its architecture by assuming that each axis and the altitude can be controlled independently. This assumption holds for reasonable deviations from the hovering state. The controller's behavior is duplicated by the simulator implementation and added in the solution of tu-darmstadt-ros-pkg. As a result, simulations will accurately mimic the behavior of the real drone.

It is worth noting that AR.Drone 2.0's controller deviates slightly from Figure 3.1. As a

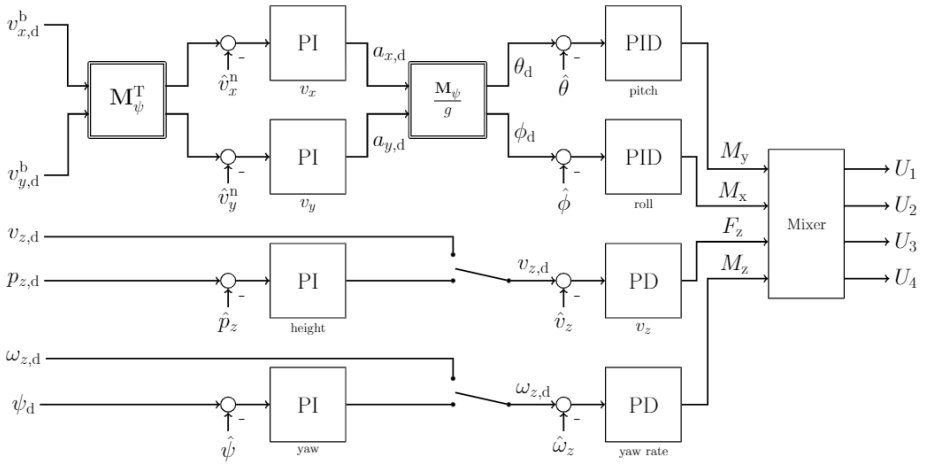



**Figure 3.1:** The controller realized in the `tu-darmstadt-ros-pkg` ROS package, in which `tum_simulator` bases their solution on. The block diagram is taken from [67].

result of the sensors the drone inhabits, it is not able to control the desired altitude $p_{z,d}$ nor the desired heading $\psi_d$, only their velocities $v_{z,d}$ and $\omega_{z,d}$. In the figure, this translates to the two bottom switches in the figure always being connected to the velocity signals. Thus, the input to the velocity controller becomes a 4-dimensional vector expressing the desired horizontal and vertical velocities in addition to the yaw rate, namely

$$\boldsymbol{v}_d = \begin{bmatrix} v_{x,d} \\ v_{y,d} \\ v_{z,d} \\ \omega_{z,d} \end{bmatrix}. \tag{3.1}$$

It is assumed that all velocities in (3.1) are expressed in the body frame. The simulator operates with ranges to these values, where maximum velocity in the negative direction translates to setting the corresponding value to $-1$, while 1 yields maximum velocity in the positive direction.

Figure 3.1 illustrates that the desired horizontal velocities expressed in the body frame, $v^b_{x,d}$ and $v^b_{y,d}$, are translated to desired roll and pitch angles, $\phi_d$ and $\theta_d$, and controlled based on these values. This is emphasized in Figure 3.2, where the figure portrays how the roll and pitch angle of the drone respond when a step to the desired velocity in $x$ and $y$ is applied. The figure also shows how accurate the velocity controller is. According to Figure 3.2 velocity controller seems to be quite rapid in its response, and seems to stabilize its velocity in addition to its angles quite quickly after the desired velocity is acquired.

As most of the sensors the drone is equipped with have to integrate their measurements in order to obtain position and orientation, these quantities are subject to drifting behavior given the design of the velocity controller. This is emphasized in Figure 3.3. The inclusion

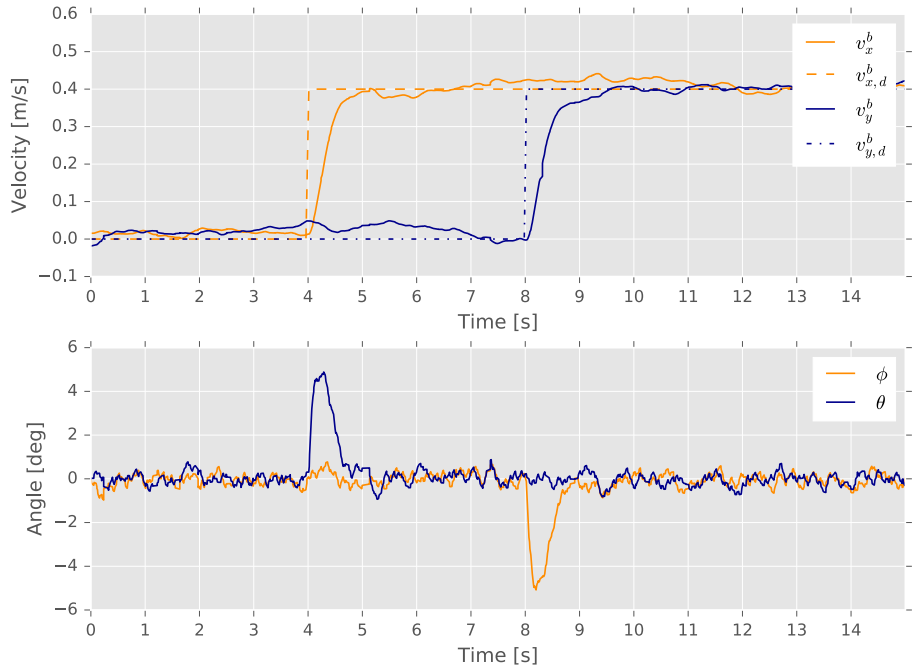



**Figure 3.2:** Exemplification of how the velocity controller's design introduces the coupled behavior between the horizontal velocities and the roll and pitch angle. At $t = 4$ the drone receives a velocity command purely in $x$. This induces a response in the pitch angle such that the drone tilts and generates a horizontal force, pushing the drone in positive $x$ direction. At $t = 8$ the same incident occurs, although in the $y$ direction. This induces a *negative* roll angle to generate velocity in the positive $y$ direction. Figure courtesy of [37].

of the pressure and ultra-sound sensors aids the quadrotor to directly measure its vertical position $z$ rather than estimating it indirectly. The drone is therefore quite competent in altitude estimation, and therefore altitude control. Compared to the vertical position, the horizontal position $(x, y)$ drifts considerably since these are not controlled directly, but rather through the pitch and roll angles. Since the roll and pitch angles are controlled directly by the velocity controller, these will resultantly not be as affected by drift. The yaw angle, on the other hand, is not controlled in the same fashion and will therefore experience drift. This is also portrayed in Figure 3.3.

The inclusion of the velocity controller may be advantageous, seen that it allows the developer to base the control schemes on top of this controller and resultantly not having to assess underlying dynamics when applying commands to the plant. It provides common ground between the simulated plant in Gazebo and the real drone, since the dynamics of the plant is already accounted for in the velocity controller. This can potentially lead to higher accuracy of the simulated drone dynamics and less ambiguous behavior between simulator and real quadrotor.

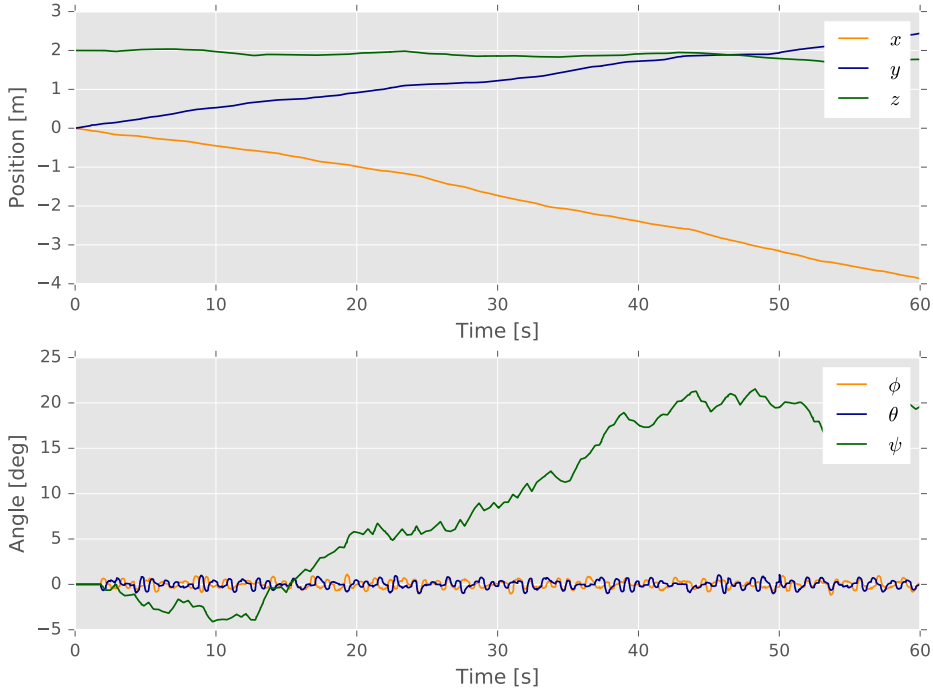



**Figure 3.3:** Simulation highlighting the drifting conditions of the drone's pose, namely its position and orientation. The simulation was conducted over a 60 second interval, where the drone was initialized at $[x, y, z, \phi, \theta, \psi] = [0.0, 0.0, 2.0, 0.0, 0.0, 0.0]$. There were no applied velocity commands during the simulation. Figure courtesy of [37].

Unfortunately, at the time of writing there were no scientific research publications aiming to examine the accuracy of the simulator compared to the real drone. However, the simulator developers have published a video showcasing a comparison in behavior between the real drone and the simulated drone when both are applied with the same series of velocity commands [68].

The velocity controller can also prove to serve as a stabilizing safety guard for training the quadrotor. With the controller the drone suggests behavior showing that no series of velocity commands result in the quadrotor flipping upside down or rolling or pitching such that the model in Section 2.4 no longer is valid. In contrast, such behavior would most likely not have been present if the developer had direct access to modification of the voltage signals.

Simultaneously, it could be argued that an RL based agent may learn more efficiently by directly manipulating and learning the low-level voltage commands. The main argument for desiring such an end-to-end solution is that it allows an agent to optimize its overall behavior instead of optimizing individual modules of a hierarchy [69, 70]. Preferably, the developers should have access to manipulating both the higher level control inputs as

well as the low-level voltage signals for comparative and flexibility purposes for having the ability to develop an end-to-end system. On a general note the drone market seems to lack alternatives fulfilling this requirement, and it should still be feasible to acquire a satisfactory behavior of an agent using this velocity controller as a base.

Chapter 4

# Methodology and system design

## 4.1 System architecture and setup

Many quadrotor control approaches include a set of PID controllers aiming to regulate one or several states. Although these approaches have proved to work quite well, there are some significant drawbacks. Tuning the parameters of the different controllers is often not a trivial task. Since the system is highly nonlinear and several states are coupled, changes to the behavior of one state may propagate through the system and lead to undesired behavior in other states. Tuning the controllers too aggressively may render the drone unstable, while too cautious and passive tuning may impair the drone's agility and responsiveness, which may lead to the drone not being able to respond to wind gusts or other unforeseen events.

Another popular approach is MPC. This method also exhibits well-behaved solutions, though these are often in either controlled environments where the model is accurately defined and wind gusts are not present, or where disturbances and other external forces are accurately modeled. The solutions are generally inflexible to changes in the environment or deviations in the model. There are several ways of mitigating these shortcomings, among these being to taking e.g. wind gusts into account during the modeling phase and do extensive testing in order to obtain an accurate model of the plant and external components. These measures can yield sufficient solutions that are capable of handling most cases in quadrotor flight, but are often dismissed due to the nearly infeasible amount of time and resources it would require to develop such a detailed system.

Resultantly, both methods will lead to suboptimal global performance if not enough time and care is put into the modeling phase, motivating for alternative and more robust solutions that are not as dependent on this step. Appropriately, this thesis proposes a DRL inspired solution to quadrotor control. DRL methods have the benefit of learning a global policy whose task is to translate the drone's state to velocity commands and hence eradicat-

ing the need for designing several coupled PID controllers. Further, DRL agents optimize a performance measure defined to complete the objective. While improving its performance the agent implicitly takes the system dynamics into account, where methods such as MPC would have to do this explicitly. As such, DRL agents have the potential of ”understanding” the underlying dynamics without having an explicit model, and act appropriately after it has been trained.

This thesis aims to divide the general objective of the landing mission into three steps: hover, descend and land. The first step considered the case where the drone starts at an arbitrary position and would maneuver towards the landing platform and hover above it at a specified height. The second step would follow up with descending towards the landing platform while maintaining the horizontal position above it, before it reaches a specific height. The goal is to descend while still keeping its horizontal position static. When the drone is close enough to the platform for it to land without missing the target nor damaging the drone itself, the final step of landing is initiated. The evolution of these steps are illustrated in Figure 4.1, and constitute a simple planning system for completing the landing task in the given environment where there are no obstacles and no external disturbances other than the drift conditions previously described.

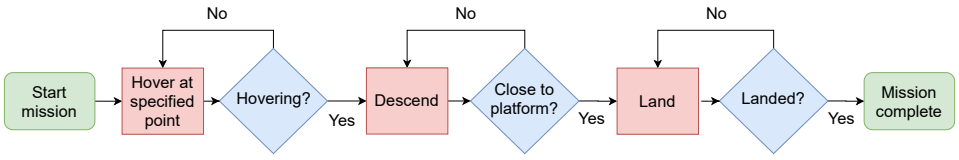



**Figure 4.1:** Flow chart of the different steps to be conducted for completing the objective. Using conventional flow chart notation, the activities are in rectangles and the decisions in diamonds. Each decision has a set of specific requirement to output either yes or no.

Note that the planning approach presented is very rudimentary, where no real fault tolerance nor edge cases have been considered. For a more robust system aimed to be utilized in real-world applications and more advanced missions, the planning system is vital and must be investigated and designed with higher level of detail. Since it was assumed that there were no obstacles nor external forces hindering the drone’s movement and that the agent knew from the get-go where the landing platform was located, this simplistic planning scheme proved to be sufficient for the scope of this thesis and was therefore not augmented further.

The two former steps of Figure 4.1 are solved independently by using the novel DDPG algorithm, as discussed in detail in Section 4.2. Since the quadrotor inhabits a built-in landing command both for the real plant and for the simulated drone in Gazebo, the final landing step could be conducted using this feature instead of designing and training a dedicated landing agent. Resultantly, this thesis aims to develop a hovering agent and a descending agent for accomplishing autonomous landing through DRL. Each agent is independent from the other, where their respective objectives are described as follows:

1. The hovering agent is tasked to control the drone’s linear position, $(x, y, z)$, towards a specific point in which the quadrotor has to be able to see the entirety of the

platform while hovering at still.

2. The descending agent is assigned to approach the agent's linear positions, $(x, y, z)$, towards the landing platform, given that the quadrotor's initial position is in the proximity of the point of hover. This agent is also subject to control the yaw rotation, $\psi$.

It can be argued that a more direct and less time-consuming solution would be to train *one* agent to directly approach the landing platform instead of executing the intermediate step of hovering at a specific height. However, since the long term goal of this project was to create a real-life implementation with external influences, this scheme was favorable in terms of safety and robustness of the solution. Though the quadrotor most likely will consume considerably more power and be less efficient seen from a time aspect, it was concluded that robustness and safety of the mission was of higher importance than temporal constraints and power consumption.

The Gazebo environment used for training and testing was a world containing free space with the helipad and the simulated AR.Drone 2.0, as depicted in Figure 4.2. Since the Gazebo simulator had the ability to run simulations faster than real-time, the environment for training was designed to be as simplistic as possible with only the necessary models. This was done mainly to limit the underlying calculations that has to be conducted in order to render other, nonessential entities in the simulation. This design detail in mind yielded enhanced speed-up and reduced training time.

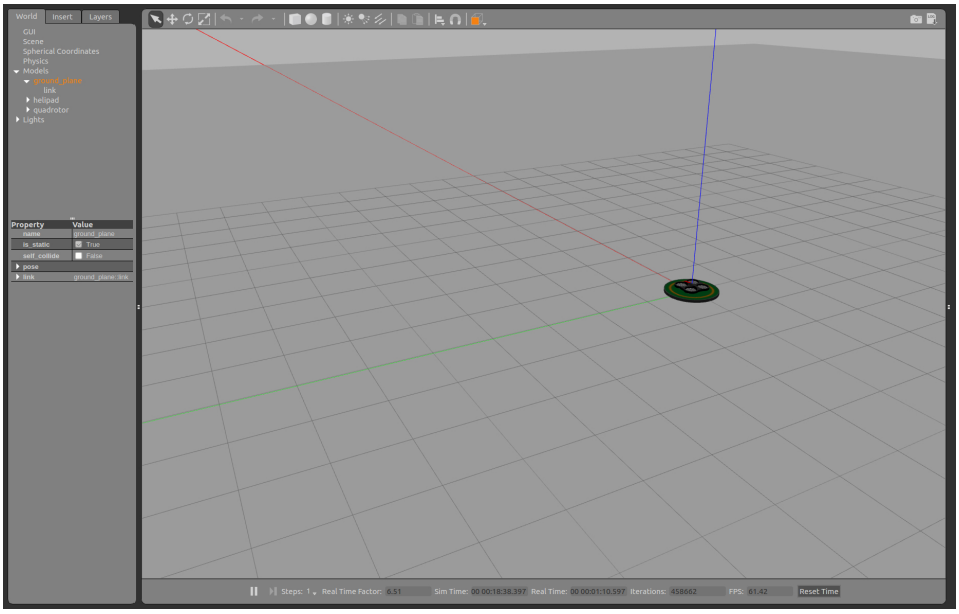



**Figure 4.2:** The simple simulation environment modeled in Gazebo during training.

The following sections describe how the two previously mentioned submissions were designed and implemented using DRL in order to obtain a drone system able to land on a

platform.

## 4.2 Quadrotor control using DDPG

As discussed earlier, this thesis mainly aims to contribute through research on how DRL may be applied to learn control policies for controlling a quadrotor to land on a platform. By assuming that the position of the helipad in question is known, the objective is to create a robust system able to hover above, descend towards and land on the platform. To accomplish this, the novel DDPG algorithm presented in Section 2.3.3 was used.

The reasoning behind adopting DDPG as the controller scheme was mainly due to it being able to handle continuous state and action spaces, in addition to being an off-policy algorithm. The latter meant that a DDPG agent can learn a desired policy without having to follow it itself. In practical terms, this implied that a DDPG agent has the ability of learning its objective by observing transitions generated from a different agent. Since exploratory learning schemes seldom avoid executing suboptimal actions, the off-policy element of the learning process is deemed highly advantageous for a physical drone that may be subject to safety regulations.

In coherence with the conventional actor-critic scheme illustrated in Figure 2.8, an overview of the architecture of the implemented DDPG solution is given in Figure 4.3. The main components are the control policy and the value function, $\mu(\boldsymbol{s})$ and $Q(\boldsymbol{s}, \boldsymbol{a})$, respectively. The former pledges to generate a velocity command given the drone state, while the latter has the task of evaluating how fitting the command was in that specific state.

In DDPG, both of the actor and critic are estimated through neural networks. The actor network, $\mu(\boldsymbol{s}|\boldsymbol{\theta}^{\mu})$, was built as a fully connected network with two hidden layers. The input layer was of the same dimension as the state. The first hidden layer had 400 units, while the second had 300 units in the original implementation. Both hidden layers had the rectified linear unit ($\texttt{ReLU}$)[1] as activation [71]. This was done in order to allow the network to approximate nonlinearities in the policy. The output layer had the same dimension as the action commands, since this layer would express the estimated optimal action to execute, given the inputted state. In order to scale the outputs to the range $[-1, 1]$, the activation function at the output layer was set to the hyperbolic tangent, $\texttt{tanh}$. Following the notation introduced in Section 2.1.3, the actor network could be presented as:

$$\begin{aligned}
\boldsymbol{h}^1(\boldsymbol{s}) &= \texttt{ReLU}(\boldsymbol{W}^{1,\mu}\boldsymbol{s} + \boldsymbol{b}^{1,\mu}) \\
\boldsymbol{h}^2(\boldsymbol{s}) &= \texttt{ReLU}(\boldsymbol{W}^{2,\mu}\boldsymbol{h}^1(\boldsymbol{s}) + \boldsymbol{b}^{2,\mu}) \\
\mu_{\texttt{raw}}(\boldsymbol{s}) &= \texttt{tanh}(\boldsymbol{W}^{3,\mu}\boldsymbol{h}^2(\boldsymbol{s}) + \boldsymbol{b}^{3,\mu})
\end{aligned}$$

In order to ensure that the action magnitudes were within saturating limits of the actuator controlled, namely

$$\boldsymbol{a}_{\texttt{min}} \leq \boldsymbol{a}_t \leq \boldsymbol{a}_{\texttt{max}}, \tag{4.1}$$

[1] The activation function of the rectifier is given as $\texttt{ReLU}(x) = \max(0, x)$.

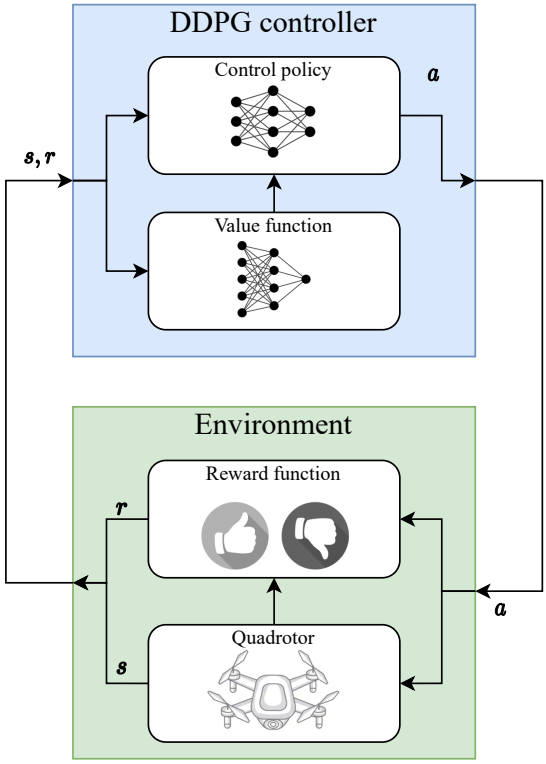



**Figure 4.3:** A figure highlighting the components and the signal flow of the system developed in this thesis. The DRL controller is highlighted in blue and the environment encapsulating the quadrotor platform in green.

the raw output would need to be scaled and shifted through a linear transformation:

$$\mu(\boldsymbol{s}|\boldsymbol{\theta}^{\mu}) = \mu_{\texttt{raw}}(\boldsymbol{s})\boldsymbol{a}_{\texttt{scale}} + \boldsymbol{a}_{\texttt{shift}}. \tag{4.2}$$

This transformation would facilitate avoiding wear and tear on the quadrotor motors in addition to restricting the plant's behavior such that the control policy would not yield unneeded aggressive actuation.

Assuming the shift and scale variables were known, this architecture constituted the following actor network parameters

$$\boldsymbol{\theta}^{\mu} = [\boldsymbol{W}^{1,\mu}, \boldsymbol{b}^{1,\mu}, \boldsymbol{W}^{2,\mu}, \boldsymbol{b}^{2,\mu}, \boldsymbol{W}^{3,\mu}, \boldsymbol{b}^{3,\mu}].$$

In similar fashion, the critic network, $Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}^{Q})$, was also modeled as a fully connected network with two hidden layers. Each layer used `ReLU` and had 400 and 300 units, respectively. Since the critic was tasked to approximate the Q-function, the network was inputted a state-action pair and the output was the estimated Q-value of that specific pair. The state was inputted at the first layer, while the action was not included until the second

hidden layer. Summarized, the critic network had the following architecture:

$$\begin{aligned} \boldsymbol{h}^1(\boldsymbol{s}) &= \texttt{ReLU}(\boldsymbol{W}^{1,Q}\boldsymbol{s} + \boldsymbol{b}^{1,Q}) \\ \boldsymbol{h}^2(\boldsymbol{s},\boldsymbol{a}) &= \texttt{ReLU}(\boldsymbol{W}^{2,Q}\begin{bmatrix}\boldsymbol{h}^1(\boldsymbol{s}) \\ \boldsymbol{a}\end{bmatrix} + \boldsymbol{b}^{2,Q}) \\ Q(\boldsymbol{s},\boldsymbol{a}|\boldsymbol{\theta}^Q) &= \boldsymbol{W}^{3,Q}\boldsymbol{h}^2(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{b}^{3,Q} \end{aligned}$$

Analogous to the actor parameters, the critic network parameters were given as

$$\boldsymbol{\theta}^Q = [\boldsymbol{W}^{1,Q}, \boldsymbol{b}^{1,Q}, \boldsymbol{W}^{2,Q}, \boldsymbol{b}^{2,Q}, \boldsymbol{W}^{3,Q}, \boldsymbol{b}^{3,Q}].$$

With the same notation as in Section 2.1.3, Figure 4.4 summarizes the architecture used for the actor and critic networks in this thesis.

A more detailed figure reinforcing Figure 4.3 is given in Figure 4.5. This figure shows that the 5-tuple transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1}, d)$ is stored inside the replay buffer for each step inside the environment, where $d$ constitutes a flag that is set whenever the agent is *done* and has terminated the episode. This flag would only be set when the agent exited the valid area. Training of the actor and critic was also done at each step, after storing the transition. A batch of random transitions in the replay buffer was extracted and the optimizers of the networks ensured that the network parameters were updated accordingly.

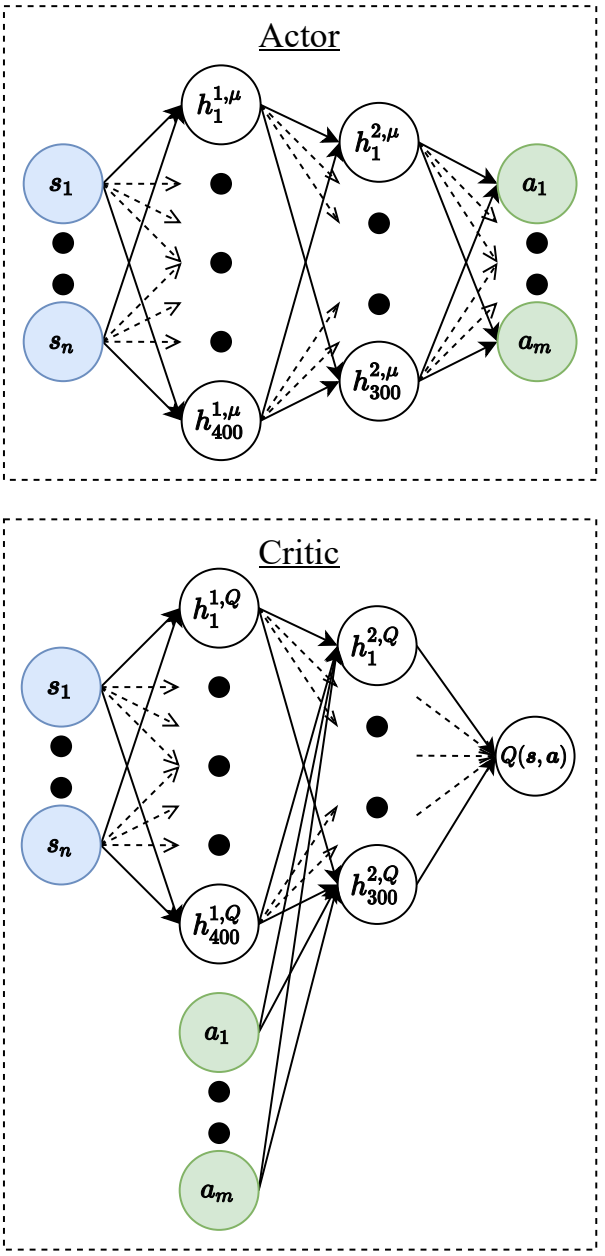



**Figure 4.4:** The network architecture of the actor and critic in the implemented DDPG solution. The state and action dimensions are denoted $n$ and $m$, respectively.

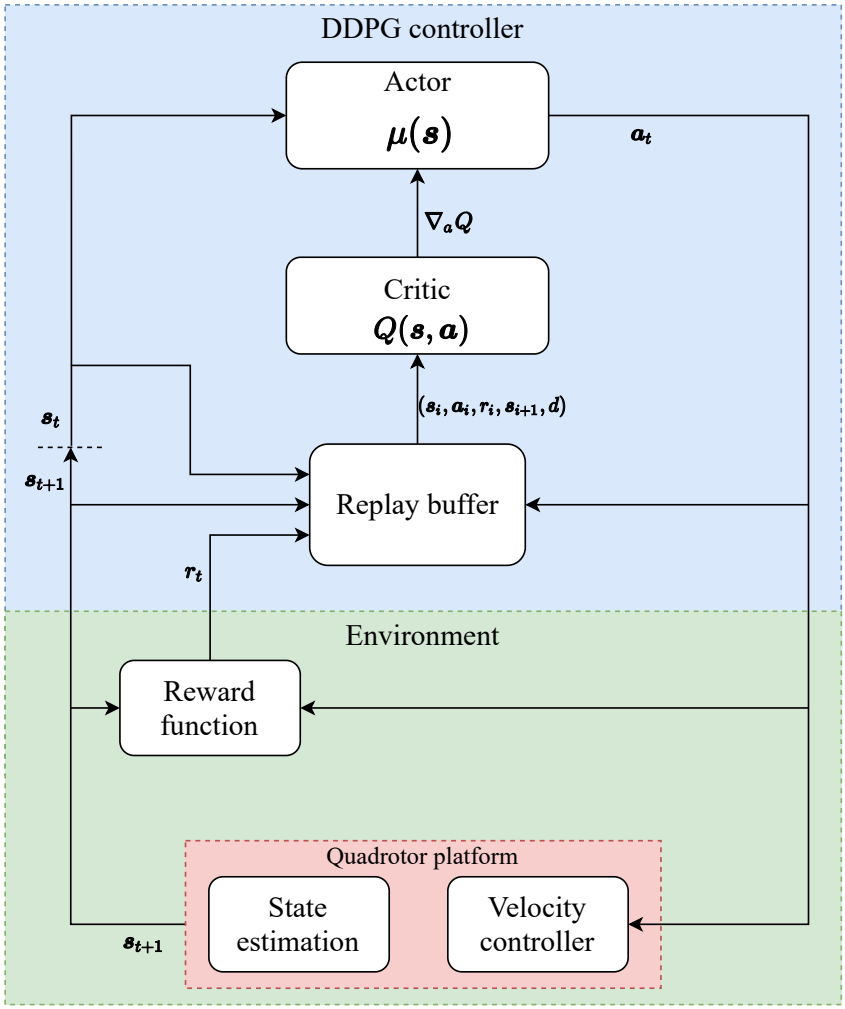



**Figure 4.5:** A more detailed block diagram portraying the architecture of the system developed in this thesis. The figure augments Figure 4.3 and illustrates the controller, highlighted in blue, and the environment, including the quadrotor platform, in green. Further, the figure introduces the replay buffer in addition to the modules that the quadrotor platform consists of, namely the state estimator and built-in velocity controller. The two latter entities are boxed in red.

### 4.2.1 State and action representation

For controlling a plant using DRL, it is important to give the agent sufficient amounts of information for it to be able to learn and optimize its objective. This information is conveyed through the state vector $\boldsymbol{s}$. For position control of a quadrotor, a minimalistic state vector would be the linear positions, namely

$$\boldsymbol{s} = \boldsymbol{x} = [x, y, z]^\intercal, \tag{4.3}$$

This would, however, lead to suboptimal policies, as the solution would neither be invariant to rotation, nor accounting for the momentum of the drone when approaching a setpoint. A state vector that holds limited knowledge will increase the difficulty in the mapping from $\boldsymbol{s}$ to $\boldsymbol{a}$ and hence make the extraction of a policy more demanding. This forces the agent to do extensively complex transformations of the state in the actor and critic networks. Even though these networks are most likely capable of computing such transformations, by manually including additional states to the state vector one reduces the amount of network-computed transformations that need to be carried out, and subsequently decreases the architecture complexity and training time. This renders a more flexible agent easier to train without adding extensive computations or network restructuring.

As mentioned, the work presented in this thesis constitutes the design and implementation of two separate agents designated for hovering and descending, respectively. The following sections present the state and action representations of these agents.

#### 4.2.1.1 Hover

Since the objective of the hovering agent is to control the drone such that it stays at a given 3-dimensional point, it was imperative to include the positional error between the drone's position in $\mathcal{W}$, $\boldsymbol{x}^w$, and its desired hovering setpoint in $\mathcal{W}$, $\boldsymbol{p}_h = [p_{x,h}, p_{y,h}, p_{z,h}]^\intercal$, namely

$$\tilde{\boldsymbol{x}}^w = \boldsymbol{x}^w - \boldsymbol{p}_h = \begin{bmatrix} x - p_{x,h} \\ y - p_{y,h} \\ z - p_{z,h} \end{bmatrix}. \tag{4.4}$$

The superscript $w$ points out that this positional error is given in the world frame, since both $\boldsymbol{x}$ and $\boldsymbol{p}_h$ are given in $\mathcal{W}$. Further, by taking the arguments previously mentioned into account, it is advantageous to include states that offer information regarding the momentum of the quadrotor. In light of this, the translational velocities,

$$\boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}, \tag{4.5}$$

and translational accelerations of the drone,

$$\dot{\boldsymbol{v}} = \begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ \dot{v}_z \end{bmatrix}, \tag{4.6}$$

were included in the state vector.

While $\boldsymbol{v}$ and $\dot{\boldsymbol{v}}$ are calculated through the manifold of sensors mounted on the drone and thus given in the body frame, $\tilde{\boldsymbol{x}}^w$ is given in the world frame. Since the velocity commands are also given in the body frame, as stated in (3.1), a transformation between the states and the actions is necessary in order to maintain coherence between the state at timestep $t$, $\boldsymbol{s}_t$, the velocity command at that timestep, $\boldsymbol{a}_t$, and the resulting state in the next timestep, $\boldsymbol{s}_{t+1}$. By assuming that the quadrotor predominantly maintains small angles in both roll and pitch, these angles can be neglected. Thus, the transformation from $\mathcal{W}$ to $\mathcal{B}$ only consists of a negative yaw rotation about $z$. This rotation is illustrated in Figure 4.6 and is expressed with the following rotation matrix:

$$\boldsymbol{R}_w^b = \boldsymbol{R}_{z,-\psi} = \begin{bmatrix} \cos(-\psi) & -\sin(-\psi) & 0 \\ \sin(-\psi) & \cos(-\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

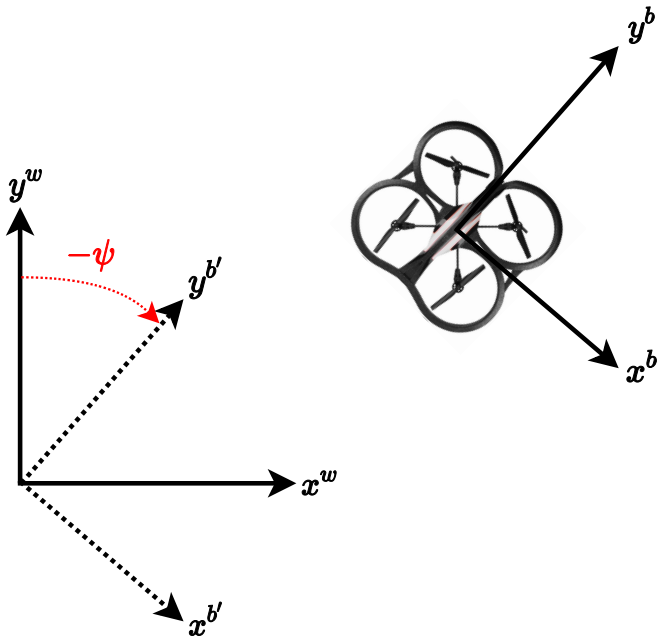



**Figure 4.6:** A bird's-eye view illustration of the relationship between $\mathcal{W}$ and $\mathcal{B}$ in the quadrotor environment. The frame $\mathcal{B}'$ symbolizes the body frame translated to the world frame origin. The figure bases its axes upon the right-hand rule, such that the $z$ axes of all three frames point outwards from the paper.

Resultantly, the positional error in the body frame is obtained in the following manner:

$$\tilde{\boldsymbol{x}} = \boldsymbol{R}_w^b \tilde{\boldsymbol{x}}^w$$

Finally, the state vector for the hovering agent became the 9-dimensional vector

$$\boldsymbol{s} = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \boldsymbol{v} \\ \dot{\boldsymbol{v}} \end{bmatrix}. \tag{4.8}$$

It could be argued that the yaw angle could have been included in the state vector such that the agent itself would be able to extract this transformation, but since the transformation was easily accessible this was deemed redundant.

In the implementation of the hovering controller, the control signal was on the form

$$\boldsymbol{a} = \begin{bmatrix} v_{x,d} \\ v_{y,d} \\ v_{z,d} \end{bmatrix}, \tag{4.9}$$

where the subscript $d$ denotes desired velocity. Note that all components are expressed in the body frame, as suggested earlier. Despite the fact that the velocity controller allowed control of the angular velocity about $z$, as suggested in Section 3.2.2, this was omitted from this control task since it had no significant role for the drone completing the hovering objective.

#### 4.2.1.2 Descend

The descending agent would resemble the hovering agent notably, though having some slight adjustments due to the difference in objective. In addition to the translational velocities and accelerations, it also made sense to include the positional error between the quadrotor's position $\boldsymbol{x}^w$ and the descending point $\boldsymbol{p}_d$, both given in $\mathcal{W}$. Furthermore, the objective of the descending agent included maintaining a relatively small yaw angle, thus augmenting its functionality and objective relative to the hovering agent. This addition to the objective was mainly motivated by future development where a computer vision module would be adopted to estimate the landing platform's position, see Section 4.2.3. Resultantly, the yaw angle $\psi$ would need to be included in the control signal $\boldsymbol{a}$:

$$\boldsymbol{a} = \begin{bmatrix} v_{x,d} \\ v_{y,d} \\ v_{z,d} \\ \omega_{z,d} \end{bmatrix}, \tag{4.10}$$

where $\omega_{z,d}$ denoted the desired angular velocity in the $z$ direction, thus controlling the yaw rate of the plant. Identical to the hovering agent, all components of the control signal in the descend agent is given in $\mathcal{B}$.

As stated, the descending agent had the additional responsibility of driving the yaw angle towards a desired value. The yaw angle also had to be included in the state vector in order to convey sufficient information to the agent when it was exhibiting satisfying control in yaw. Concatenating the position vector with the yaw angle in the world frame,

$$\boldsymbol{x}_a^w = \begin{bmatrix} \boldsymbol{x}^w \\ \psi \end{bmatrix}, \tag{4.11}$$

and defining the setpoint for the descending agent,

$$\boldsymbol{p}_d = \begin{bmatrix} p_{x,d} \\ p_{y,d} \\ p_{z,d} \\ p_{\psi,d} \end{bmatrix} \tag{4.12}$$

yielded the *augmented* positional error vector:

$$\tilde{\boldsymbol{x}}_a = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \tilde{\psi} \end{bmatrix} = \begin{bmatrix} x - p_{x,d} \\ y - p_{y,d} \\ z - p_{z,d} \\ \psi - p_{\psi,d} \end{bmatrix}. \tag{4.13}$$

This allowed the state vector $\boldsymbol{s}$, in similarity to the control signal $\boldsymbol{a}$, to be augmented for the descending agent's application. Following the extended objective of controlling the yaw angle, a 10-dimensional vector including the yaw angle was defined as the state vector for the descending agent:

$$\boldsymbol{s} = \begin{bmatrix} \tilde{\boldsymbol{x}}_a \\ \boldsymbol{v} \\ \dot{\boldsymbol{v}} \end{bmatrix}. \tag{4.14}$$

The reason behind controlling the yaw angle towards a specific value was mainly to ensure that the body of the drone, and thus the camera mounted on the drone, would not be rotated with respect to the landing platform. This ability would be especially advantageous for later augmentations of the project, where the down-facing camera on the drone would be used by an external perception module to estimate the quadrotor's position relative to the landing platform.

Note that since the yaw angle now was included in the state vector, there was no need to apply the transformation (4.7) in the same manner as for the hovering agent. The descending agent would inhabit enough information regarding its rotation relative to the world frame to calculate this transformation itself using the actor and critic networks.

#### 4.2.1.3 State normalization

Deep learning algorithms often benefit from data normalization, especially when the different features have varying scales [72]. In both the hovering agent and the descending agent the state vectors were subject to normalization such that every variable spanned a specific interval, aiming to boost training speed and also performance of the solutions.

In the case of this thesis, both agents were normalized such that each state variable spanned the range $[-1, 1]$. If a state variable $s$ was originally given in the range $[s_{\texttt{min}}, s_{\texttt{max}}]$ and was subject to normalization to the range $[a, b]$, the normalized state $s_{\texttt{norm}}$ would be given as

$$s_{\texttt{norm}} = (b - a)\frac{s - s_{\texttt{min}}}{s_{\texttt{max}} - s_{\texttt{min}}} + a. \tag{4.15}$$

Since $x$, $y$, $z$ and $\psi$ were to be controlled by the two agents, they were naturally added to the respective state vectors. However, none of these values are restricted to any bounds due to the open environment used, as the linear and angular position of a drone may range between any numbers the environment allows. The agents would, in practice, have designated regions to work in, preferably in the proximity of their setpoints. As such, boundary values were defined for $x$, $y$, $z$ and $\psi$ in the environment. Their values depended on the agent and is presented in greater detail in Section 5.1.

In the case of the translational velocities and accelerations that had no preset boundaries, tests were conducted for obtaining the minimum and maximum that the plant could acquire for these quantities could in the environment. The following values were obtained from several test runs for each variable:

$$\begin{aligned} -1.50\,m/s \leq v_x \leq 1.50\,m/s \\ -1.55\,m/s \leq v_y \leq 1.45\,m/s \\ -0.55\,m/s \leq v_z \leq 0.55\,m/s \\ -0.3\,g \leq a_x \leq 0.3\,g \\ -0.3\,g \leq a_y \leq 0.3\,g \\ 0.8\,g \leq a_z \leq 1.2\,g \end{aligned}$$

These values were subsequently used for normalizing the remaining states in $\boldsymbol{s}$ in the hover and descend agent that were not manually bounded. Note that the translational acceleration in $z$ direction never exceeds the bound $[0.8, 1.2]$. This is justified by the fact that the inertia in $z$ is too large while the agent's control command is simultaneously not large enough to cause a greater change in the acceleration in $z$ compared to the gravitational force acting on the plant. Hence, the majority of the acceleration in this direction will be caused by counteracting the gravitational force generated by Earth.

### 4.2.2 Reward function

As discussed in Section 2.2.2, the reward function in reinforcement learning methods is fundamental for the agent to "understand" its objective. Hence, the agent's behavior is said to be shaped by this function. By convention, the reward function is not assumed to be a part of the agent but rather the environment, as shown in Figure 4.3.

Depending on the desired behavior, there are various options for designing a reward function aiming to control a quadrotor to a desired position. Since the ideal state of the drone includes driving the positional error towards zero, it was desirable to design a reward function that pays the agent dividends for accomplishing this. One such function is the boundary function, expressed as

$$\mathcal{R}(x) = \begin{cases} a, & \text{if } |x| < b. \\ 0, & \text{otherwise.} \end{cases} \tag{4.16}$$

where $a$ is the maximum value and $b$ is the boundary width from the center.

The literature suggests that an RL agent will struggle to converge towards optimal behavior if it is given sparse rewards, such as with the boundary function. This is mainly due to the agent's exploratory behavior and that there is no guarantee that it will encounter the high-reward areas in the environment. Additionally, using the boundary function will not guarantee that the quadrotor will hover at the exact setpoint specified, as it reaps the same rewards as long as it is inside the boundary. Hence, the agent has no way of distinguishing between mediocre behavior, where the drone is *in the proximity* of the setpoint, and desired behavior, where the drone is *at* the setpoint. Resultantly, a reward function increasing monotonically will guide the agent towards areas of the state space with even higher rewards.

#### 4.2.2.1 Hover

In light of the arguments presented, a function that increases monotonically towards a peak where the Euclidean norm of the positional error $||\tilde{\boldsymbol{x}}||$ equals 0 is desirable for the hovering agent. Given that $\tilde{\boldsymbol{x}} = \boldsymbol{R}_w^b \tilde{\boldsymbol{x}}^w$, the positional error norm can be rewritten in the following manner:

$$\begin{aligned} ||\tilde{\boldsymbol{x}}|| &= \tilde{\boldsymbol{x}}^\intercal \tilde{\boldsymbol{x}} \\ &= (\boldsymbol{R}_w^b \tilde{\boldsymbol{x}}^w)^\intercal \boldsymbol{R}_w^b \tilde{\boldsymbol{x}}^w \\ &= (\tilde{\boldsymbol{x}}^w)^\intercal (\boldsymbol{R}_w^b)^\intercal \boldsymbol{R}_w^b \tilde{\boldsymbol{x}}^w \\ &= (\tilde{\boldsymbol{x}}^w)^\intercal \tilde{\boldsymbol{x}}^w \\ &= ||\tilde{\boldsymbol{x}}^w|| \\ &= \sqrt{(x - p_{x,h})^2 + (y - p_{y,h})^2 + (z - p_{z,h})^2} \end{aligned}$$

Thus, $||\tilde{\boldsymbol{x}}||$ is equal to the Euclidean distance between the drone's position $\boldsymbol{x}$ and the setpoint $\boldsymbol{p}_h$, which serves as a reasonable quantity to optimize.

There are many functions with the attribute of increasing monotonically. In this thesis we propose the Gaussian distribution as a reward function that endorses small positional errors, namely

$$\mathcal{R}(\boldsymbol{s}) = ae^{-(||\tilde{\boldsymbol{x}}|| - \mu)^2 / 2\sigma^2}, \tag{4.17}$$

where $a$ is the peak amplitude, $\mu$ is the mean, $\sigma$ is the standard deviation and $e$ is the base of the natural logarithm.

There are several reasons behind this choice of reward function. Firstly, it is quite similar to the boundary function and relatively painless to tweak. A comparison between the boundary function and the Gaussian is given in Figure 4.7. Secondly, a reward function shaped as a Gaussian promotes convergence towards the peak of the bell curve, in this agent's case this is approaching the setpoint of hover. In addition, this reward function shape yields a larger reward area meaning the rewards are no longer sparse, which again facilitate faster learning.

Since the Gaussian is quite flat at the maximum, the agent may struggle to find the states where the reward is maximized fully. The flatness might therefore yield a behavior where

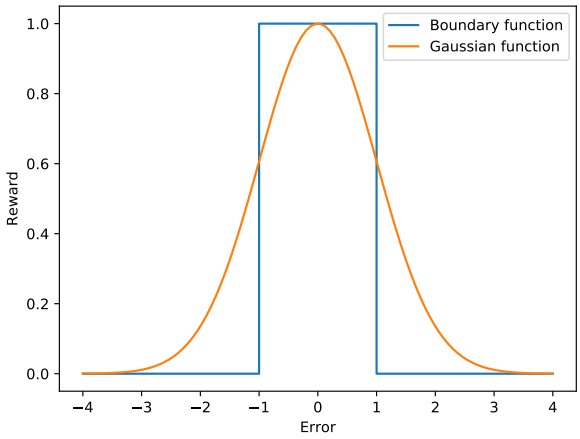



**Figure 4.7:** Comparison between a boundary function with $a = 1.0$ and $b = 1.0$, and a Gaussian distribution with $a = 0.0$, $\mu = 0.0$ and $\sigma = 1.0$. The figure illustrates that a smooth function such as the Gaussian allows the agent to experience small increments in rewards even when the error is high. This detail may greatly benefit the agent's convergence towards desired behavior.

the agent settles for a reward of 0.95 instead of 1.0. If the standard deviation of the reward function is sufficiently large, this difference in reward may translate to a consequential positional error. On the other hand, if the standard deviation is sufficiently small, the rewards may be experienced as sparse to the agent. As a means to mitigate this type of behavior where the agent would be "satisfied" with hovering e.g. 30 centimeters away from the setpoint, $\sigma$ would need to be chosen with care.

Many implementations use the Gaussian as a *base function*, meaning that this function communicates the main target of the agent. Using a base function in the reward supports the inclusion of additional terms that express sub-goals. These are often included to tweak the agent's behavior marginally. In this project, one such sub-goal was smoother control actuation. During development, it was noticed that the agent would lean towards giving maximum velocities when approaching the setpoint. No penalization on actuation would result in the agent converging towards a bang-bang protocol, since this approach would be optimal with only the Gaussian base function. When dealing with physical systems, as in this thesis, such behavior may cause wear and tear on the actuators while also consuming unnecessary power. Additionally, the drone may struggle to keep its roll and pitch angles small if the actuation is aggressively tuned, which may lead to a crash in the worst case. Resultantly, the base function was augmented with a term penalizing high velocity commands:

$$\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}) = ae^{-(||\tilde{\boldsymbol{x}}||-\mu)^2/2\sigma^2} - c_{||\boldsymbol{a}||}||\boldsymbol{a}||, \tag{4.18}$$

where $c_{||\boldsymbol{a}||}$ was a small constant.

It was also noticed during development that solely setting $\sigma$ to a small value was not

enough for the drone to stay at the setpoint. Lowering the standard deviation further would make the rewards too sparse and the agent would either learn very slowly or not converge towards a solution at all. Resultantly, an additional sub-goal was added to the reward function in order to convey more clearly to the agent when it is hovering well. If the quadrotor found itself within a very small sphere with radius $R$ from the setpoint, it would receive an additional reward of magnitude $A$ for every timestep it was inside this area. Hence, the reward function for the hovering agent was finally given as

$$\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}) = ae^{-(||\tilde{\boldsymbol{x}}||-\mu)^2/2\sigma^2} - c_{||\boldsymbol{a}||}||\boldsymbol{a}|| + \begin{cases} A, & \text{if } ||\tilde{\boldsymbol{x}}|| < R. \\ 0, & \text{otherwise.} \end{cases} \tag{4.19}$$

It could be argued that the inclusion of the last boundary term would yield a more sparse reward function and thus slow down learning. Recall that the Gaussian is set as a base for this reward function and will hence guide the agent towards this high-reward area, allowing the agent to explore this space more frequently than it would have with only a boundary function. This would also be shown in practice later, in Chapter 5. Note that no penalty was given when the agent exited the valid area of the environment, although the episode ended.

An important consideration to factor in when designing a reward function is how large the accumulated reward can become. Revisiting the cumulative discounted reward in (2.11), an upper bound of the value function $V(s)$ can be calculated if the maximum obtainable reward in each timestep, $r_{\texttt{max}}$, is known and the total numbers of steps $N$ goes towards infinity:

$$V(s) \leq \sum_{n=0}^{N} \gamma^n r_{\texttt{max}} \overset{N\to\infty}{=\!=} \frac{r_{\texttt{max}}}{1-\gamma}, \ \forall s \in \mathcal{S}. \tag{4.20}$$

With the reward in (4.19), $r_{\texttt{max}}$ is achieved when hovering at $||\tilde{\boldsymbol{x}}|| = \mu$ with $|\boldsymbol{a}| = 0$. As such, $r_{\texttt{max}} = a + A$ in this case. Setting $\gamma = 0.99$ yielded $V(s) \leq 100 \cdot (a + A)$. If the cumulative reward grows significantly large, it has the potential of threatening the numerical stability of the calculations and may yield very large network parameters which can, in turn, render slower training. Hence, $a$ and $A$ would have to be chosen wisely.

#### 4.2.2.2 Descend

The descending agent adopted many of the ideas from the hovering agent's reward function. Given (4.13), the pseudo-Euclidean distance of the augmented positional error is

$$||\tilde{\boldsymbol{x}}_a|| = \sqrt{(x - p_{x,d})^2 + (y - p_{y,d})^2 + (z - p_{z,d})^2 + (\psi - p_{\psi,d})^2}. \tag{4.21}$$

Motivated by the hover agent, the reward function of the descend agent, who was also tasked to control $\psi$, was given as

$$\mathcal{R}(\boldsymbol{s}, \boldsymbol{a}) = ae^{-(||\tilde{\boldsymbol{x}}_a||-\mu)^2/2\sigma^2} - c_{||\boldsymbol{a}||}||\boldsymbol{a}|| + \begin{cases} A, & \text{if } ||\tilde{\boldsymbol{x}}_a|| < R. \\ 0, & \text{otherwise.} \end{cases} \tag{4.22}$$

One subtle, though important difference to point out between the calculation of the reward for the hovering and descend agent follows the fact that the yaw angle does not have the same unit as the rest of the variables in the augmented positional error. The yaw angle is more prone to larger alterations in value between two time steps, due to the inertia of the drone and the nature of its movement. This would yield relatively large errors in $\psi$ even when the behavior was satisfactory. An error of 1 meter in the $x$ direction would yield the same reduction in reward as an error of 1 degree in yaw, although the latter was of lesser significance than the former.

It should be mentioned that, for both the hovering and descending agent, the positional error with respect to the setpoint was fed to the agents when calculating the appropriate action commands. As such, the specific values of $\boldsymbol{p}_h$ and $\boldsymbol{p}_d$ themselves were not of interest for the respective agents, but rather only the value of the error in each state controlled. As such, both agents were independent of the specific value of the setpoints. As a result, $\boldsymbol{p}_h$ and $\boldsymbol{p}_d$ could be altered after completed training, as long as the boundary values were perturbed accordingly.

### 4.2.3 Case study using perception estimate

In the preceding sections, it has been assumed that the position of the quadrotor is given by the Gazebo simulator through ROS topics broadcasting the drone's ground truth pose in the simulator. Using these topics it was possible to extract $\tilde{\boldsymbol{x}}^w$ and $\psi$. While this works in the simulated environment, it is advantageous for the flexibility of the system to be able to estimate its pose independently of the ground truth topic, so the transition from simulator to real-life settings is more seamless. This has been done in a cooperating master's thesis [39], where the bottom camera of the AR.Drone 2.0 was used in order to conduct pose estimation through image streams, and broadcast this estimate to a dedicated ROS topic. As such, the $x$, $y$, $z$ and $\psi$ values of the drone relative to the helipad could be obtained for pose estimation, independently of the ground truth estimate. Note that the linear positions in addition to the yaw angle were the only entities that were estimated through the ground truth ROS topic updated from the Gazebo simulator. Since the velocities and accelerations were being estimated through the internal sensors of the drone, these values would be available both in simulations and real-life testing. As such, there was no need to alter any logic regarding these quantities.

There were four elements in the estimation process that ensured that the drone had an estimate of $x$, $y$, $z$ and $\psi$ at all times. Three of the methods used traditional computer vision schemes when the helipad was visible to the quadrotor's camera. The last method exploited the IMU sensors and executes dead-reckoning to reason its position.

The computer vision methods based themselves on the geometric properties of the landing platform, depicted in Figure 4.8, which was designed and constructed such that they could exploit geometric features of the helipad for efficient, accurate and robust estimation. The helipad was designed with three distinct colors, namely green, orange and white. As a result, the three methods were distinguished between what they would detect and base their estimate on. Each method used the outer rim of the platform, the orange arrow and the inner corners of the H, respectively. First, the three colors were distinguished from

each other using segmentation. Further, points on the outer rim of the green circle were calculated as well as the orange arrow and the inner corners of the H. Thanks to the design of the helipad, these elements could be identified using edge detection [73] and corner detection [74]. This laid the foundation for the three methods used to calculate the position and yaw angle.

- *Ellipse*: Fits all points of the outer rim of the green circle to an ellipse. The center of the ellipse constitutes the center of the helipad and the largest radius in the ellipse corresponds to the radius of the platform. This method has no way of estimating the yaw angle.
- *Arrow*: The centroid of the H is calculated through image moments [75], and the magnitude of the vector from the centroid to the orange arrow is used to calculate the radius of the helipad. This can be done since the relationship between this distance and the radius of the helipad is known. Further, the angle of this vector constitutes the yaw angle.
- *Corners*: The four inner corners of the H is used to obtain center of the landing platform, its radius and the yaw angle.

Lastly, the perception module included dead-reckoning logic such that it used the drone's internal sensors to reason for its position and rotation when the helipad was out of sight. This allowed an estimate of the quadrotor's position even when it could not detect the platform with imagery. It was worth noting that the dead-reckoning logic assumed that the drone had an initial estimate for the platform's position, thus requiring a calibration process before takeoff.

The estimate returned from the perception module depended on the distance between drone and platform, and what was in the camera's field of view at a given time step. If the helipad was not in the field of view, the dead-reckoning module would activate. For larger distances where the platform is visible, the estimator would calculate the quadrotor's position relative to the platform using the ellipse method. When the drone approached the helipad, its position and yaw angle were estimated through the arrow method. If sufficiently close, it would use the corners of the H. The first three modules ran at 10 Hz, while the dead-reckoning module ran at 100 Hz. This resulted in a ROS topic with a frequency of 100 Hz, where the estimate would stay constant for 10 samples if the dead-reckoning module was not used.

The fact that the yaw could be estimated was of high importance for robustness of the solution. Despite the fact that the internal sensors of the drone offered an estimate for $\psi$, that estimate was prone to drifting due to how the IMU conducts this estimation process. As a result, the perception module allowed more accurate prediction of the yaw angle and could help substantially when the descending agent was to align its yaw angle with the helipad.

Since the thesis was originally aimed to apply the presented results to the Revolt vessel, a realistic scenario with surroundings resembling this real-life setting was designed. Appropriately, the simulator's world was augmented from previous training, where a vessel in an ocean with dynamic waves was adopted. A figure illustrating this is given in Figure 4.9


**Figure 4.8:** The landing platform, also referred to as helipad, seen from a bird's-eye view. The helipad was designed such that the perception module would have the required amount of features to work with for the computer vision modules. The H is of size 0.20 m $\times$ 0.28 m. The distance from the center of the helipad to the arrow tip is 0.30 m, while the radii of the orange and green circle are 0.26 m and 0.40 m, respectively. Figure courtesy of [39].

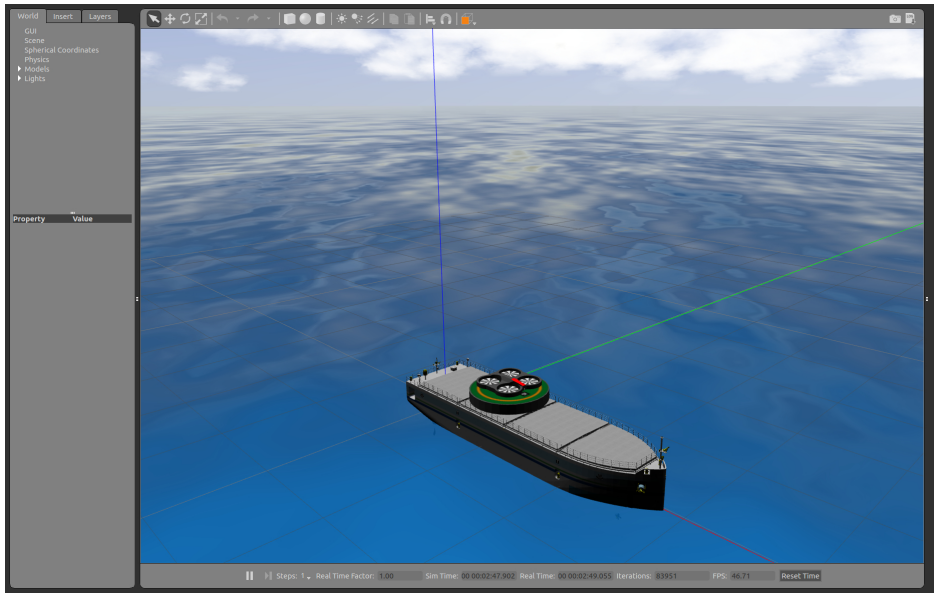



**Figure 4.9:** The augmented simulation environment modeled in Gazebo when using the perception estimates. Figure courtesy of [39].

Chapter 5

# Results

## 5.1 Training framework

For the training stage of this thesis to be conducted, several parameters and values were needed to be set. Both the hovering and descending agent incorporated the same parameters and values for their respective DDPG solutions. Emanating from Algorithm 3, the hyperparameters to the algorithm were therefore set consistently with the original DDPG paper [30], and are summarized in Table 5.1. Both networks adopted Adam, a widely used optimizer within DL applications that extends the functionality of the stochastic gradient descent [47]. In order to avoid excessively large gradient updates to the networks due to a high Q-value outputted from the critic in addition to aggressive action commands from the actor, the output layers in both these entities were initialized using a random uniform distribution spanning the range $[-0.003, 0.003]$.

| **Parameter** | **Value** |
|---|---|
| Batch size | 64 |
| Replay buffer size | $10^6$ |
| Discount factor $\gamma$ | 0.99 |
| Polyak parameter $\tau$ | $10^{-3}$ |
| Actor learning rate $\alpha_\mu$ | $10^{-4}$ |
| Critic learning rate $\alpha_Q$ | $10^{-3}$ |

**Table 5.1:** Table encapsulating the parameters of the DDPG solution for both the hover and descend agent.

The literature suggests that a very large batch size for training the networks would yield poor generalization, while a small value would lead to inefficient training [76, 77]. As a

result, the batch size in this thesis was set to 64 where each batch was sampled from a replay buffer incorporating $10^6$ transitions. The replay buffer acted like a FIFO priority queue, where the oldest transitions were substituted when the buffer was filled. This was done since the agent would incorporate more high-quality transitions as time passes, while the first transitions would mostly be influenced by exploratory actions and low returns. The Ornstein-Uhlenbeck process given in (2.46) affected all states independently, with $\mu = 0$, $\theta = 0.15$ and $\sigma = 0.20$.

Ultimately, the goal was to extract a policy $\mu(\boldsymbol{s})$ from training that exhibited satisfying behavior for the objective of this thesis. As mentioned, this was conducted through the DDPG algorithm. For each episode during training for both the hovering and descending problem, the agents were initialized at a randomized positions inside the valid area of each specific training environment. The hovering agent had randomized its $x$, $y$ and $z$ components, while the descending agent also randomized $\psi$. Both agents were able to execute up to 50 steps before the episode terminated. If an agent exited the valid area, the episode would end prematurely and the agent would receive no further rewards. Thanks to how the reward function was shaped, both agents would eventually realize that they do not benefit from exiting the environment early, because they would miss out on additional rewards. The training was concluded when the agent had experienced in total 50.000 training steps across all episodes.

Both controllers developed were set to operate at a frequency of 3 Hz between each action command to the velocity controller. The extraction of this number was also a case of trial-and-error. Using higher frequencies could, during operation, yield smoother behavior and a more responsive agent. Albeit, it was observed that the instability during training increased with the controller rate, where the agents would spend excessive amount of time to train or even not converge. Although no extensive research was conducted to exploit the reason as to why this was the case, it can be argued that the replay buffer was being filled rapidly with relatively similar transitions when the frequency was large. This could reduce the informative value between transitions and thus confuse the agent with respect to its objective.

In the work conducted the point of hover the hovering agent sought to converge towards was 2 meters above the origin in the simulator, namely

$$\boldsymbol{p}_h = \begin{bmatrix} 0.0 \\ 0.0 \\ 2.0 \end{bmatrix}. \tag{5.1}$$

It was logical to assume that the drone would often be in immediate proximity to the set-point. Resultantly, the positional error would be restricted to predefined boundary values during training, meaning the agent would terminate the episode prematurely if it found itself outside these bounds. In accordance with Section 4.2.1, the boundaries of $\tilde{\boldsymbol{x}}$ in the hovering agent was set to the following:

$$\begin{aligned} -2.0\,m \leq \tilde{x}_1 \leq 2.0\,m \\ -2.0\,m \leq \tilde{x}_2 \leq 2.0\,m \\ 1.0\,m \leq \tilde{x}_3 \leq 3.0\,m \end{aligned}$$

As such, the hovering agent was restricted to a 3-dimensional box of $4m \times 4m \times 2m$. The boundaries were set with the realistic real-life scenarios in mind, allowing extensive flexibility when the agent was far from the setpoint in the horizontal plane. It was argued that the agent would most likely find itself at varying heights from the ground, both below and above the setpoint, and the boundary was set accordingly. These state restrictions were introduced mainly to streamline training by avoiding training episodes where the agent would drift further and further away from the goal and explore states that would be inessential to the agent's objective. Such behavior would fill the replay buffer with numerous transitions that would confuse the agent and resultantly slow down or even hinder convergence.

Further, the setpoint of the descending agent, given in (4.12) could be expressed as

$$\boldsymbol{p}_d = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.3 \\ 0.0 \end{bmatrix}. \tag{5.2}$$

The reasoning behind choosing 0.3 meters above the landing platform as the desired height for the descending agent was that this distance rendered the drone close enough to the platform initiate the landing phase, while simultaneously being far enough for the chassis of the quadrotor not to hit the platform or the ground.

Since it was assumed that the descending agent would directly succeed the hovering agent during the landing mission, the restrictions to the former agent's positional error were slightly altered in order to adapt to the new environment and objective. Assuming that the hovering agent was successful in its mission to control the quadrotor towards the hovering point $\boldsymbol{p}_h$, the boundaries of the descending agent were set appropriately. As a result, the augmented positional error $\tilde{\boldsymbol{x}}_a$ was confined in to the following values during training:

$$\begin{aligned} -0.15\,m \leq \tilde{x}_1 &\leq 0.15\,m \\ -0.15\,m \leq \tilde{x}_2 &\leq 0.15\,m \\ 0.0\,m \leq \tilde{x}_3 &\leq 2.5\,m \\ -20.0^\circ \leq \tilde{\psi} &\leq 20.0^\circ \end{aligned}$$

As for the reward functions used for the two agents, values for the parameters in (4.19) and (4.22) had to be defined. For the hovering agent, the basis for choosing the parameters for the base Gaussian was the largest Euclidean distance that was possible to obtain using the framework chosen. Given the boundaries set for the hovering agent, the largest Euclidean distance the hovering agent can find itself from $\boldsymbol{p}_h$ was

$$\begin{aligned} ||\tilde{\boldsymbol{x}}||_{\max} &= \sqrt{(x - p_{x,h})^2 + (y - p_{y,h})^2 + (z - p_{z,h})^2} \\ &= \sqrt{(2m - 0m)^2 + (2m - 0m)^2 + (3m - 2m)^2} \\ &= 3\,m. \end{aligned}$$

This value served as a guideline for choosing appropriate parameters for the Gaussian of the hover reward function. Since the goal was to minimize the positional error, it fell

natural to set $\mu$ as 0. Further, the choice of $a$ only affected the maximum reward that was possible to obtain, given in (4.21). In order to evade a situation leading to numerical instability for the agents, $a$ was set to 1. Lastly, $\sigma$, $c_{||\boldsymbol{a}||}$, $A$ and $R$ had to be identified through trial and error. Summarized, the following values were given for the hovering agent's reward function:

$$\begin{aligned} a &= 1 \\ \mu &= 0 \\ \sigma &= \sqrt{0.025} \\ c_{||\boldsymbol{a}||} &= 0.08 \\ A &= 0.3 \\ R &= 0.03 \end{aligned}$$

The choice of parameters for the descending agent was conducted in the same manner as previously presented with the hovering agent. Firstly, in order to mitigate the scaling problem between the linear positions and the yaw angle introduced in Section 4.2.2.2, all errors in $\tilde{\boldsymbol{x}}_a$ were normalized before calculating the reward function, so they spanned the range of $[-1, 1]$. Since $\boldsymbol{p}_d$ was known and the boundaries of every variable in $\tilde{\boldsymbol{x}}_a$ was defined, it was possible to calculate the maximum and minimum error achievable and normalize in the same manner as in Section 4.2.1.3 based on this. Resultantly, the largest pseudo-Euclidean distance was calculated to be

$$\begin{aligned} ||\tilde{\boldsymbol{x}}_a|| &= \sqrt{(x - p_{x,d})^2 + (y - p_{y,d})^2 + (z - p_{z,d})^2 + (\psi - p_{\psi,d})^2} \\ &= \sqrt{1^2 + 1^2 + 1^2 + 1^2} \\ &= 2. \end{aligned} \tag{5.3}$$

This value would naturally deviate from the maximum Euclidean distance calculated for the hovering agent, which had to be taken into account when choosing the value for the width of the Gaussian curve. As opposed to $||\tilde{\boldsymbol{x}}||$ for the hovering agent, the unit of $||\tilde{\boldsymbol{x}}_a||$ was not meters, since the error in yaw was expressed in degrees. When choosing the parameter values for (4.22), it fell naturally to adopt $a = 1$ from the hovering agent. The descending agent also wished to drive $||\tilde{\boldsymbol{x}}_a||$ towards 0, so $\mu$ was set to 0, similar to the hover agent. Since the pseudo-Euclidean distance for the descending agent would in general be smaller than the Euclidean distance for the hovering agent due to the normalization, a smaller value for $\sigma$ in (4.22) was chosen for the descending agent in order to counteract this. Simultaneously, since the descend agent was tasked to controlling the yaw angle as well as the linear positions, it was more challenging to reach this area. Therefore, the radius for the additional reward was set higher. The values of $\sigma$, $c_{||\boldsymbol{a}||}$, $A$ and $R$ were found through trial and error and the final values for the reward function parameters for

the descend agent was hence summarized as:

$$\begin{aligned} a &= 1 \\ \mu &= 0 \\ \sigma &= \sqrt{0.015} \\ c_{||\boldsymbol{a}||} &= 0.08 \\ A &= 0.3 \\ R &= 0.05 \end{aligned}$$

It is worth noting that no exhaustive search was conducted to find optimal values for neither the hovering nor the descending agent, which could potentially lead to more optimal solutions. However, these values lead to the agents exhibiting satisfying behavior, which was sufficient for the scope of this thesis.

Recall that the velocity commands presented in (3.1) were all bounded in the range $[-1, 1]$, where the lower bound and upper bound would yield maximum velocity in the positive and negative direction, respectively. The directions were given in the body frame and are determined by the right-hand rule as per Figure 2.9. As such, (4.1) in our framework was

$$\begin{bmatrix} -1.0 \\ -1.0 \\ -1.0 \end{bmatrix} \leq \boldsymbol{a} \leq \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} \tag{5.4}$$

for the hovering agent and

$$\begin{bmatrix} -1.0 \\ -1.0 \\ -1.0 \\ -1.0 \end{bmatrix} \leq \boldsymbol{a} \leq \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} \tag{5.5}$$

for the descending agent.

Since the activation function of the last layer in the actor was the hyperbolic tangent and the saturating limits of the quadrotor was in the range $[-1, 1]$, the values for the linear transformation of the raw action in (4.2) were resultantly as

$$\boldsymbol{a}_{\texttt{scale}} = 1, \tag{5.6}$$

$$\boldsymbol{a}_{\texttt{shift}} = 0. \tag{5.7}$$

$\boldsymbol{a}_{\texttt{shift}}$ was set to 0 mainly due to the velocity commands having no bias towards any direction.

## 5.2 Ground truth results

The following results were extracted by using the ground truth estimate given by the Gazebo simulator, offering access to the exact values of all the necessary states. The

estimate had no internal drift nor biases, and yielded accurate values within the simulated environment.

Note that the subsequent sections present the results obtained by fragmenting the landing mission, meaning the full landing mission was split hovering and descend. As a result, the performance of the hovering and descending agent are decomposed into two parts for analytical purposes and transparency in this section.

### 5.2.1 Hover

For the hovering aspect of the landing mission the training progression is depicted in Figure 5.1. Figure 5.1a portrays the episodic reward obtained by the agent in each episode, while Figure 5.1b shows the corresponding number of steps conducted by the agent in each episode. Recall that the agent is forced to exit an episode after 50 steps, while it exits automatically if it finds itself outside the boundary values before all steps in an episode is completed.

It is evident that the agent spends the first portion of training exploring the environment, where it seems to prematurely exit the environment quite frequently. Across the 1925 episodes the agent conducted in total, it experienced an upswing in the number of executed steps in each episode after approximately 1050 episodes, or 5700 steps. This is closely related to the fact that it also observes a notable increase in obtained rewards, where a steep upsurge is observed between episode number 1050 and 1200 in Figure 5.1a. Assuming that approximately 1200 episodes, or 11.000 steps, passed before it could be characterized as the agent had started learning the target. Using 3 Hz sampling rate for the environment and controller, this process took roughly 61 minutes real-time. However, the Gazebo simulator managed to run faster than real-time while still maintaining the same sampling frequency relative to the run-time. The simulator managed to speed up the simulation relative to real-time by a factor of 8, meaning that the agent in fact learned after less than 8 minutes.

Considering the relative complexity of the environment with two sub-goals in the reward function and a quite large state vector, this can be deemed a quite sufficient result with acceptable training time. The full training process was concluded after the 50.000 steps were conducted, translating to 35 minutes.

Given the reward function (4.19) the maximum reward obtainable in an episode is 65. However, this value assumes that $||\boldsymbol{a}|| = 0$ in addition to the agent spawning and staying at $\boldsymbol{p}_h$ from start to finish. The average episodic reward after the agent started learning was found to be $24.9$, which is quite robust considering that it would rarely start close enough to $\boldsymbol{p}_h$ to reap any reward of significance in the first steps of an episode. Additionally, it was unrealistic to assume that $||\boldsymbol{a}||$ would be close to zero, since the agent preferred to be penalized for large action commands as long as it arrived at the setpoint quickly, since this would ultimately yield the highest total reward.

Resulting from the training process, the hovering agent was tested with respect to how its response was when starting at the edges of the valid area. With the hovering agent's area of use in mind, five tests were conducted where the agent started above the setpoint for all tests, with varying initial positions in the horizontal plane. The drone's response for all

three controlled states are illustrated in Figure 5.2 with five distinct initial positions in the world frame. Figure 5.3 depicts the same tests, but from a bird's-eye view as a means of illustrating how the response in the horizontal plane was.

As the figures show, the hovering agent was capable of converging towards the setpoint quite efficiently, evidently balancing between convergence towards $\boldsymbol{p}_h$ and damped controller input originating from the penalization of action magnitude[1]. Based on the reward accumulated during training given by Figure 5.1a in coherence with the responses in Figure 5.2 and Figure 5.3, the agent seemed to find a satisfactory middle ground between the goal of hovering at the given setpoint and maintaining composed actuation. The tests show that, with varying initial position, the hovering is able to converge towards its setpoint between 20 and 30 time steps, translating to maximum 10 seconds real-time.

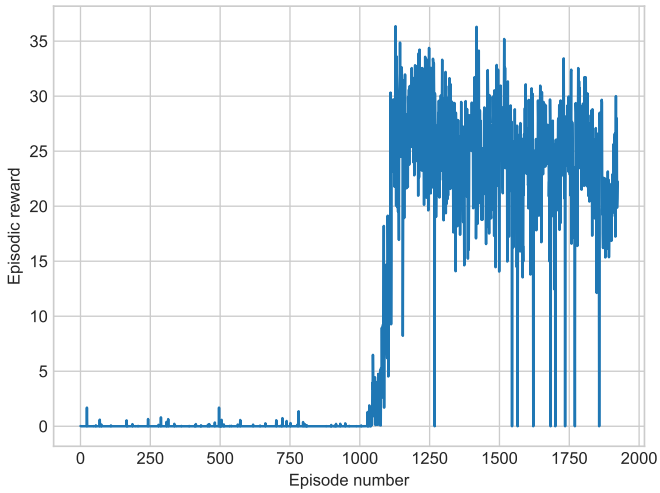
The plots in Figure 5.3 establish that the agent, for the most part, learned that the fastest path towards the goal in the horizontal plane leading to maximized rewards was a straight line from the initial position, which theory supports. However, this was not the case for all initial positions. Although the agent eventually arrived at the setpoint, Figure 5.3c illustrates that the drone occasionally deviated from the optimal straight-line behavior. The fluctuating behavior exhibited in $z$ in Figure 5.2a, Figure 5.2b, Figure 5.2c and Figure 5.2d was also not optimal. Furthermore, the behavior where the agent deviates from $x = y = 0$ in Figure 5.2e was also a noteworthy shortcoming, where the optimal solution given that initial position would naturally be actuation such that the horizontal position was kept while the height decreased towards the desired value in $z$. For all of these shortcomings it can be argued that, with an increased number of training steps, the agent could fine-tune its behavior and diminish such suboptimal performance. Increased training steps would naturally lead to more training, which would lead to the agent explore actions that would improve the rewards obtained and resultantly improve its behavior. However, due to time constraints further training was not conducted.

When it comes to the steady-state hovering, where the agent has reached the setpoint and is tasked to remain there, Figure 5.4 illustrates the agent's performance. It can be observed from the figure that the total positional error $||\tilde{\boldsymbol{x}}||$ has a mean at around $7.5$ centimeters and a standard deviation of approximately $2.5$ centimeters. The components contributing to this error is also illustrated in the figure. Relative to the mean and standard deviation of the error in $x$ and $y$, the error in $z$ is quite small. Although this result can emanate from a range of reasons, it is worth recalling from Section 3.2.2 that the drone inhabited sensors that directly measures $z$, while it has to estimate $x$ and $y$ through integration of IMU measurements. Also recall that the horizontal position elements for this particular drone drift in a much larger degree, as illustrated in Figure 3.3. These reasons may justify the fact that the hovering agent performs significantly better in controlling $z$ relative to $x$ and $y$.

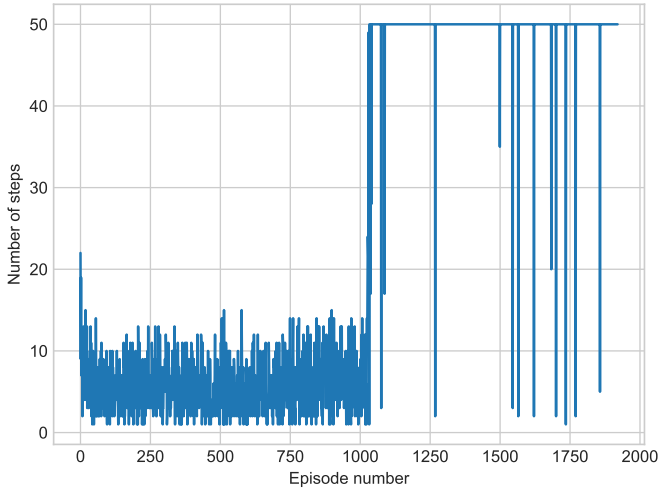
Although a DRL based method such as DDPG should be able to generalize such that it does no longer need to be within the bounds of the training environment during testing,

[1]Despite it being evident from both the rewards in addition to observed behavior that the agent avoided excessive control inputs, extensive analysis of the actuation fed to the velocity controller for both the hovering and descending agent was omitted due to time constraints.

it was concluded that this behavior would only be a bonus and not a requirement. The reasoning behind this was that for the scope of the objective, it was assumed that the drone was placed sufficiently close to the landing platform and such behavior was thus not strictly necessary.

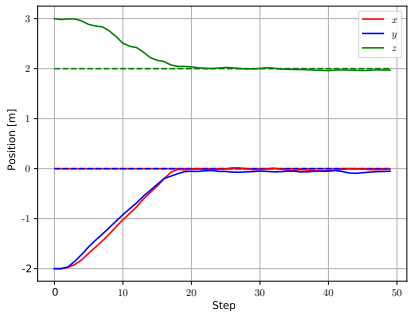



**(a)** The reward the hovering agent accumulated over each episode during training, where the reward was given as (4.19). The reason behind the rewards close to 0 after the agent started exhibiting learning behavior is probably due to the fact that the agent is initialized close to the boundary and the exploration policy pushes it outside immediately.

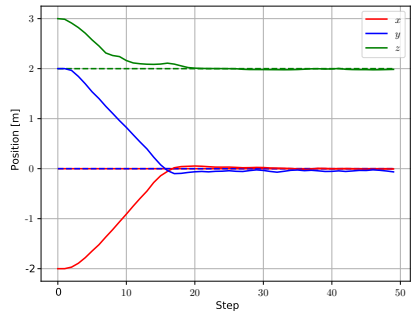



**(b)** The number of steps per episode during training of the hovering agent. One can clearly see the correlation between the episodic reward in Figure 5.1a and the number of steps per episode, where a low episodic reward corresponded to an episode with a low number of steps.

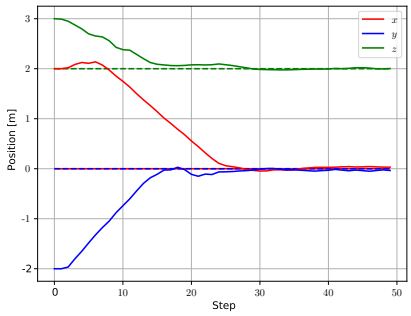
**Figure 5.1:** The reward and number of steps for each episode in the hovering agent's training phase.



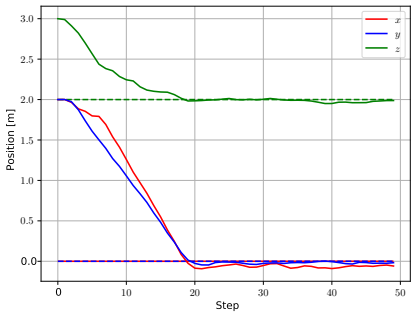

**(a)** Initial position $\boldsymbol{x}^w = [-2.0, -2.0, 3.0]^\intercal$



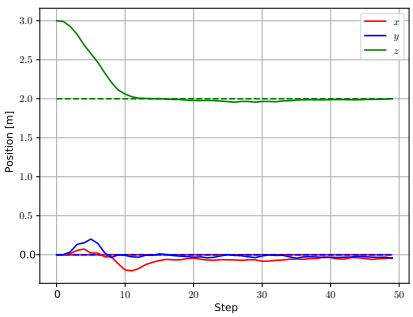

**(b)** Initial position $\boldsymbol{x}^w = [-2.0, 2.0, 3.0]^\intercal$




**(c)** Initial position $\boldsymbol{x}^w = [2.0, -2.0, 3.0]^\intercal$

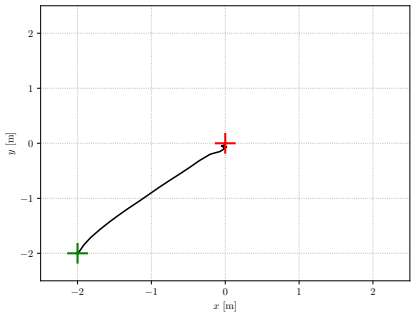



**(d)** Initial position $\boldsymbol{x}^w = [2.0, 2.0, 3.0]^\intercal$

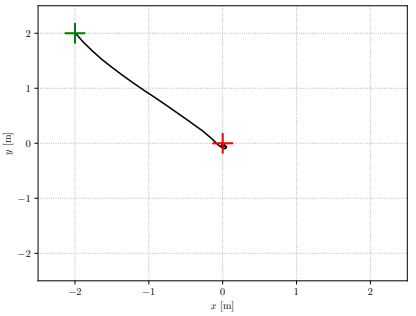



**(e)** Initial position $\boldsymbol{x}^w = [0.0, 0.0, 3.0]^\intercal$

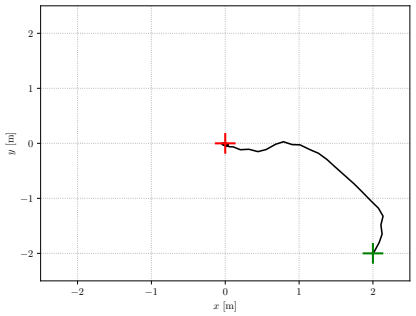
**Figure 5.2:** Five tests of how the hovering agent approaches the setpoint, in dotted lines, with varying initial positions. The horizontal axes illustrate time steps, while the vertical axes constitute the drone's position in the three dimensions.



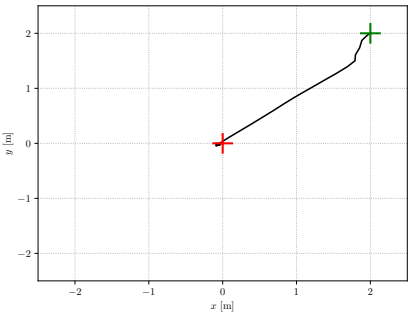

**(a)** Initial position $\boldsymbol{x}^w = [-2.0, -2.0, 3.0]^\intercal$



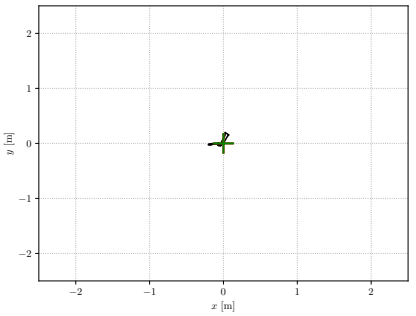

**(b)** Initial position $\boldsymbol{x}^w = [-2.0, 2.0, 3.0]^\intercal$




**(c)** Initial position $\boldsymbol{x}^w = [2.0, -2.0, 3.0]^\intercal$




**(d)** Initial position $\boldsymbol{x}^w = [2.0, 2.0, 3.0]^\intercal$




**(e)** Initial position $\boldsymbol{x}^w = [0.0, 0.0, 3.0]^\intercal$

**Figure 5.3:** A bird's-eye view of the tests illustrated in Figure 5.2. The green and red crosses constitute the starting and finishing positions, respectively. The unit of all axes are given in meters.

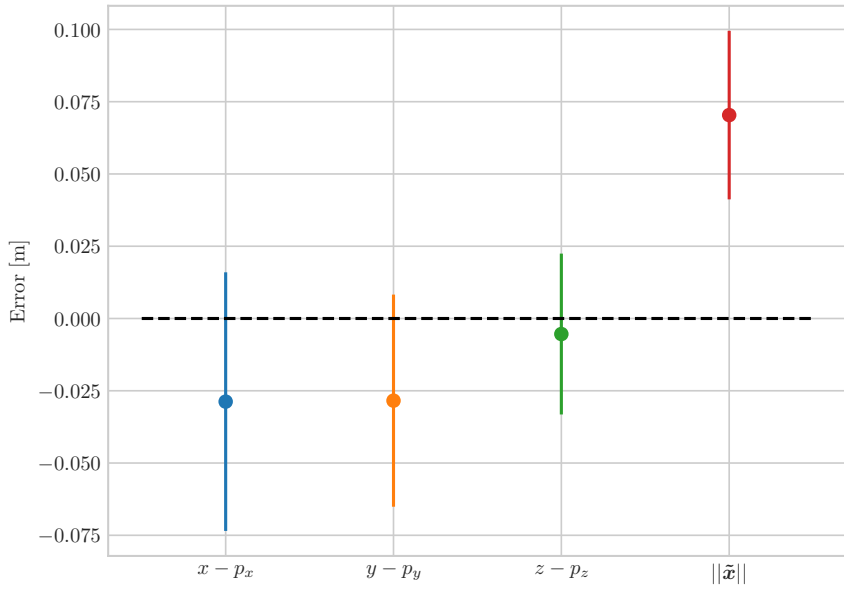



**Figure 5.4:** Means and standard deviations of the drone's linear position and positional error relative to the hovering setpoint. The errors are given in meters and are calculated under the assumption that the quadrotor is within close proximity of its setpoint. The results portrayed were subject to the same initial positions as in Figure 5.2, where the agent was spawned at each initial position 20 times. This resulted in 100 independent runs.

### 5.2.2 Descend

Through simulations it was observed that, despite having relatively similar environments, the training phase of the descending agent deviated from the hovering agent, especially when the agent started to learn its objective. Recall that the valid area of the descending agent was substantially stricter compared to the hovering agent due to the predefined boundaries. This was especially emphatic in the horizontal plane. By observation and analysis during the training process, it became evident that no more than one single aggressive action in the horizontal direction was needed for the agent to exit the boundaries of $x$ or $y$. Recall that the actions calculated by the agent during training were affected by exploration noise, as described by (2.46). Considering that $\boldsymbol{a}$ was not not being limited by any saturation limits to ensure full flexibility for the drone, the noise component contributed to excessive control signals to the velocity commands, resulting in the agent finding itself outside the valid area more often. Accordingly, the descending agent was prone to exiting the valid area more frequently than the hovering agent. This turned out to be the case, as shown in Figure 5.5.

Although the agent experienced an upswing in rewards similar to the hovering agent, the figure shows that it still struggled to stay within the valid area in the environment after approximately 2700 episodes or 4200 steps. After the agent experienced this upswing, it went on to conduct approximately 1500 more episodes. Out of these, it was calculated that 1267 episodes ended prematurely due to the agent exiting the valid area, meaning merely 233 episodes were conducted with 50 steps through the entire training process. Considering the total number of episodes in the training process, this is a substantially reduced number. In comparison, the hovering agent exited the valid area only 10 times after it had started learning.

If the episodes where the agent exited rather immediately were omitted, it can be argued that the agent had started learning its objective after approximately 2800 episodes, or 17.000 steps. This translated to 95 minutes real-time and 12 minutes with the speed-up from the simulator. Similarly to the hovering agent, the training process ended after approximately 35 minutes when all 50.000 steps were completed.

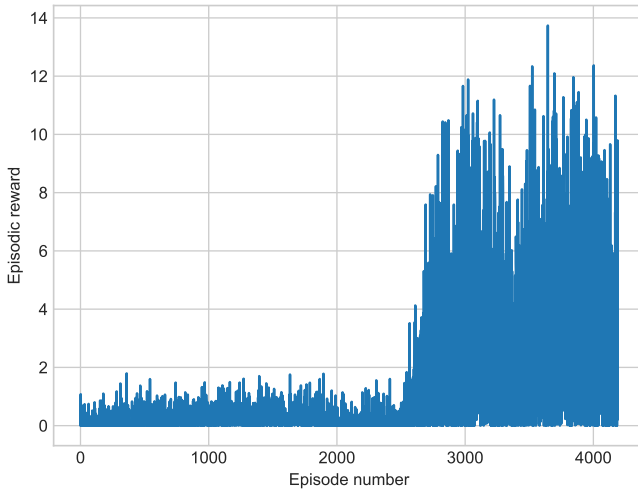
After the agent started learning its objective it would go on to average an episodic reward of 6.8 for the episodes that did not end prematurely. Although the descending agent's maximum possible reward was 65, the same arguments for not reaching this number applied as for the hovering agent. To obtain the maximum reward the agent had to start at $\boldsymbol{p}_d$ and would require no actuation, which is highly unlikely. Although this can be viewed as a small number considering that the maximum reward obtainable is 65, recall that that the yaw angle was contributing significantly to increase the augmented positional error. In addition, $\sigma$ in the descending agent's reward function was lower relative to the hovering agent, supporting the fact that that it could be more difficult for the descending agent to reach this area. Considering these additional alterations relative to the hovering agent, this was considered an acceptable result for the training process of the descending agent.

Figure 5.6 shows that the descending agent mitigated the hovering agents behavior from Figure 5.2e, where the hovering agent experienced deviations in $x$ and $y$ when $\boldsymbol{p}_h$ was placed directly below the initial position. Albeit this behavior presented by the hovering

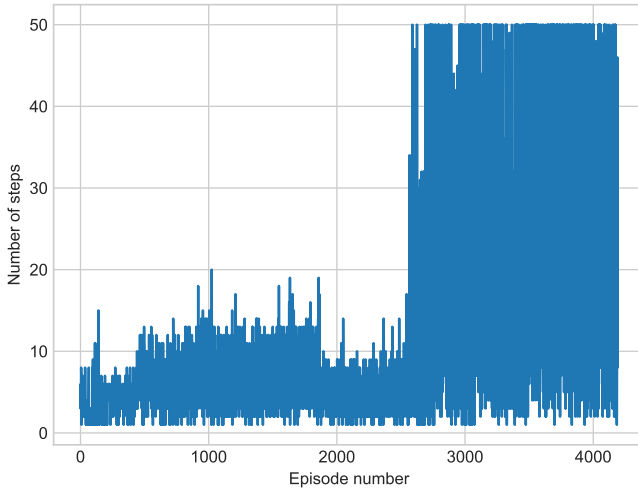
agent was not a big discrepancy, it was considered advantageous that the descending agent kept the values of $x$ and $y$ close to zero as it was approaching its setpoint. Further, the descending agent exhibited satisfying behavior in relation to controlling all four states towards their desired values, as Figure 5.6 shows. Initialized $x$, $y$, $z$ and $\psi$ at their boundary values set for the training of the descending agent, the drone managed to converge towards its setpoint $\boldsymbol{p}_d$ without fail for the eight tests that Figure 5.6 portrays. The tests show that the descending agent, with varying initial position and yaw angle, never uses more than 40 steps, or 14 seconds, to arrive sufficiently close to the desired setpoint.

There were cases where the descending agent would occasionally overshoot or keep a non-zero error in the horizontal directions, especially manifested in Figure 5.6c and Figure 5.6d. In the former, $y$ overshoots approximately $20$ centimeters in positive direction. In the latter, the agent struggles to decrease the error in $y$. It can also be seen that the yaw angle struggles with fluctuating behavior which may cause wear and tear on the propellers of the plant. This suggests that the penalty on actuation imposed by (4.22) needs to be adjusted for the yaw component of the action command, or even add an additional penalty where the difference in velocity commands between time steps is considered. However, since the mission of descending was aimed to take no more than 50 steps, or 17 seconds with the 3 Hz controller, it was concluded that the fluctuations in $\psi$ could be endured for the scope of this thesis.

Emanating from the fact that the tests suggested that the agent managed to reach its setpoint irrespective of the starting position, it was of interest to see how accurate this agent is when it reaches its setpoint. Figure 5.7 illustrates the means and standard deviations of the four controlled states, in addition to the positional and augmented positional error. The linear position errors and positional error are given in meters. The yaw error is given in radians and the augmented positional error has no SI unit, see (5.3). The results constitute the behavior of the agent when it arrived at steady-state, i.e. in proximity to $\boldsymbol{p}_d$, where the initial positions in Figure 5.6 were repeated 15 times, resulting in 120 independent runs. The agent converged towards its setpoint for all of these runs. Similar to the results from the hovering agent, the descending agent excelled notably in controlling $z$, where the mean is at $0.4$ centimeters with standard deviation at $1$ centimeter. This particular agent seems to struggle with centering both horizontal elements towards zero, although these errors do not seem to exceed $8$ centimeters for the 120 tests conducted. As with the hovering agent, this result is most likely connected with the fact that the agent drifts more horizontally, while the built-in velocity controller controls $z$ well. The yaw angle is centered at approximately 0.025 radians, or 1.4 degrees, and has a standard deviation of merely 1.5 degrees. The positional error has a mean and standard deviation of 7.0 centimeters and 0.2 centimeters, respectively. The positional error and yaw error suggests that this controller controls the relevant states quite adequately. For completeness it can be stated that the augmented positional error was centered at 7 units with 1 unit standard deviation, although these quantities are hard to interpret due to the lack of a tangible unit for $||\boldsymbol{x}_a||$. Considering that $||\boldsymbol{x}_a||$ also included the yaw angle, this result can be considered quite robust.

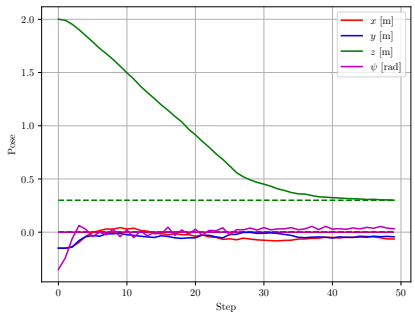



**(a)** The reward the descending agent accumulated over each episode during training, where the reward was given as (4.22). The reason behind the rewards close to 0 after the agent started exhibiting learning behavior is probably due to the fact that the boundary of the agent in the horizontal directions were quite strict, and one aggressive exploratory action may lead the agent outside the valid area.

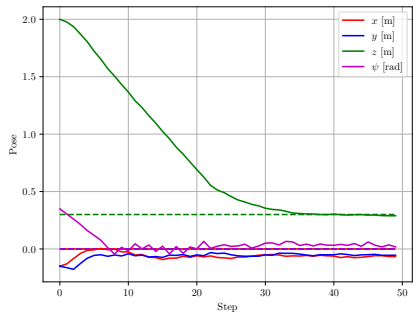



**(b)** The number of steps per episode during training of the descending agent. One can clearly see the correlation between the episodic reward in Figure 5.5a and the number of steps per episode, where a low episodic reward corresponded to an episode with a low number of steps.

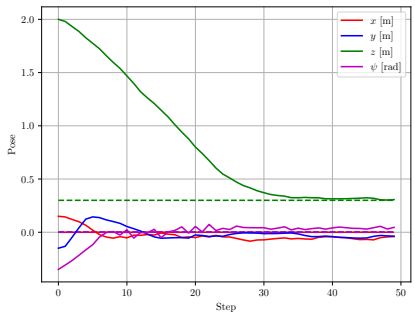
**Figure 5.5:** The reward and number of steps for each episode in the descending agent's training phase.



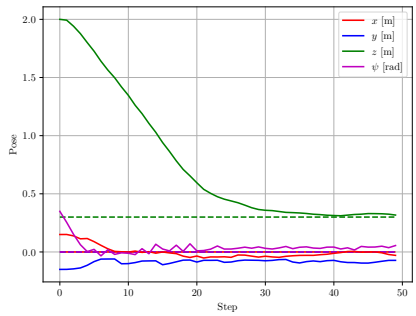

**(a)** Initial position $\boldsymbol{x}_a^w = [-0.15, -0.15, 2.0, -20.0]^\intercal$



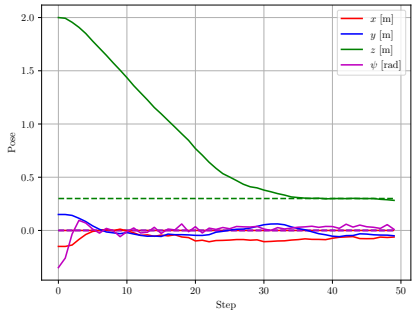

**(b)** Initial position $\boldsymbol{x}_a^w = [-0.15, -0.15, 2.0, 20.0]^\intercal$



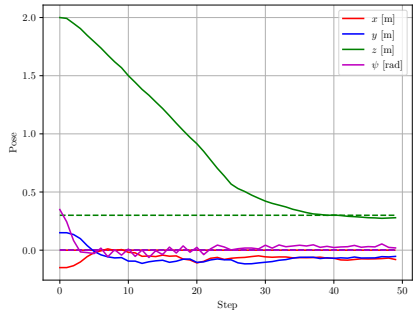

**(c)** Initial position $\boldsymbol{x}_a^w = [0.15, -0.15, 2.0, -20.0]^\intercal$



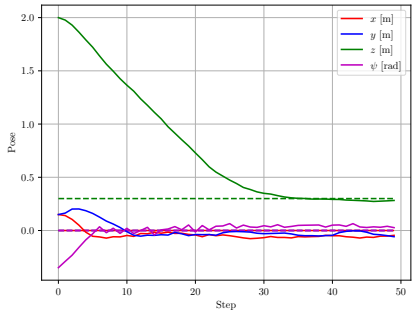

**(d)** Initial position $\boldsymbol{x}_a^w = [0.15, -0.15, 2.0, 20.0]^\intercal$



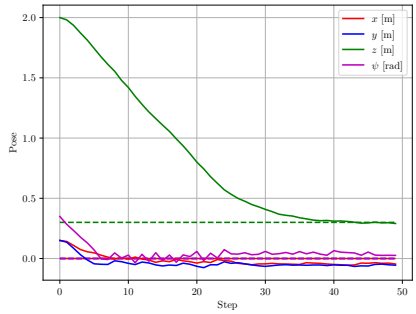

**(e)** Initial position $\boldsymbol{x}_a^w = [-0.15, 0.15, 2.0, -20.0]^\intercal$




**(f)** Initial position $\boldsymbol{x}_a^w = [-0.15, 0.15, 2.0, 20.0]^\intercal$




**(g)** Initial position $\boldsymbol{x}_a^w = [0.15, 0.15, 2.0, -20.0]^\intercal$




**(h)** Initial position $\boldsymbol{x}_a^w = [0.15, 0.15, 2.0, 20.0]^\intercal$

**Figure 5.6:** Eight tests of how the descending agent approaches the setpoint, in dotted lines, with varying initial positions. The horizontal axes illustrate time steps, while the vertical axes constitute the drone's position in the three dimensions together with the yaw angle. $\psi$ is converted to radians in the figures for visual purposes.

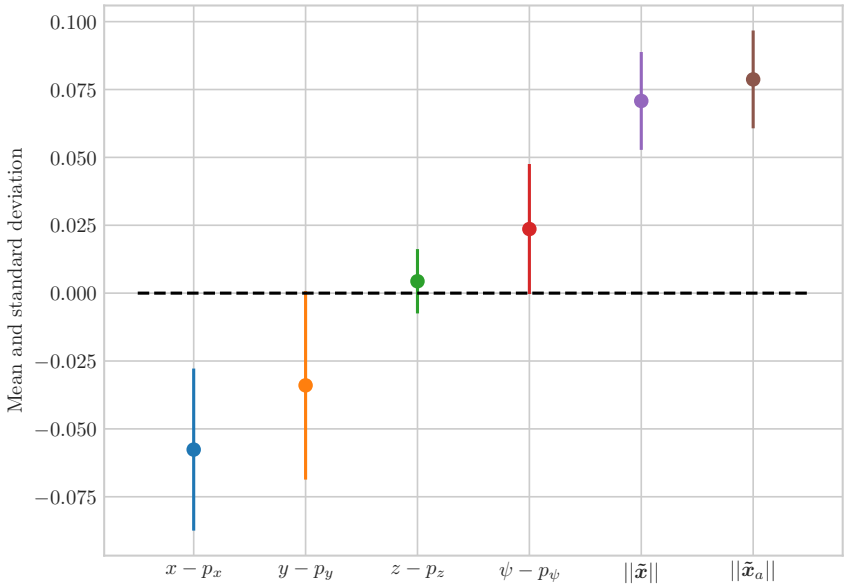



**Figure 5.7:** Means and standard deviations of the drone's error in linear position $\tilde{\boldsymbol{x}}$, yaw angle $\tilde{\psi}$ given in radians and the pseudo-Euclidean distance $||\tilde{\boldsymbol{x}}_a||$, where latter is also computed with the yaw angle being given in radians. Further, the positional error $||\tilde{\boldsymbol{x}}||$ is illustrated for comparative purposes with respect to the hover agent given in Figure 5.4.

## 5.3 Case study results

The results presented in this section were derived from merging the presented framework in this thesis together with the work conducted in [39]. As expressed in Section 4.2.3, the main idea of this case study was to detach the solution from ground truth estimates, which are only available in the simulated environment, and rather adopt an independent perception module able to estimate the drone's pose relative to the landing platform. The chosen solution would utilize a stream of images from the down-facing camera of the drone to calculate the drone's values in $x$, $y$, $z$ and $\psi$ relative to the helipad, and feed these values to the hovering and descending agent. In contrast to the ground truth tests, this section merges all steps of the landing mission, such that the descending agent directly follows the hovering agent, and that the landing command is executed when descending is completed.

Since an external perception module was used to estimate $x$, $y$, $z$ and $\psi$ relative to the landing platform, the drone had to start at the landing platform and initialize and calibrate the dead reckoning module. Further, the drone had to be manually flown to an initial position where the platform was visible to the agent, in order for the dead reckoning module to be able to estimate the drone's position from the point of initialization. As such, there was no trivial way of initializing the agent as accurately to a specific point as in previous tests. In previous sections, the hovering and descending agent were analyzed separately for analyzing their performances irrespective of each other to ensure that each entity worked as intended. This case study serves as an augmentation and the subsequent analysis describes the full landing mission, from initialization to landing. As such, the tests conducted in this section constitute results from incorporating and merging the hovering and descending agent into one single system through the planning scheme illustrated in Figure 4.1.

With respect to this planner used, the system considered that the hovering step was completed when the hovering agent fulfilled

$$||\tilde{\boldsymbol{x}}|| < 0.03\,m \tag{5.8}$$

for 3 consecutive time steps, or 1 second with the 3 Hz controller. Through testing it was found that this value was long enough for the agent to stabilize at $\boldsymbol{p}_h$ while still brief enough to avoid redundant postponement of the descending mission. Although 1 second may seem insufficient for assuring that the hovering state has been completed and the drone may still inhabit enough inertia to drift away, none of the tests conducted suggested that the hovering agent was not stable after this duration. Additionally, the magnitude of the actions imposed by the hovering agent when close to the setpoint was quite small, yielding low values for velocity and acceleration. Furthermore, the descending agent would be able to handle any discrepancy that would result from the unlikely event of the hovering agent not being fully stable when the system propagates from hover to descend. As such, 1 second was considered a reasonable amount of time before it evolved its state from hover to descend. Similarly, the system would have to observe the descending agent accomplishing

$$||\tilde{\boldsymbol{x}}_a|| < 0.05 \tag{5.9}$$

for 12 time steps, or 4 seconds, in order to evolve from descend to land. The hovering agent was stricter with its boundary due to the fact that the descending agent also had to

take the error in yaw into account. Simultaneously, the descending agent had to hold its position close to the setpoint for a longer period of time, since the transition from descend to land was considered to be a move inhabiting higher risk with respect to the landing mission. A slight deviation when transitioning from hover to descend could be rectified by the descending agent, while this could not be done when evolving from descend to land, considering that the landing command did not inhabit any positional control.

Due to time constraints only five tests were conducted to test the hovering and descending agents with the perception module. The hovering and descending setpoints were unaltered from previous analysis since they were still deemed reasonable values, even for this case study. The respective tests are depicted in Figure 5.8, Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12. Also consistent with previous analysis, the figures show $\psi$ in radians for readability and analysis, although the initial coordinates are given in degrees. Note that the figures do not illustrate that the landing step is conducted, due to the simulator ending the test runs immediately after the landing command has been received. However, videos capturing each respective test run are attached in the captions to illustrate the performance of the solution.

Generally, comparing the results in this case study to the results obtained using ground truth estimates, the agent struggled with completing the hovering step notably due to the unstable behavior in the horizontal plane. Both $x$ and $y$ in all five tests seemed to deviate notably relative to their respective setpoints, hindering the agent from completing (5.8). On the other hand, the hovering agent generally seemed to do a quite good job with respect to controlling $z$ to its desired value. As the tests show, despite the extra efforts to complete the hovering step the hovering agent eventually settled its task and the system evolved to descend.

From the tests conducted, it seemed that the descending agent excelled in its task. $z$ decreases monotonically towards the desired value while the responses in $x$ and $y$ are decreased significantly compared to the hovering agent. The latter point is illustrated especially well in Figure 5.9, where the agent switched from hover to descend. These results can be due to the fact that this agent is overall closer to the platform, which may be an advantage for the perception module in terms of stability and accuracy in the estimation calculations.

The results properly exhibit the potential and advantage that DRL solutions offer. Figure 5.11 and Figure 5.12 have initial positions in $z$ outside the boundary value for the hovering agent. The latter even has its initial $y$ value outside the training boundary. Yet, the agent manages to generalize well enough to reach its setpoint in both cases.

Although all tests resulted in a successful landing, there are some challenges that arise from these findings. The aggressive responses in the horizontal directions are unfavorable and might even drive the drone to a position where it is not able to perceive the landing platform, thus leaving the agent with no grounds for estimation, other than dead-reckoning, which is prone to drift. Accordingly, such behavior may lead to failure in the landing mission.

A big disadvantage with this framework is that the quadrotor has to see the helipad from

the beginning of each run in order to have a robust estimate of its position, which limits the system's range of applicability.

Although the findings in this case study facilitate successful integration between the drone control module and the perception module, they still give rise for questions that are not easily researched nor answered. For instance, it is not a trivial task to research why the horizontal components are behaving in such an aggressive manner compared to when using only the ground truth estimates, where the responses come across as more smooth and controlled. As such, it is difficult to say why the altitude of the drone seems to be well controlled while the horizontal components struggle. Furthermore, it is difficult to argue what the reason for the large disparity in the duration of runs, where Figure 5.12 spends up to 100 steps before initiating landing while Figure 5.11 arrives at the same stage after a mere 61 steps. Overshooting behavior is also noticed in all tests, which was not present using the ground truth results. It is worth mentioning that further investigation to these details was nontrivial due to the constrained collaboration.

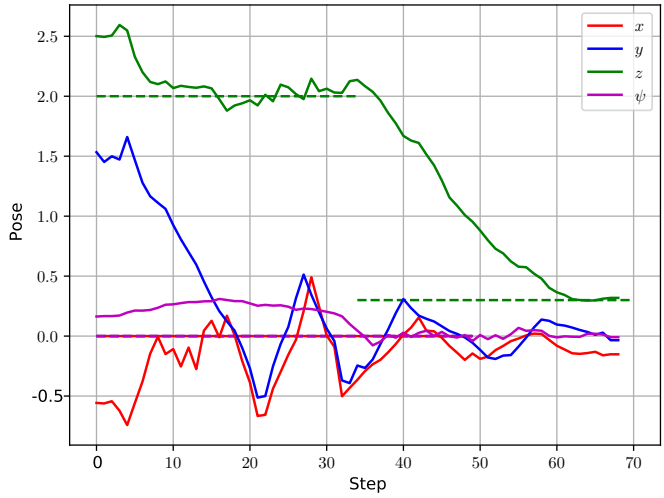



**Figure 5.8:** Initial position at $\boldsymbol{x}_a = [-0.55, 1.53, 2.50, 9.34]^\intercal$ relative to the helipad. The system switches from hover to descend at step 34, and lands after 70 steps, meaning it completed hovering after 11 seconds and initiated landing after 24 seconds. Footage of the test.

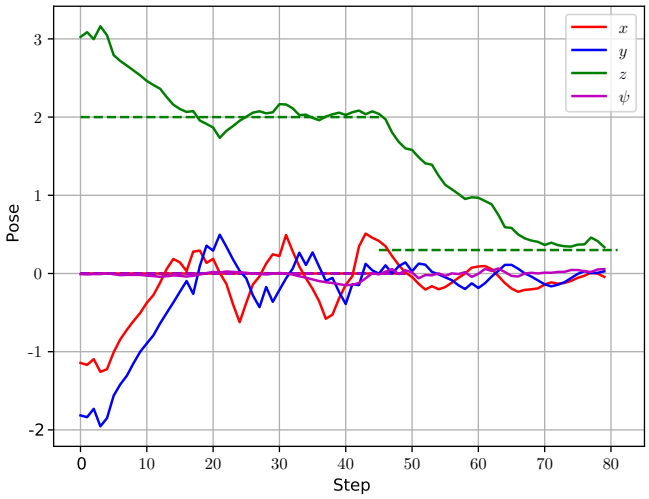



**Figure 5.9:** Initial position at approximately $\boldsymbol{x}_a = [-1.14, -1.81, 3.02, -0.47]^\intercal$ relative to the helipad. The system switches from hover to descend at step 45, and lands after 81 steps, meaning it completed hovering after 15 seconds and initiated landing after 27 seconds. Footage of the test.

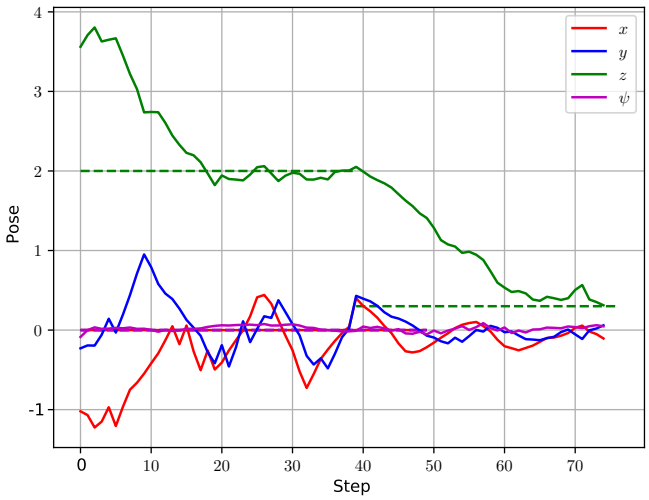



**Figure 5.10:** Initial position at approximately $\boldsymbol{x}_a = [-1.02, -0.23, 3.56, -4.96]^\intercal$ relative to the helipad. The system switches from hover to descend at step 39, and lands after 76 steps, meaning it completed hovering after 13 seconds and initiated landing after 26 seconds. Footage of the test.

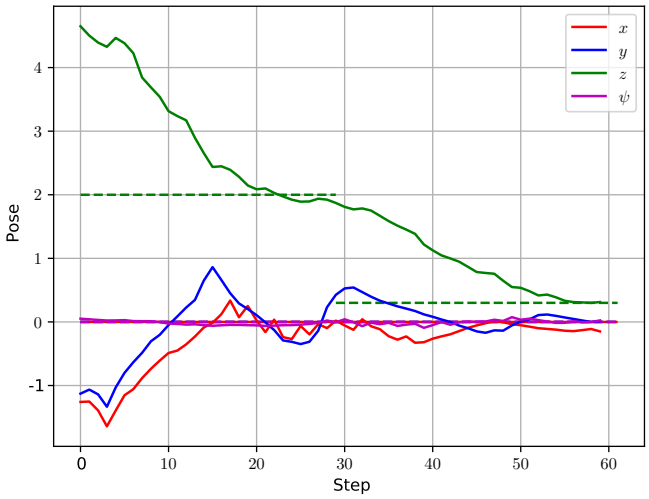



**Figure 5.11:** Initial position at approximately $\boldsymbol{x}_a = [-1.26, -1.12, 4.65, 2.93]^\intercal$ relative to the helipad. The system switches from hover to descend at step 29, and lands after 61 steps, meaning it completed hovering after 10 seconds and initiated landing after 21 seconds. Footage of the test.

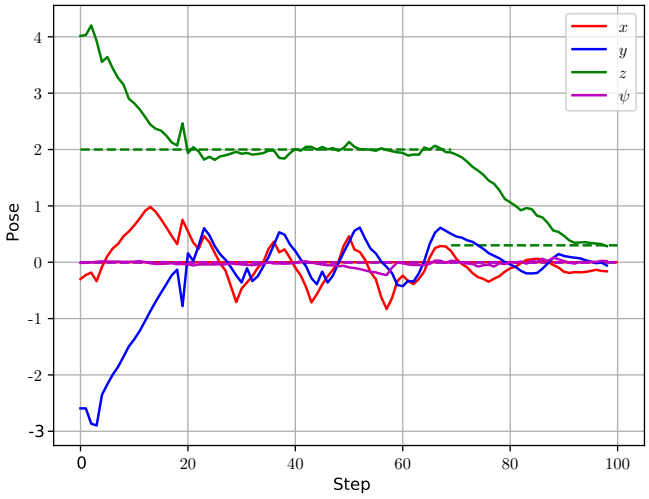



**Figure 5.12:** Initial position at approximately $\boldsymbol{x}_a = [-0.29, -2.59, 4.01, -0.71]^\intercal$ relative to the helipad. The system switches from hover to descend at step 69, and lands after 100 steps, meaning it completed hovering after 23 seconds and initiated landing after 34 seconds. Footage of the test.

Chapter 6

# Future work

From the findings using both the ground truth estimate and the perception module, it is evident that the solution proposed is able to complete the landing mission for a drone well. However, there are some elements in the work done that could be augmented or even changed completely, both for researching how the behavior changed and most certainly enhance the performance and applicability of the solution.

To begin with, it would be advantageous not to manually normalize the states and consequently bound them to specified values during training. Despite the fact that it is desirable to limit the environment to a specific area in order to limit training time, it can be argued that the chosen design did not favor flexibility in the derived solutions. For instance, if the setpoint was changed the boundaries had to be altered correspondingly for the two entities to still have the same relative relationship. Further, only the values at the input layer were normalized, while none of the inputs to the hidden layers were. As a means of mitigating these elements, batch normalization could be used to automatically normalize the input to the networks, and also the input to the hidden layers. Continuously calculating a mean and variance for input to a node could boost training time and convergence while simultaneously yielding a more flexible system where the normalization would no longer depend on the setpoint. It should be mentioned that strenuous attempts were conducted to implement batch normalization layers to the networks using the Keras framework. However, after extensive research it was concluded that this layer had not been implemented by the creators as it was intended to, where the running mean and variances were not calculated properly, resulting in the network outputs being erroneous. Although it would be possible to change framework from Keras to e.g. Tensorflow's own and substitute `tf.keras` framework for the stand-alone Keras framework, this was not completed due to time constraints.

For research purposes, it could be of significant use to attempt various state vectors to observe how each individual state would affect an agent in reaching its setpoint. An idea during design and development was add the roll and pitch angles of the drone to the state vector, and rather than penalizing actuation, the reward function would penalize these

quantities, such that the agent would not tilt in either direction and thus stay horizontal and move at low speeds. Another potential state augmentation would be to add the action executed in the preceding time step and penalize big alterations in action commands. This would mitigate aggressive action control and thus oscillating behavior. Simultaneously, it would promote smooth convergence towards the setpoint rather than a bang-bang approach.

Further, in order to properly benchmark the results obtained, it would be advantageous to rigorously compare the solution obtained with more traditional methods, such as a PID controller or MPC. Also adding disturbances, such as wind gusts or other aerodynamic forces could help painting the picture with respect to how robust the solution is. This way, it would be trivial to compare the DRL agents derived with the methods that seem to be predominantly used in most real-life applications to date. Also conducting a multitude of tests in the case study would be of interest to see how successful and robust the system is with respect to varied initializations. It would also be favorable to analyze the actuation and see in more detail how the penalization factor in the reward function affected the behavior of the agents.

It would also be of interest to augment the system to run on the physical drone version of the AR.Drone 2.0 in a laboratory, and observe the differences between a simulated and a controlled, but real, environment. Ultimately, it would also be of great interest to augment the functionality in order to deploy the system in a real-life setting where external forces, realistic drifting behavior and signal delays are present.

Since the hovering agent and descending agent are quite similar in design and overall objective, it would also be of interest to investigate how transfer learning could be applied in this situation. Although training was quite rapid and seldom required more than one hour, this could shorten training time remarkably and also promote better results.

Also the overall architecture of the solution could be optimized if a dedicated planning framework, such as T-REX [78], Graphplan [79] or STRIPS [80] would have been used. This could, as a result, open the possibility for developing one single position controller that would have the ability to converge towards any setpoint the user inputs, effectively eliminating the need of dedicated hovering and descending agents. It could also yield a more robust system able to do obstacle avoidance, where the planner would have the potential of overcoming such challenges along the way.

Chapter 7

# Conclusion

This thesis presented an approach for applying deep reinforcement learning to drone control for autonomous landing on a helipad with a predefined location. The motivation behind adopting a more state-of-the-art solution rather than the more traditional PID and MPC approaches was mainly to develop a model-free control approach for optimal control. By letting the DRL agent explore the surroundings, learning what actions are good and bad relative to the reward fed back and optimizing its behavior based on this value, the thesis proposed two agents for hovering and descending, respectively, and merging their functionality to a complete system using a rudimentary planner. Reward functions specific to their final objective were developed and tailored for efficient training while simultaneously obtaining as satisfying behavior for the agents as possible. The two agents were subject to extensive analysis regarding their training progress as well as their convergence, robustness and accuracy post-training.

Comparing the two learning phases of the agents it was safe to say that the hovering agent had the more efficient training, since it started learning notably earlier than the descending agent. This allowed the hovering agent to explore the environment close to the goal more extensively. This could contribute to fine-tuned behavior close to the hovering point. Since the descending agent learned slower, it would not have the same opportunities to experience the space close to its setpoint in the same manner. Additionally, the latter agent was also hindered by the strict boundaries set for the environment, thus robbing the agent for a substantial number of episodes it could have spent exploring the environment.

The descending agent could benefit from a more forgiving environment that would allow some deviation in the horizontal direction. Although the agent seemed to learn its objective quite well, it can be argued that this change would boost efficiency in training significantly, as it would not experience environment exiting as frequently.

With very high accuracy for both hovering and descending agent using the ground truth estimates in addition to seamlessly integrate the external perception, it can be argued that

the work in this thesis contribute in exhibiting the great potential of DRL solutions for robotic control tasks and may hopefully standardize DRL as a control approach in such frameworks. For all 100 runs for the hovering agent and 120 runs for the descending agent using ground truth estimates, the agents converged towards their setpoints without fail. Also, for 5 the case study tests, all resulted in successful landing.

Although the findings in this thesis suggest that DRL approaches grant well-performing, model-free approaches for controlling highly nonlinear systems, there are drawbacks that must be taken into consideration when adopting such solutions. An important point to make is that since DRL solutions base themselves on estimating the optimal policy and value function through artificial neural networks, these methods merely compute *approximations* of the optimal control law relative to a given reward function. As such, there are no guarantees that the agent will converge after training is completed. Increasing the complexity in these networks could yield more accurate approximations to the true optimal functions, but would demand additional computational power to realize. Emerging from this is one of the main disadvantages with deep learning, namely unstable training. The former poses many challenges to the design of the environment and the agent, where a small change in the reward function may be the difference between convergence to a sufficient policy or not. This is a very intricate subject to research and, at the time of writing, the author has not been able to find material suggesting any clarification to the matter. Furthermore, DRL methods famously offer no stability guarantees, mainly due to the fact that analyzing the input-output relationship in a deep neural network with numerous hidden layers and nodes is a highly nontrivial task. It is important to consider that many of the robotic systems to employ the DRL methodologies are not meant to serve as recommendation systems or to play Atari games, but rather physical systems with a possibility of being damaged or even being dangerous to people. Since such solutions grant no guarantees that an agent will behave as intended for all initial states, they must be deployed with a high regard for safety.

With that being said, DRL in general poses a very flexible framework for complex problems that would otherwise struggle with modeling. The areas of use for DRL methods in the last years have grown immensely, and research continues to improve the frameworks and performances of such methods. With time the disadvantages and drawbacks presented will most likely be mitigated, paving way for more robust, safe and well-performing solutions.

# Bibliography


[1] Daniel Tavakoli. *Planning and Control for Autonomous Drone Landing*. Project report in TTK4550. Department of Engineering Cybernetics, NTNU – Norwegian University of Science and Technology, 2019.

[2] *Udacity Nanodegree Program*. https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893. Accessed: 2020-01-22.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[4] *ROS package:* `ardrone_autonomy`. http://wiki.ros.org/ardrone_autonomy. Accessed: 2019-09-23.

[5] *ROS package:* `tum_simulator`. http://wiki.ros.org/tum_simulator. Accessed: 2019-09-24.

[6] *ROS package:* `hector_gazebo`. http://wiki.ros.org/hector_gazebo. Accessed: 2019-09-23.

[7] Yogianandh Naidoo, Riaan Stopforth, and Glen Bright. "Development of an UAV for search rescue applications". In: Sept. 2011, pp. 1–6. ISBN: 978-1-61284-992-8. DOI: 10.1109/AFRCON.2011.6072032.

[8] Sophie Jordan, Julian Moore, Sierra Hovet, John Box, Kevin Kirsche, Jason Perry, Dexter Lewis, and Zion Tse. "State of the Art Technologies for UAV Inspections". In: *IET Radar, Sonar Navigation* 12 (Oct. 2017). DOI: 10.1049/iet-rsn.2017.0251.

[9] Eduard Semsch, Michal Jakob, Dusan Pavlícek, and Michal Pechoucek. "Autonomous UAV Surveillance in Complex Urban Environments". In: vol. 2. Jan. 2009, pp. 82–85. DOI: 10.1109/WI-IAT.2009.132.

[10] Rey Koslowski and Marcus Schulzke. "Drones Along Borders: Border Security UAVs in the United States and the European Union". In: *International Studies Perspectives* 19 (July 2017). DOI: 10.1093/isp/eky002.

[11] Francesco Nex and Fabio Remondino. “UAV for 3D mapping applications: A review”. In: *Applied Geomatics* 6 (Mar. 2014). DOI: 10.1007/s12518-013-0120-x.

[12] Emmanouil Barmpounakis, Eleni Vlahogianni, and John Golias. “Unmanned Aerial Aircraft Systems for transportation engineering: Current practice and future challenges”. In: 5 (Feb. 2017). DOI: 10.1016/j.ijtst.2017.02.001.

[13] Raimundo Felismina, Miguel Silva, Artur Mateus, and Cândida Malça. “Development of a Universal Seeder System to Be Applied in Drones”. In: *Journal of Advanced Agricultural Technologies* 4 (Jan. 2017), pp. 123–127. DOI: 10.18178/joaat.4.2.123-127.

[14] Yi Feng, Cong Zhang, Stanley Baek, Samir Rawashdeh, and Alireza Mohammadi. “Autonomous Landing of a UAV on a Moving Platform Using Model Predictive Control”. In: *Drones* 2 (Oct. 2018), p. 34. DOI: 10.3390/drones2040034.

[15] Kostas Alexis, George Nikolakopoulos, and Anthony Tzes. “Switching model predictive attitude control for a quadrotor helicopter subject to atmospheric disturbances”. In: *Control Engineering Practice* 19 (Oct. 2011), pp. 1195–1207. DOI: 10.1016/j.conengprac.2011.06.010.

[16] José Alfredo Macés Hernández, François Defaÿ, and Corentin Chauffaut. “Autonomous landing of an UAV on a moving platform using model predictive control”. In: Dec. 2017, pp. 2298–2303. DOI: 10.1109/ASCC.2017.8287533.

[17] Naila Qayyum, Aamer Bhatti, and Muwahida Liaquat. “Landing control of unmanned aerial vehicle using continuous model predictive control”. In: May 2017, pp. 1804–1808. DOI: 10.1109/CCDC.2017.7978809.

[18] Kiam Heong Ang, G. Chong, and Yun Li. “PID control system analysis, design, and technology”. In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576.

[19] Robert Paz. “The Design of the PID Controller”. In: (Jan. 2001).

[20] Erdem Yilmaz and Junling Hu. “CFD Study of Quadcopter Aerodynamics at Static Thrust Conditions”. In: Apr. 2018.

[21] Dhwanil Shukla and Narayanan Komerath. “Multirotor Drone Aerodynamic Interaction Investigation”. In: *Drones* 2 (Dec. 2018), p. 43. DOI: 10.3390/drones2040043.

[22] D.E. Seborg. *Process Dynamics and Control*. John Wiley & Sons, Incorporated, 2012. ISBN: 9781119929857. URL: https://books.google.no/books?id=r70prgEACAAJ.

[23] Lemma Dendena Tufa and Zhi Kai Chong. “Effect of Model Plant Mismatch on MPC Performance and Mismatch Threshold Determination”. In: *Procedia Engineering* 148 (Dec. 2016), pp. 1008–1014. DOI: 10.1016/j.proeng.2016.06.518.

[24] Mark Lee. *Evaluative Feedback*. URL: http://incompleteideas.net/book/first/ebook/node14.html.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[26] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. “Mastering the game of Go with deep neural networks and tree search”. In: (2016). URL: https://www.nature.com/articles/nature16961.pdf.

[27] Riccardo Polvara, Sanjay Sharma, Jian Wan, Andrew Manning, and Robert Sutton. “Towards autonomous landing on a moving vessel through fiducial markers”. In: (2017). URL: https://ieeexplore.ieee.org/abstract/document/8098671.

[28] Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. “Control of a Quadrotor with Reinforcement Learning”. In: *CoRR* abs/1707.05110 (2017). arXiv: 1707.05110. URL: http://arxiv.org/abs/1707.05110.

[29] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. “Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search”. In: *CoRR* abs/1509.06791 (2015). arXiv: 1509.06791. URL: http://arxiv.org/abs/1509.06791.

[30] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *CoRR* abs/1509.02971 (2015).

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[32] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning”. In: *ICML*. 2016.

[33] Kyowoon Lee, Sol-A Kim, Jaesik Choi, and Seong-Whan Lee. “Deep Reinforcement Learning in Continuous Action Spaces: a Case Study in the Game of Simulated Curling”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 2937–2946. URL: http://proceedings.mlr.press/v80/lee18b.html.

[34] Lucian Buşoniu, Tim de Bruin, Domagoj Tolić, Jens Kober, and Ivana Palunko. *Reinforcement learning for control: Performance, stability, and deep approximators*. 2018. DOI: https://doi.org/10.1016/j.arcontrol.2018.09.005. URL: http://www.sciencedirect.com/science/article/pii/S1367578818301184.

[35] Andreas Bell Martinsen. “End-to-end training for path following and control of marine vehicles”. In: 2018.

[36] Yuanda Wang, Jia Sun, Haibo He, and Changyin Sun. “Deterministic Policy Gradient With Integral Compensator for Robust Quadrotor Control”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* PP (Jan. 2019), pp. 1–13. DOI: 10.1109/TSMC.2018.2884725.

[37] Lasse Hansen Henriksen. “Hovering Control of a Quadrotor Using Monocular Images and Deep Reinforcement Learning”. MA thesis. NTNU, 2019.

[38] Alejandro Rodríguez Ramos, Carlos Sampedro Pérez, Hriday Bavle, Paloma de la Puente, and Pascual Campoy. “A Deep Reinforcement Learning Strategy for UAV Autonomous Landing on a Moving Platform”. In: *Journal of Intelligent and Robotic Systems* (July 2018). DOI: 10.1007/s10846-018-0891-8.

[39] Thomas Sundvoll. “A Camera-based Perception System for Autonomous Quadcopter Landing on a Marine Vessel”. MA thesis. NTNU, 2020.

[40] Haomiao Huang, Gabriel Hoffmann, Steven Waslander, and C.J. Tomlin. “Aerodynamics and control of autonomous quadrotor helicopters in aggressive maneuvering”. In: June 2009, pp. 3277–3282. DOI: 10.1109/ROBOT.2009.5152561.

[41] Yingjun Pei and Xinwen Hou. *Learning Representations in Reinforcement Learning: an Information Bottleneck Approach*. 2020. URL: https://openreview.net/forum?id=Syl-xpNtwS.

[42] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* (1958), pp. 65–386.

[43] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning internal representations by error propagation”. In: 1986.

[44] Kurt Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4 (1991), pp. 251–257.

[45] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[46] D. E. Rumelhart and J. L. McClelland. “Learning Internal Representations by Error Propagation”. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. MITP, 1987, pp. 318–362. URL: https://ieeexplore.ieee.org/document/6302929.

[47] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (Dec. 2014).

[48] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting.” In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958. URL: http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf.

[49] A.Y. Ng. “Feature selection, l1 vs. l2 regularization, and rotational invariance”. In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 2004, p. 78.

[50] Lutz Prechelt. “Early Stopping - But When?” In: (Mar. 2000). DOI: 10.1007/3-540-49430-8_3.

[51] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. “Parametric Noise Injection: Trainable Randomness to Improve Deep Neural Network Robustness against Adversarial Attack”. In: *CoRR* abs/1811.09310 (2018). arXiv: 1811.09310. URL: http://arxiv.org/abs/1811.09310.

[52] A. Mikołajczyk and M. Grochowski. “Data augmentation for improving deep learning in image classification problem”. In: *2018 International Interdisciplinary PhD Workshop (IIPhDW)*. 2018, pp. 117–122.

[53] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[54] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[55] Richard Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. URL: http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%5C%20programming%5C%20richard%5C%20e%5C%20bellman&f=false.

[56] H. van Hasselt and M. A. Wiering. “Convergence of Model-Based Temporal Difference Learning for Control”. In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. 2007, pp. 60–67. DOI: 10.1109/ADPRL.2007.368170.

[57] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML’14. Beijing, China: JMLR.org, 2014, pp. I-387–I-395. URL: http://dl.acm.org/citation.cfm?id=3044805.3044850.

[58] B. T. Polyak and A. B. Juditsky. “Acceleration of Stochastic Approximation by Averaging”. In: *SIAM J. Control Optim.* 30.4 (July 1992), pp. 838–855. ISSN: 0363-0129. DOI: 10.1137/0330046. URL: http://dx.doi.org/10.1137/0330046.

[59] A. G. Malliaris. “Wiener Process”. In: *Econometrics*. Ed. by John Eatwell, Murray Milgate, and Peter Newman. London: Palgrave Macmillan UK, 1990, pp. 276–278. ISBN: 978-1-349-20570-7. DOI: 10.1007/978-1-349-20570-7_38. URL: https://doi.org/10.1007/978-1-349-20570-7_38.

[60] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *CoRR* abs/1802.09477 (2018). arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477.

[61] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[62] Jinhyun Kim, Min-Sung Kang, and Sangdeok Park. “Accurate Modeling and Robust Hovering Control for a Quad–rotor VTOL Aircraft”. In: *Journal of Intelligent and Robotic Systems* 57 (Jan. 2010), pp. 9–26. DOI: 10.1007/978-90-481-8764-5_2.

[63] Russell Smith. *Open Dynamics Engine*. http://www.ode.org/. 2008. URL: http://www.ode.org/.

[64] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[65] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[66] François Chollet et al. *Keras*. https://keras.io. 2015.

[67] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. “Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo”. In: vol. 7628. Nov. 2012, pp. 400–411. DOI: 10.1007/978-3-642-34327-8_36.

[68] Hongrong Huang. *Gazebo Simulator for the Parrot AR.Drone quadrocopter*. YouTube. 2012. URL: https://www.youtube.com/watch?v=s_SBLeXRrhE&feature.

[69] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. “End-to-End Training of Deep Visuomotor Policies”. In: *CoRR* abs/1504.00702 (2015). arXiv: 1504.00702. URL: http://arxiv.org/abs/1504.00702.

[70] Sergey Levine. “Exploring Deep and Recurrent Architectures for Optimal Control”. In: *CoRR* abs/1311.1761 (2013). arXiv: 1311.1761. URL: http://arxiv.org/abs/1311.1761.

[71] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2018. URL: http://arxiv.org/abs/1803.08375.

[72] M. Hodnett and J.F. Wiley. *R Deep Learning Essentials: A Step-By-step Guide to Building Deep Learning Models Using TensorFlow, Keras, and MXNet, 2nd Edition*. Packt Publishing, 2018. ISBN: 9781788992893. URL: https://books.google.no/books?id=M3cguwEACAAJ.

[73] J. Canny. “A Computational Approach to Edge Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698.

[74] Chris Harris and Mike Stephens. “A combined corner and edge detector”. In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151.

[75] C.-H Teh and Roland T Chin. “On Image Analysis by the Method of Moments”. In: vol. 10. July 1988, pp. 556–561. ISBN: 0-8186-0862-5. DOI: 10.1109/CVPR.1988.196290.

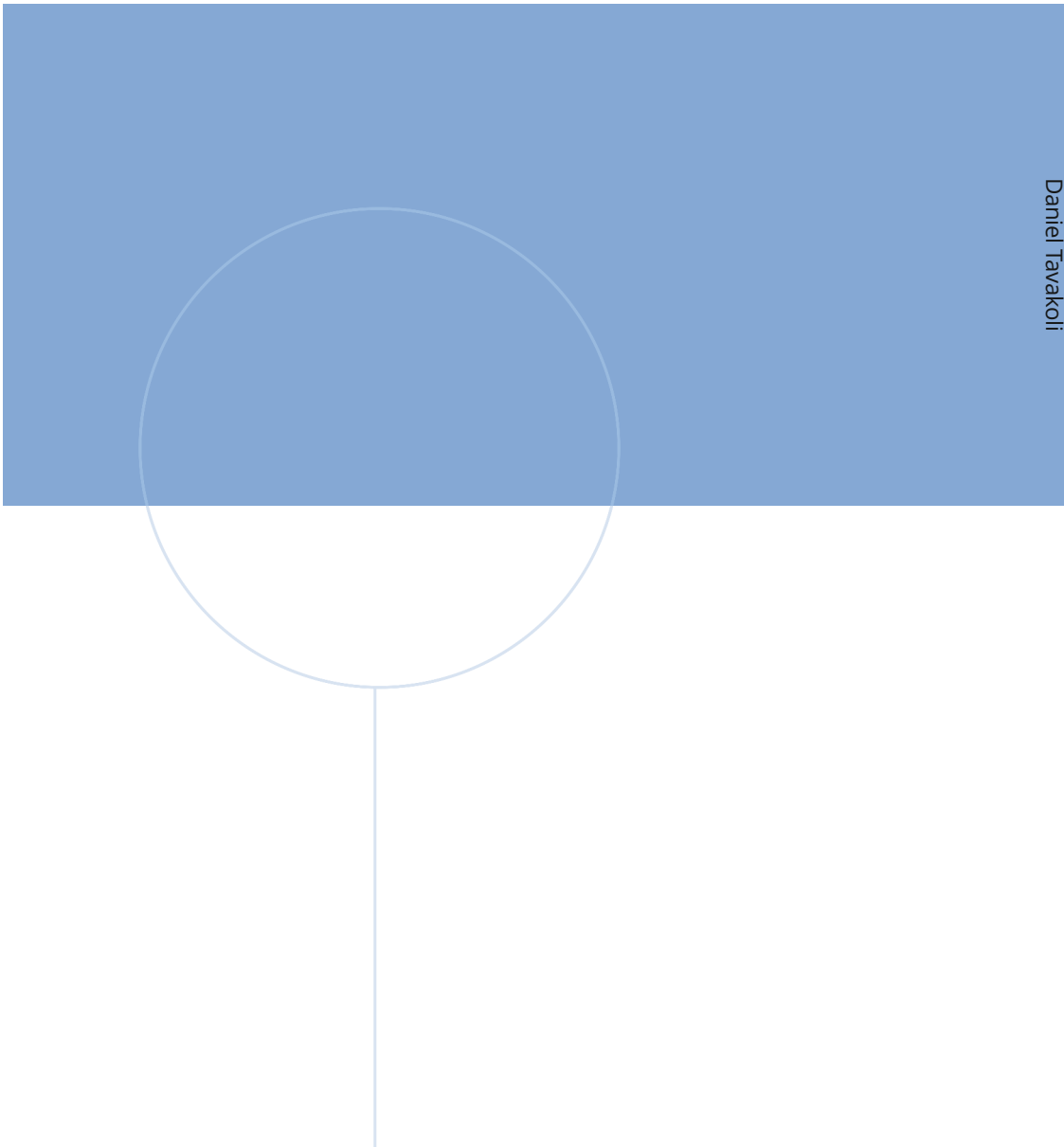
[76] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W. Mahoney. “Large batch size training of neural networks with adversarial training and second-order information”. In: *CoRR* abs/1810.01021 (2018). URL: http://arxiv.org/abs/1810.01021.

[77] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[78] Daniel S. Brown, Wonjoon Goo, Prabhat Nagarajan, and Scott Niekum. “Extrapolating Beyond Suboptimal Demonstrations via Inverse Reinforcement Learning from Observations”. In: *CoRR* abs/1904.06387 (2019). arXiv: 1904.06387. URL: http://arxiv.org/abs/1904.06387.

[79] Avrim L. Blum and Merrick L. Furst. “Fast planning through planning graph analysis”. In: *Artificial Intelligence* 90.1 (1997), pp. 281–300. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(96)00047-1. URL: http://www.sciencedirect.com/science/article/pii/S0004370296000471.

[80] Richard Fikes and Nils J. Nilsson. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artif. Intell.* 2 (1971), pp. 189–208.



Daniel Tavakoli

NTNU
Norwegian University of
Science and Technology