Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Siqueira Eduardo

Deep Reinforcement Learning for Quadrotor Trajectory Control in Virtual Environments

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Engenharia Elétrica, do Departamento de Engenharia Elétrica da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Engenharia Elétrica.

Advisor: Prof. Wouter Caarls

Rio de Janeiro
March 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Guilherme Siqueira Eduardo

Deep Reinforcement Learning for Quadrotor Trajectory Control in Virtual Environments

Dissertation presented to the Programa de Pós–graduação em Engenharia Elétrica da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Engenharia Elétrica. Approved by the Examination Committee:

Prof. Wouter Caarls
Advisor
Departamento de Engenharia Elétrica – PUC-Rio

Prof. Antonio Candea Leite
Norwegian University of Life Sciences

Profª. Karla Figueiredo Leite
Universidade do Estado do Rio de Janeiro

Prof. Eduardo Costa da Silva
Departamento de Engenharia Elétrica – PUC-Rio

Rio de Janeiro, March the 26th, 2021

Guilherme Siqueira Eduardo

Majored in Control and Automation Engineering at the Pontifical Catholic University of Rio de Janeiro in 2018. Together with the AeroRio team, participated in international competitions and conferences on autonomous drones. Since then, he specializes in the field of robotics and artificial intelligence.

Bibliographic data

Siqueira Eduardo, Guilherme

Deep Reinforcement Learning for Quadrotor Trajectory Control in Virtual Environments / Guilherme Siqueira Eduardo; advisor: Wouter Caarls. – 2021.

118 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica, 2021.

Inclui bibliografia

1. Engenharia Elétrica – Teses. 2. Controle de quadrotor. 3. Veículo aéreo não-tripulado (VANT). 4. Aprendizado por reforço profundo. 5. Soft Actor-Critic (SAC). 6. Navegação visual.. I. Caarls, Wouter. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Elétrica. III. Título.

CDD: 621.3

To my parents, for their love and support.

Acknowledgments

I would like to thank my advisor Prof. Wouter Caarls for backing the development of this project and for his invaluable and impeccable advices, lectures and support.

I would like to thank Prof. Antonio Leite, Prof[a]. Karla Figueiredo and Prof. Eduardo Costa, for the learning opportunities in the various projects I had a chance to be a part of.

I would like also to express my gratitude for the professors of the DEE Master's programme.

I would like to thank as well my friends and colleagues made in PUC-Rio, especially those of AeroRio, for the happy moments and the travel experiences together.

I thank the financial support provided by PUC-Rio and CAPES.

Finally, and most importantly, I want to thank my family for all the support given through my life, my education and my endeavours.

"This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001"

Abstract

Siqueira Eduardo, Guilherme; Caarls, Wouter (Advisor). **Deep Reinforcement Learning for Quadrotor Trajectory Control in Virtual Environments**. Rio de Janeiro, 2021. 118p. Dissertação de Mestrado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

With recent advances in computational power, the use of novel, complex control models has become viable for controlling quadrotors. One such method is Deep Reinforcement Learning (DRL), which can devise a control policy that better addresses non-linearities in the quadrotor model than traditional control methods. An important non-linearity present in payload carrying air vehicles are the inherent time-varying properties, such as size and mass, caused by the addition and removal of cargo. The general, domain-agnostic approach of the DRL controller also allows it to handle visual navigation, in which position estimation data is unreliable. In this work, we employ a Soft Actor-Critic algorithm to design controllers for a quadrotor to carry out tasks reproducing the mentioned challenges in a virtual environment. First, we develop two waypoint guidance controllers: a low-level controller that acts directly on motor commands and a high-level controller that interacts in cascade with a velocity PID controller. The controllers are then evaluated on the proposed payload pickup and drop task, thereby introducing a time-varying variable. The controllers conceived are able to outperform a traditional positional PID controller with optimized gains in the proposed course, while remaining agnostic to a set of simulation parameters. Finally, we employ the same DRL algorithm to develop a controller that can leverage visual data to complete a racing course in simulation. With this controller, the quadrotor is able to localize gates using an RGB-D camera and devise a trajectory that drives it to traverse as many gates in the racing course as possible.

Keywords

Quadrotor control; Unmanned Aerial Vehicle (UAV); Deep Reinforcement Learning; Soft Actor-Critic (SAC); Visual navigation.

Resumo

Siqueira Eduardo, Guilherme; Caarls, Wouter. **Aprendizado por Reforço Profundo para Controle de Trajetória de um Quadrotor em Ambientes Virtuais**. Rio de Janeiro, 2021. 118p. Dissertação de Mestrado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

Com recentes avanços em poder computacional, o uso de novos modelos de controle complexos se tornou viável para realizar o controle de quadrotores. Um destes métodos é o aprendizado por reforço profundo (do inglês, Deep Reinforcement Learning, DRL), que pode produzir uma política de controle que atende melhor as não-linearidades presentes no modelo do quadrotor que um método de controle tradicional. Umas das não-linearidades importantes presentes em veículos aéreos transportadores de carga são as propriedades variantes no tempo, como tamanho e massa, causadas pela adição e remoção de carga. A abordagem geral e domínio-agnóstica de um controlador por DRL também o permite lidar com navegação visual, na qual a estimação de dados de posição é incerta. Neste trabalho, aplicamos um algorítmo de *Soft Actor-Critic* com o objeivo de projetar controladores para um quadrotor a fim de realizar tarefas que reproduzem os desafios citados em um ambiente virtual. Primeiramente, desenvolvemos dois controladores de condução por *waypoint*: um controlador de baixo nível que atua diretamente em comandos para o motor e um controlador de alto nível que interage em cascata com um controlador de velocidade PID. Os controladores são então avaliados quanto à tarefa proposta de coleta e alijamento de carga, que, dessa forma, introduz uma variável variante no tempo. Os controladores concebidos são capazes de superar o controlador clássico de posição PID com ganhos otimizados no curso proposto, enquanto permanece agnóstico em relação a um conjunto de parâmetros de simulação. Finalmente, aplicamos o mesmo algorítmo de DRL para desenvolver um controlador que se utiliza de dados visuais para completar um curso de corrida em uma simulação. Com este controlador, o quadrotor é capaz de localizar portões utilizando uma câmera RGB-D e encontrar uma trajetória que o conduz a atravessar o máximo possível de portões presentes no percurso.

Palavras-chave

Controle de quadrotor; Veículo aéreo não-tripulado (VANT); Aprendizado por reforço profundo; Soft Actor-Critic (SAC); Navegação visual.

Table of contents

List of figures

List of tables

List of Abreviations

ANN – Artificial Neural Network

BDF – Backward Differential Formula

CAM – Class Activation Maps

CNN – Convolutional Neural Network

DDPG – Deep Deterministic Policy Gradient

DDR – Deep Drone Racing

DoF – Degrees of Freedom

DPG – Deterministic Policy Gradient

DQL – Deep Q-learning

DQN – Deep Q-network

DRL – Deep Reinforcement Learning

GPS – Global Positioning System

IMU – Inertial Measurement Unit

LIDAR – Light Detection And Rangefinder

MBRL – Model-Based Reinforcement Learning

MDP – Markov Decision Process

ML – Machine Learning

MLP – Multi-Layer Perceptron

MPC – Model Predictive Control

MSBE – Mean Squared Bellman Error

NeurIPS – Annual Conference on Neural Information Processing Systems

ODE – Ordinary Differential Equation

PID – Proportional Integral Derivative controller

PPO – Proximal Policy Optimization

PWM – Pulse Width Modulation

ReLU – Rectified Linear Unit

ResNet – Residual Network

RGB-D – Red-Green-Blue-Depth image/camera

RL – Reinforcement Learning

RMSE – Root Mean Squared Error

RPM – Revolutions Per Minute

SAC – Soft Actor-Critic

SGD – Stochastic Gradient Descent

TD – Temporal Difference learning

TD3 – Twin Delayed Deep Deterministic Policy Gradient

TRPO – Trust Region Policy Optimization

UAV – Unmanned Aerial Vehicle

VAE – Variational Autoencoder

VODE – Variable-cofficient Ordinary Differential Equation

List of Symbols

Deep learning

$\sigma(\cdot)$ – Activation function

W_i – Activation weights of layer i

b_i – Activation bias of layer i

θ – Network parameters (weights)

α – Learning rate

$\mathcal{L}(\cdot)$ – Loss function

y – Ground truth data

$\hat{y}$ – Network predictions

Reinforcement learning

s – State

a – Action

s' – State resulting from taking action a at state s

$\mathcal{S}$ – State space

$\mathcal{A}$ – Action space

$r(\cdot)$ – Reward/reward function

G_t – Expected returns

R_t – Expected future rewards

γ – Discount factor

$\pi(a|s)$ – Stochastic control policy

$V_\pi(s)$ – Value function under policy π

$Q_\pi(s, a)$ – Q-value/action-value function under policy π

$\mathbb{E}$ – Expectations

$V*, Q*$ – Optimal value function

a' – Next action from s' following a particular control policy

$\mu(s)$ – Deterministic control policy

ϵ – Exploration rate

ϕ – Value function parameters/network weights

$y(\cdot)$ – Value update target

$\mathcal{R}$ – Replay buffer

$\mathcal{D}$ – Minibatch of transitions

θ – Control policy parameters/network weights

τ – Target network update rate

$\mathcal{N}(\cdot)$ – Gaussian probability density function

$\mathcal{H}(\cdot)$ – Entropy of a probability distribution

$\sigma(\cdot)$ – Noise variance of a stochastic control policy

h – Target average entropy constraint

Quadrotor model

T_i – Thrust/throttle of rotor i

Q_i – Momentum of rotor i

X, Y, Z – 3-dimensional axes/positions

$\dot{x}, \dot{y}, \dot{z}$ – 3-dimensional velocities

θ – Roll

ϕ – Pitch

ψ – Yaw

Ω_i – Propeller i angular velocity

d – Propeller diameter

α – Propeller pitch angle

F_p – Force vector generated by rotors

M_p – Momentum vector generated by rotors

L – Length of quadrotor arms

b – Thrust-momentum constant

I – Momentum of inertia

R – Rotation matrix

S – Attitude frame transfer matrix

p – Position vector (X, Y, Z)

$\dot{p}$ – Velocity vector in the inertial frame (dx, dy, dz)

Φ – Attitude vector in the inertial frame (θ, ϕ, ψ)

$\dot{\Phi}$ – Angular velocity/attitude rate $(\dot{\theta}, \dot{\phi}, \dot{\psi})$

ν – Linear velocities vector in the local frame

ω – Angular velocities vector in the local frame

δ – Command vector (thrust, roll, pitch, yaw)

Δ – Allocation matrix

k – PID gain

Valeu a pena? Tudo vale a pena
Se a alma não é pequena.
Quem quer passar além do Bojador
Tem que passar além da dor.
Deus ao mar o perigo e o abismo deu,
Mas nele é que espelhou o céu.

Fernando Pessoa, *Mensagem: X. Mar Português.*

1
Introduction

"Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain."

– *A. M. Turing* [1]

The idea of learning by interacting with the environment is a very well established concept in biology. For millions of years, animal behavioural patterns are passed along generations when offspring observe their parents and try to reproduce their actions. When doing so, the individual is able to connect the causes to the effects of their actions, established by a learning process. This phenomenon is also observable in human children, which inspires Turing's passage regarding machine learning and, subsequently, a number of algorithms in the field.

Reinforcement learning (RL) is a series of machine learning approaches based on interactions with the environment. These algorithms seek to map observable situations to actions, in order to maximize a numerical reward signal, which is given by the environment. In contrast to supervised learning algorithms, RL models do not know *a priori* which actions are the most suitable to achieve the desired behavior. Thus, RL algorithms need to explore possible paths to the final goal by interacting with the environment and observe which of these paths leads to the highest reward over time [2].

Deep learning and robotic control

According to the definition by the International Organisation for Standardisation, robot autonomy is the ability to perform intended tasks based on current state and sensing, without human intervention [3]. For example, in order to consider a flying robot as reactive-autonomous, it should be capable of maintaining the current position or trajectory despite external perturbations, such as winds and obstacles, as well as maintaining a safe, predefined distance from the ground, coordinate with moving objects, take off and land.

Recent advances in computational power and in the efficiency of machine learning algorithms have made it possible to use neural networks for various types of complex applications that require autonomy and comprehensive perception of the environment. These advances made it faster to calculate the output of these networks even in embedded controllers, enabling novel approaches to control stability, navigation, perception, *et cetera*, in mobile robots [4, 5].

The quadrotor is an example of such a robot that is able to move in 3-dimensional space using four rotors, each consisting of a motor and a propeller. This vehicle is open loop unstable during flight, requiring constant actuation to hover and move around without flipping or falling to the ground [6]. Conventionally, underlying PID controllers are used to provide closed loop stability to the quadrotor so that higher level controllers can focus on more difficult tasks [7]. The issue with this kind of controller is that it assumes that the system it tries to control is linear and time-invariant, while the quadrotor has non-linear dynamics and has some time-variant variables, such as battery voltage, motor performance (due to wear or temperature), or even mass, if the vehicle is tasked to interact with a payload. For these uncertainties, a robust PID design strategy [6] and adaptive controller [8] have been employed. These methods, however, require extensive development and knowledge input by an engineer. Reinforcement learning and other machine learning methods allow the computer to learn the control task by itself, without requiring extensive prior proficiency in modelling and controlling the system in question.

Classic closed-loop controllers, or feedback control systems, compute the control actions based on the relationship between the output of the controlled system and a reference input [9]. As universal function approximators [10], artificial neural networks (ANNs) can replace traditional methods of controller design by fitting a function that describes the mentioned relationship between the feedback error and control actions for the controlled system. ANNs employ non-linear features into their hidden layers [11]. As such, they are able to model the non-linear aspects of the quadrotor, conceivably even if these aspects are not directly observable. Neural networks with multiple neuron layers built on top of each other are regarded as "deep" neural networks, thus establishing the process of deep learning [11].

Generally, a neural network is obtained via supervised learning, where training data consists of a paired set of inputs and outputs for the desired function to approximate [11]. However, there is no way to obtain a training set for a low-level quadrotor controller without a fully developed knowledgeable controller. In this case, we investigate an RL-based approach: instead of

finding the desired control output for the network, we need to design a reward function that describes the desired behaviour of the quadrotor, which should be trivial [2], as we could, for example, attribute rewards for successfully hovering and costs to destabilization. On the other hand, there are other challenges considered when employing RL for robotic control tasks. Using ANNs as approximators for value and policy functions causes the learning process to become unstable [5], imprecise [12] and expensive in terms of their sample complexity [13]. The forecited works improve upon the Deterministic Policy Gradient [14] actor-critic algorithm to address these problems and other problems that arise from this form of approximation.

Deep learning and robotic perception

Perception is the ability to become aware of the state of the world through the senses. For a robot, perception means to extract relevant information through electronic sensors. Cameras are widely used in small flying vehicles to extract high level information of the environment due to their reduced mass and energy consumption compared to other methods, such as LIDARs and sonars. Relative to their mass and energy requirements, cameras are able to gather richer information, such as object colours and shapes, and span wider fields of view, that can be further adjusted by using various types of lenses [15]. In addition, even depth information can be extracted from the environment by using specific types of cameras capable to do so, for example, RGB-D or a stereo system, which employs image triangulation to extract this kind of information [16].

There are several methods to extract features from a camera feed. These methods constitute the field of computer vision [17]. More recently, due to the rise in popularity of artificial intelligence algorithms led by the aforementioned advances in computational power, deep learning methods for image processing have been favoured over traditional computer vision techniques [11]. These methods are widely regarded as "intelligent" because they are able to identify underlying patterns in sets of data without the need of mediation by a specialist. By simply "learning" from sets of inputs paired with desired outputs, deep learning methods are capable by themselves of identifying the features necessary to make a decision [11]. This aspect is known as end-to-end learning, in which a complex task is not broken up into smaller components, but, in RL, for example, viewed through the lens of a simple reward [18]. Image-based perception methods include, but are not restricted to, vehicle localization in the environment [19], object detection [20, 21], collision detection [22, 23] and

velocity (magnitude and direction) estimation [24, 25].

The end-to-end learning concept is not restricted only for use in supervised image processing techniques. An important advantage of RL-methods is the production of a control policy without the imposition of explicit rules. As mentioned, it only requires the design of a reward function that roughly describes the final desired behaviour and the algorithm, by itself, comes up with the means to reach this behaviour [2].

Objectives

The objective of this work is to develop and test Deep Reinforcement Learning (DRL)-based navigation controllers that are able to learn to control quadrotors in a virtual environment. The general idea is to address possible mismatches between quadrotor dynamic models used in simulation and the dynamics of a real vehicle. For the first part of this work, we focus on the basic quadrotor structure: the controller should be able to account for differences in quadrotor parameters, such as size and mass, and propeller parameters, such as diameter and pitch angle. Not only this capability of generalization should be useful for smoothing out the transition from simulation to the real world, but also for accounting for additional differences introduced by the performance of various tasks.

For the second part of this work, we focus on dealing with localization based on computer vision. In GPS-denied settings, the quadrotor relies on its own sensors to estimate localization data. In this work, this data is the quadrotor position relative to scene features, in this case, gates in a racing course. This localization data is estimated using an ANN and then used as observations for the RL controller. This form of estimation is often noisy and unreliable without additional filters and data processors. We expect the RL controller to learn by itself to deal with these challenges. For both mentioned parts, the controller development is approached from an end-to-end perspective: we refrain from imposing too much knowledge and supporting features to the controller, instead letting the RL model learn by itself how to tackle the challenges encountered during training.

In this work, we start from developing a simple waypoint guidance controller and improve upon it to perform more complex tasks, such as payload carrying and racing through a gate course. In the end, this controller is coupled with a visual perception model in order to attempt the racing course without externally available positional data. This general objective is divided in two sub-objectives: the development of an agnostic low-level waypoint guidance

controller and the development of a gate-crossing visual navigation controller.

For the first objective, the DRL-based agnostic controller should be able to control quadrotors in simulations with a large range of vehicle parameters, such as mass and propeller size, without retraining the model. Controllers designed upon a single set of vehicle parameters, despite taking advantage of the non-linearity of neural networks, are unlikely to perform as well in widely different simulator parameters. Thus, we introduce an approach that does not rely on a single set of vehicle parameters, but on a wide, randomized, set.

To accomplish that, we employ Soft Actor-Critic (SAC) [13], a recent DRL algorithm that includes entropy maximization during training, which, among other advantages, encourages many alternatives for control, where possible, enabling the adaptativeness necessary to maintain optimal behaviour in such a range of simulation parameters. We develop two kinds of RL-based pose controllers: one that controls directly motor inputs from sensor data and another that controls only the desired velocity, working together with a low-level PID controller that provides the corresponding motor inputs.

Further, we propose a payload pickup and drop course that takes full advantage of the agnosticism of the conceived controller. When picking up or dropping a payload, the quadrotor mass changes suddenly and significantly, requiring the controller to be able to not only maintain flight stability, but must present equal performance in carrying out the remainder of the task.

Ultimately, a robust controller that is able to account for a wide range of quadrotor parameters is not only useful to account for the mentioned disturbances, but also to smooth out the transition from simulation to reality, one of the challenges of low-level DRL controllers [26], as we can train the model to expect the uncertainties of a real quadrotor.

For the second objective, we combine the development of the low-level RL controller with a convolutional neural network (CNN) to come up with a visual perception-based navigation controller to complete a drone racing course of gate checkpoints. The racing courses we attempt to complete are the *Soccer Field* easy and medium courses of *Microsoft*'s 2019 NeurIPS conference[1] challenge.

To accomplish that, we divide the racing task in two sub-tasks: gate localization and checkpoint guidance. Gate localization is done by a CNN trained in a supervised fashion for this task. For checkpoint guidance, we use the same approach proposed for waypoint guidance, we train the quadrotor to follow these checkpoints using the RL algorithm Soft Actor-Critic. Then,

[1] <https://www.microsoft.com/en-us/research/blog/game-of-drones-at-neurips-2019-simulation-based-drone-racing-competition-built-on-airsim/> Accessed: 12 Mar. 2021

both localization and guidance models are combined in one single RL model for further training and fine tuning in the proposed racing courses.

Contributions

While previous works focus on developing controllers for a specific given quadrotor [7, 26–28], often because of restrictions imposed by the simulations used or the employment of real vehicles, this work focus on a generalized parameter-agnostic approach for designing quadrotor navigation controllers using reinforcement learning techniques. This work outlines the capabilities and limitations of such controllers in a variety of tasks, stressing the requirement for such controllers to be:

- Stable and accurate;
- Robust to different quadrotor parameters and to sudden changes of these parameters, without the need to re-train the controller model; and
- Robust to the challenges of a visual localization model (noise and disturbances) and capable to carry out training nevertheless.

We show that it is possible to design a low-level RL controller for a quadrotor that is, at the same time, stable and accurate, and robust to a range of parameters and time varying circumstances.

Previous works in RL-based visual navigation often rely on classic underlying controllers, such as PID, that integrates with high-level decisions provided by the top-level RL controller [22, 23], which is typical for learning-based approaches [7]. In this work, we seek an end-to-end approach that should not rely on this underlying controller. Accordingly, due to limitations of the simulated environment for visual navigation, the RL controller provides the lowest level outputs that can be passed to simulation. By employing this method, the quadrotor is able to detect and cross several gate checkpoints in a racing course using visual data, while also being robust to the disturbances present in visual forms of localization.

These features should be useful not only to carry out the proposed tasks in this work, but to also smooth the transition between simulation and real world.

Dissertation structure

The current introductory chapter aimed to lay out the object of study and outline the objectives and contributions of this work. Chapter 2 seeks

to give the reader an understanding of the discussed topics and to discuss related works and state-of-the-art in the field. Chapter 3 introduces the tasks to be carried out along with the methodology of design, development and benchmarking of each of the controllers developed. Chapter 4 aims to describe in detail the implementation of training and testing environments, along with the implementation of other proposed tasks. In Chapters 5 and 6, the results of the proposed experiments are laid out and discussed, accounting for expectations and comparisons with related work.

2
Background

In this chapter, we introduce in detail the objects of study of this work, namely deep convolutional neural networks, reinforcement learning and quadrotor modelling, and the theory behind them. In the end we discuss existing work with similar objectives to ours and present the state-of-the-art in the field of drone racing.

2.1
Deep learning

Machine learning (ML) is an umbrella term used to describe a variety of algorithms and techniques in which a computer acquires insight into a dataset or a set of observations from an environment. ML methods are categorized primarily with respect to dataset pairing. Supervised methods learn from paired sets of data, in other words, it learns from predefined input-output sets. Unsupervised methods have no outputs paired to inputs and try to learn from the intrinsic patterns of the input set. Reinforcement learning methods, like unsupervised methods, also have no output sets, but rely on the maximization of a reward signal over time [29].

2.1.1
Neural networks

A common method used to implement machine learning models are artificial neural networks (ANNs) [18]. This structure is inspired by the activity of a biological brain consisting of a network of perceptrons, or neuron-like nodes, that emit a signal based on their inputs. These perceptrons are arranged in multiple layers (a multi-layer perceptron, or MLP), where the output signal y_i of each layer is given as a function of its inputs x_i, input weights W_i, an input bias vector b_i and a non-linear activation function σ [18], as defined by:

$$y_i = \sigma(W_i x_i + b_i), \tag{2-1}$$

and illustrated by figure 2.1.

Non-output layers are called hidden layers. Models with many hidden layers, or deep layers, are called deep neural networks. Hence, the set of

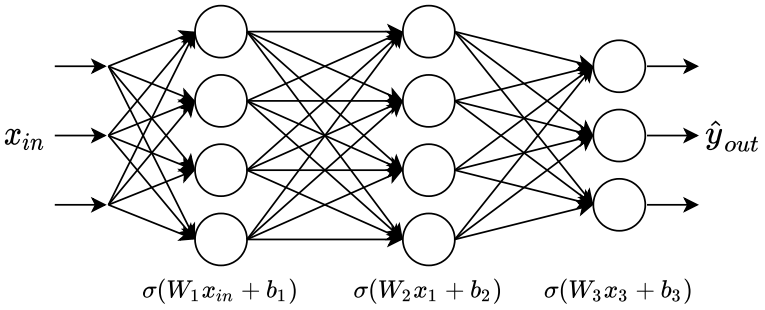


Figure 2.1: Multi-layer perceptron that maps an input array x_{in} to an array of predictions $\hat{y}_{out}$. Each layer has input weights W_i an input bias b_i.

techniques used to train these models make up the field of deep learning (DL) [11].

The model is trained by minimizing an objective function, or loss function $\mathcal{L}$, which represents the error of estimation between the network output and the desired output [18]. The parameters θ of a neural network are its combined weights $[W, b]$, which are adjusted by stepping to the negative direction of the loss function gradient $\nabla_\theta \mathcal{L}(\theta)$ with respect to θ. Mathematically, for a loss function of parameters θ, $\mathcal{L}(\theta)$, the parameters are adjusted by:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta), \tag{2-2}$$

where α is the learning rate term, a positive number. This is the simplest form of stochastic gradient descent (SGD) [30].

For each training iteration, or epoch, this update is performed with SGD, which is itself a multiple-step iterative operation. At each step, the gradient is calculated from the loss sum over a small batch of samples from the dataset, instead of the sum over the entire dataset. This method converges almost surely to a local minimum of any objective function [30].

A common approach used to compute the gradient of such a complex model is **backpropagation** [11]. In this algorithm, the estimation error is propagated backwards through the network, providing the gradient of each perceptron with respect to its weights. This process is done automatically for any model implemented through a deep learning framework, such as *Tensorflow*[1] or *PyTorch*[2].

2.1.2 Convolutional neural networks

Convolutional neural networks (CNNs) have at least one convolutional layer in their architecture, usually at the input side. These layers are specialized

[1] <https://www.tensorflow.org/> Accessed: 12 Mar. 2021
[2] <https://pytorch.org/> Accessed: 12 Mar. 2021

layers that handle spatially correlated inputs, such as images and sound signals.

A convolution operation consists of constructing an **activation map**, where each entry is the dot product between a patch of the input and a filter, or **kernel**, of the same size. Consider an input image X of size $M \times N$ and a filter k [31], the resulting (i, j) entry of the activation map A is computed by:

$$A_{i,j} = k^{\intercal} X_{i,j} + b \qquad \forall\, i \in M,\, j \in N, \tag{2-3}$$

where $X_{i,j}$ is a slice with the same size as k centered in the pixel $[i, j]$ of a padded X. A single convolutional layer usually has many learnable filters, resulting in as many activation maps as there are filters. This convolution operation is illustrated by figure 2.2.

To increase the parameter efficiency of the network, the dimensionality of activation maps is reduced prior to being fed to densely connected perceptron layers. This can be either by striding the convolution operation (applying the filter every n pixels) or using a strided max-pooling layer after a convolutional layer. These layers function just like a convolutional layer, but, instead of calculating the dot product of a subpatch with a filter, the max-pool layer only takes the highest value of each subpatch [31].

Residual blocks

Some CNN architectures may include residual blocks among their hidden layers. These blocks are used to improve training performance of CNNs by addressing the degradation of training accuracy that may arise from architectures with multiple stacked hidden layers [32]. Instead of learning the underlying mapping $F(X)$ from an input x, residual blocks learn the residual

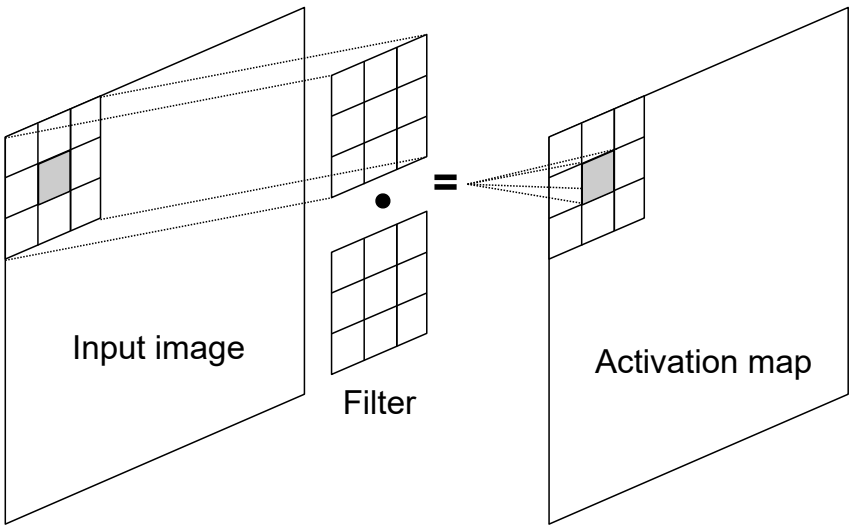


Figure 2.2: Illustrated process of convolution in an entry of an image.

mapping $F_{\text{res}} = F(X) - X$, which is recast into $F_{\text{res}} + X$, where X is brought from the skip connection [32]. In feedforward ANNs, this is done by employing a "skip connection" between the input x and the output of the block, as illustrated by figure 2.3.

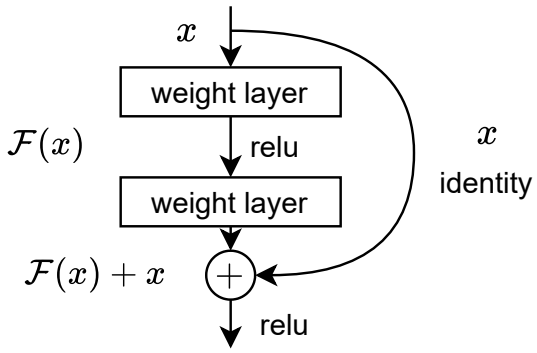


Figure 2.3: An example of a simple residual block [32].

2.2 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning used to obtain a control policy that maximizes a reward signal over time, without any other form of prior knowledge made available.

In this form of learning, an **agent**, which is the system that makes decisions, interacts with the **environment**, which is subject to control, by performing actions ($a \in \mathcal{A}$) and observing the state of the environment ($s \in \mathcal{S}$) prior to the action taken, the state after the action ($s' \in \mathcal{S}$) and a **reward** signal based on these three variables ($r(s, a, s')$), engineered to indicate the desired behaviour of the agent. This relationship is described by the diagram of figure 2.4. With these means, we can estimate the value of each state and take actions that lead up to the states with the highest value.

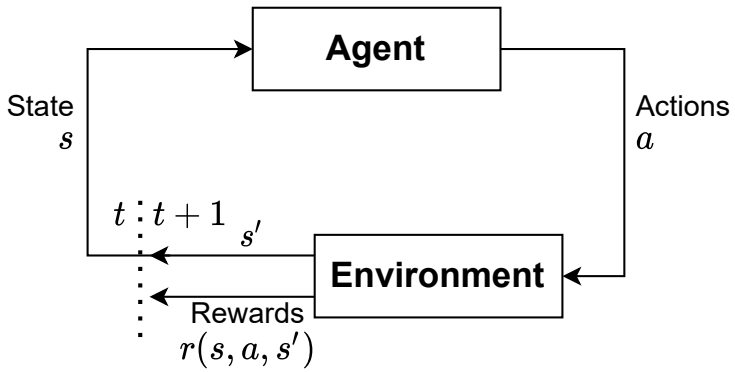


Figure 2.4: Agent-environment interaction.

In this section, we lay out the history of RL theory leading up to the state-of-the-art. Most of the basic RL theory is thoroughly compiled by Sutton & Barto [2]. From Deep Q-learning onwards, we summarize some of the most relevant works in the RL field.

2.2.1 Finite Markov decision processess

Markov decision processes, or simply MDPs, are a formalization of stochastic decision-making processes introduced by Richard Bellman [33]. Initially, the theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. As such, this theory can be employed as a mathematical idealization of the reinforcement learning problem, in which precise theoretical statements can be made. MDPs are used here to describe how actions influence, not only the immediate reward, but the subsequent states and expected rewards [2].

In finite, fully observable MDPs, the transition from state s to a state s', yielding a reward r, and resulting from action a, follows a discrete probability distribution, which can be denoted as $p(s', r|s, a)$. Stochastic processes have the Markov property if this distribution is dependent only on the current state-action pair (s, a), being independent of all previous states visited and actions taken [2].

The function of future rewards that should be maximized by the agent is called the expected return G_t. In finite episodic tasks, this expectation can be undiscounted, meaning that the expected returns at state s is simply the sum of expected future rewards. However, tasks are usually continuing, in other words, they can not be broken into episodes and can continue without a limit. In these cases, the expected return is discounted by a factor $0 < \gamma < 1$, to avoid divergence to infinity [2]. Thus, the expected return in an episode can be described by:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ ... \ = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2-4}$$

A core part of most reinforcement learning algorithms is the estimation of a **value function**, which estimates the expected return G_t defined by eq. (2-4), starting from a state S_t and following a particular control policy. This control policy can be denoted as the probability distribution $\pi(a|s)$ of taking action a given the current state s. The value function of the current state s under a policy π is denoted $V_\pi(s)$. Likewise, the value function is estimated not only considering the current state, but also considering any action that can

be taken in that state and actions taken thereafter, following the considered control policy. When estimating the value function in these terms, it is denoted as $Q_\pi(s,a)$, or Q-value function [2]. Thus, we define $V_\pi(s)$ and $Q_\pi(s,a)$ as:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t=s\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\;\middle|\;S_t=s\right], \end{aligned} \tag{2-5}$$

and:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t|S_t=s,A_t=a\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\;\middle|\;S_t=s,A_t=a\right], \end{aligned} \tag{2-6}$$

where the notation $\mathbb{E}_\pi[\,\cdot\,]$ represents the expected value of an agent under the control policy π.

These value estimations maintain an average of returns for each state visited, converging to the true expected return of a state as the amount of times it is visited converges to infinity [2]. This also would require visiting every state at least once, which is impractical for large state spaces.

A fundamental property of RL is that the definitions presented in eqs. (2-5) and (2-6) satisfy recursive relationships, or, mathematically, $G_t = R_{t+1} + \gamma G_{t+1}$. This way, the value function can be estimated iteratively by **bootstrapping**, which means that the value of each state is estimated based on the value of the successive states, considering the underlying probability distribution of state transitions and the control policy being followed [2]. Thus, the value function can be described as eqs. (2-7) and (2-8):

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t=s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1}+\gamma G_{t+1}|S_t=s\right] \\ &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r+\gamma\mathbb{E}_\pi\left[G_{t+1}|S_{t+1}=s'\right]\right] \\ &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r+\gamma V_\pi(s')\right], \end{aligned} \tag{2-7}$$

$$
\begin{aligned}
Q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t|S_t=s,A_t=a\right] \\
&= \mathbb{E}_\pi\left[R_{t+1}+\gamma G_{t+1}|S_t=s,A_t=a\right] \\
&= \sum_{s',r} p(s',r|s,a)\left[r+\gamma\mathbb{E}_\pi\left[G_{t+1}|S_{t+1}=s',A_{t+1}=a'\right]\right] \\
&= \sum_{s',r} p(s',r|s,a)\left[r+\gamma Q_\pi(s',a')\right],
\end{aligned}
\tag{2-8}
$$

which are the **Bellman equations** for V_π and Q_π, respectively. Many RL methods, such as dynamic programming and temporal difference (TD) learning leverage this process to generate an estimate of the value function. When utilizing bootstrapping, if an episode ends in a certain state s, this is considered a terminal state. In this case, the theoretical subsequent state s' will always be evaluated as 0, since no more rewards are expected thereafter [34].

2.2.2 Temporal difference learning

Temporal difference learning, or TD-learning, is a central idea of reinforcement learning. In these methods, the agent learns directly from raw experience, without requiring a model of the environment. TD methods do not wait until the end of an episode to perform the value update, as it can be done as soon the next state s' and observed reward r are known [2].

It is, however, not possible to accurately estimate the state transition probability distribution to calculate eq. (2-7), from a limited amount of samples. Thus, the value function is updated incrementally, as each state transition is a small sample from the distribution $p(s',r|s,a)$, following the policy distribution $\pi(a|s)$, resulting in the TD update in:

$$
\begin{aligned}
V(s) &\leftarrow (1-\alpha)V(s)+\alpha\left[r+\gamma V(s')\right] \\
&\leftarrow V(s)+\alpha\left[r+\gamma V(s')-V(s)\right],
\end{aligned}
\tag{2-9}
$$

for $V(s)$, with the same process applicable for the Q-value function:

$$
\begin{aligned}
Q(s,a) &\leftarrow (1-\alpha)Q(s,a)+\alpha\left[r+\gamma Q(s',a')\right] \\
&\leftarrow Q(s,a)+\alpha\left[r+\gamma Q(s',a')-Q(s,a)\right],
\end{aligned}
\tag{2-10}
$$

where the factor α is a substitution for the importance-sampling ratio for incremental implementations [2], used here as a form of learning rate. These are the simplest TD update methods, named TD(0), or one-step TD. As pointed

out in the previous section, this update, based in part on an existing estimation, is a bootstrapping method.

TD control methods have different approaches to update the value functions, which can be on-policy or off-policy. When performing the value update in eq. (2-10), on-policy methods sample the Q-value of the next state s', $Q(s', a')$, considering that the next action a' is sampled from the current policy $\pi(a|s)$.

Off-policy methods, however, sample the next action a' from a different policy than the one being followed, usually a theoretical, optimal, **greedy** policy:

$$\mu(s) = arg\max_a Q^*(s, a), \tag{2-11}$$

attempting to directly estimate the **optimal** value, or Q-value functions, respectively:

$$V^*(s) = \mathbb{E}\left[r + \gamma \max_{a'} V^*(s') \middle| s\right], \tag{2-12}$$

and:

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right]. \tag{2-13}$$

The greedy control policy consists of following states with the highest values. This is a deterministic policy, as, at state s, only one action $a \in \mathcal{A}$ has probability $\pi(a|s) = 1$ of being chosen, while others have a null probability. For these policies, the notation $\mu(s)$ is used, instead. However, a purely deterministic policy can lead to poor results, due to a lack of **exploration**. A proper exploration strategy guarantees that the agent will visit states it would not visit otherwise, providing value estimates outside poor local minima [2]. An exploration strategy can be built upon a pre-existing deterministic policy, for example, an ϵ-greedy policy, with distribution described by:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a = arg\max_a Q(s, a) \\ \dfrac{\epsilon}{|\mathcal{A}(s)|}, & \text{otherwise,} \end{cases} \tag{2-14}$$

based on an exploration factor ϵ.

This form of Q-value estimation is called Q-learning, and is considered to be one of the early breakthroughs in reinforcement learning. In this method, the learned action-value function Q, directly approximates the optimal action-value function $Q^*(s, a)$, independently of the policy being followed [2]. A generic off-policy Q-learning routine is described by Algorithm 1.

There is another TD method called actor-critic, which becomes the basis of many modern deep RL algorithms. In these methods, the control policy

Algorithm 1: Generic Q-learning routine for off-policy TD control

Initialize $Q(s, a)$ for all $s \in \mathcal{S}$
repeat
  Sample a from a policy derived from $Q(s, a)$, with a proper exploration strategy, such as ϵ-greedy.
  Take action a and observe s', r.

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

until *s is terminal*;

does not need to consult the value function to follow the highest valued states. Instead, the control policy is estimated separately and is updated at each iteration, usually via a stochastic gradient method. These methods are discussed and detailed in further sections.

2.2.3 Approximation methods and Deep reinforcement learning

Simple tabular value function methods often work for very small learning tasks, with limited space states. However, learning becomes impractical as the state set $\mathcal{S}$ becomes arbitrarily larger and more complex, especially if the state is **continuous**, as even the coarsest discretization methods eventually come across the curse of dimensionality, i.e. the amount of possible states increases exponentially with the amount of dimensions of the state space [2]. For instance, the number of possible 8-bit RGB-colour images of 320×240 pixels is $2^{1843200}$, or, approximately, a *googol* to the power of 1843.

So that learning can be properly carried out in these situations, the algorithm needs to generalize unvisited states by leveraging information from previous experience. This way, it is able to determine an approximate value function even within limited iterations.

There are several well-researched ways to perform this approximation. Parameterized approximation methods estimate the value function employing a feature weight vector ϕ which parameterizes the value function, making it **differentiable** with respect to these parameters. In this form, state-value functions are denoted $V_\phi(s)$ and action-value functions are denoted $Q_\phi(s, a)$ [2], and their gradients with respect to their parameters ϕ are denoted by ∇_ϕ.

The most popular approximation method for modern RL algorithms is the representation by deep neural networks [34]. In this form, the feature weights ϕ are the neuron connection weights, represented in figure 2.5 by the matrices $[W_{HL1}; \cdots ; W_{HLn}]$. Neural networks can be differentiated with

respect to their weights, using a process called **backpropagation**.

2.2.3.1 Playing Atari games

Mnih et al. [35] propose a variant of the Q-learning (DQN) algorithm to teach an agent to play *Atari* games by learning a control policy that maps raw video data to input actions. This variant leverages recent deep learning advancements in computer vision, by training a deep convolutional neural network as an approximation to the action-value, Q, function.

There are several challenges present when integrating deep learning and reinforcement learning. First, supervised DL algorithms require a large number of hand-labeled training data, while, as previously discussed, only a scalar reward signal is made available for RL algorithms, and it is usually sparse, noisy and delayed. Second, while DL training data is assumed to be independent and uncorrelated, RL algorithms usually deal with highly correlated sequences of states as training data. Finally, in RL, the data distribution changes after each iteration as new data is obtained through new control policies, posing a problem for DL methods that assume a fixed distribution [35].

In order to mitigate the problem of data correlation and changing distributions, the algorithm uses an experience replay mechanism [36], which generates minibatches of data from previous iterations of the training routine, smoothing out the sample distribution trough iterations. This is possible because off-policy methods try to directly approximate the optimal Q-value function $Q^*(s, a)$ (eq. (2-13)), which should be able to satisfy the Bellman equation (eq. (2-7)) regardless if sample transitions are obtained from obsolete control policies. These older state transitions are usually uncorrelated to newer samples as a result of being sampled from different control policies, helping to prevent overfitting of the model. Additionally, DQN is able to greatly improve sample efficiency by reusing older state transitions.

The Q-value function is learned by adjusting its weights ϕ aiming

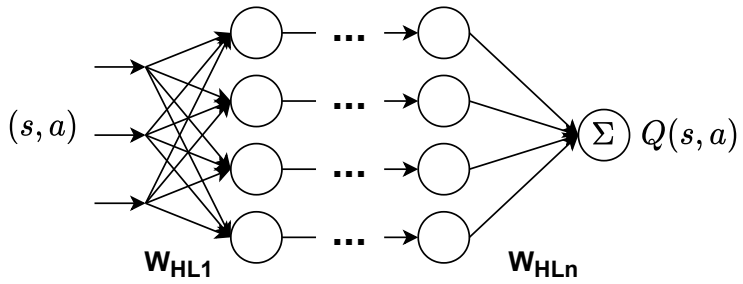


Figure 2.5: Q-value function approximation by an artificial neural network.

at minimizing the mean-squared Bellman error (MSBE), eq. (2-15), using stochastic gradient descent, which updates the value function incrementally with a target $y(r, s')$. For the DQN, this target is:

$$\mathcal{L}(\phi) = \sum_{\mathcal{D}} \left(y(r, s') - Q_\phi(s, a) \right)^2, \tag{2-15}$$

where the transitions (s, a, r, s') are obtained from a minibatch $\mathcal{D}$ sampled from the replay buffer, and:

$$y(r, s') = r + \gamma \max_{a'} Q_\phi(s', a'). \tag{2-16}$$

2.2.4 Actor-critic methods

Applying classic Q-learning methods with greedy control policies (eq. (2-11)) is remarkably difficult for agents with continuous action spaces. An immediate workaround would be discretizing the action space, but one would rapdily hit the curse of dimensionality roadblock, as the amount of possible actions grows exponentially with the dimensions of the action space.

For agents with continuous actions spaces, the control policy can be represented by a parameterized function that can be differentiated with respect to these parameters (θ). This control policy is updated by calculating its gradient, with respect to its parameters, in the direction with the highest probability of returns. Thus, these methods are called policy gradient methods [14].

DQN-based **Actor-Critic** methods are a popular form of policy gradient algorithms [14, 34], in which a differentiable control policy (the actor) is improved based on the judgment of a value function (the critic). The simplest policy gradient computed by this method is:

$$\nabla_\theta Q_\phi(s, \mu_\theta(s)), \tag{2-17}$$

which updates the policy parameters via a deterministic policy gradient (DPG) ascent [14], as to maximize the expected return of the control policy over the states.

Deep Deterministic Policy Gradients (DDPG) [5] and Twin Delayed DDPG (TD3) [12] are widely used actor-critic methods that introduce a number of features to the base DPG algorithm, aiming to improve learning stability, which is usually brittle, and speed up learning.

Replay buffers (DDPG). This is the same improvement introduced by the *Atari* study presented in the previous section. This finite replay buffer takes

advantage of the off-policy charcteristics of DDPG to greatly improve sample efficiency by reusing old uncorrelated state transitions.

Target networks (DDPG). TD methods rely on reusing the same approximation of the value function to calculate their updates. In many environments, this implementation, especially with neural networks, is highly unstable and prone to divergence. To avoid this issue, the value update target $y(s', r)$ is calculated based on separate Q and policy networks that slowly track the most recent corresponding network weights via Polyak averaging with update rate τ:

$$\begin{aligned} \phi_{\text{targ}} &\leftarrow \tau\phi + (1-\tau)\phi_{\text{targ}} \\ \theta_{\text{targ}} &\leftarrow \tau\theta + (1-\tau)\theta_{\text{targ}}. \end{aligned} \tag{2-18}$$

In this way, the update target becomes:

$$y(s', r) = r + \gamma Q_{\phi_{\text{targ}}}(s', \mu_{\theta\text{targ}}(s')), \tag{2-19}$$

and the critic parameters ϕ are updated with one-step gradient descent, based on the loss function shown in eq. (2-15), as described by the update rule in:

$$\phi \leftarrow \phi - \alpha\nabla_{\phi}\mathcal{L}(\phi). \tag{2-20}$$

In both DDPG and TD3, the policy is still updated with the gradient in eq. (2-17), using the most recent weights (ϕ, θ). The target weights $(\phi_{\text{targ}}, \theta_{\text{targ}})$ are usually updated every iteration of the algorithm, with the τ factor set to a value between 0 (slower tracking) and 1 (faster tracking), usually set closer to 0. The TD3 algorithm employs the same optimizations as DDPG, except for not using a target network for the control policy.

Soft policy updates (TD3). In DDPG, Q-function approximation errors can lead to localized narrow peaks in value estimation, which can be exploited by the control policy. To avoid this particular kind of inaccuracy, the algorithm introduces target policy smoothing, which serves as a regularizer, by adding a small random normal noise to the target policy in Q-value updates, as in:

$$a'(s') = \text{clip}(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}),\ \epsilon \sim \mathcal{N}(0, \sigma). \tag{2-21}$$

Note that, with this, the value function is not updated according to the greedy policy anymore.

Double Q-learning (TD3). To address the issue of overestimation bias of the Q-value function, caused by the instability of its estimation, the algorithm learns two concurrent Q-networks, performing the value update using the minimum of these networks, or:

$$y(s', r) = r + \gamma \min_{i=1,2} Q_{\phi_i}(s', a'(s')) \tag{2-22}$$

where $a'(s')$ is sampled from eq. (2-21) and the given update is performed for both main Q-networks.

Delayed policy updates (TD3). In TD3, the "stabilizing" effect caused by the use of a target policy network can be achieved by delaying policy updates with respect to value updates. Despite a slight increase in volatility of value estimates, experiments result in similar and faster convergent behaviours.

2.2.5 Soft Actor-Critic

A more recent iteration of a DQN-based actor-critic algorithm is the Soft Actor-Critic (SAC) [13], which is the learning algorithm employed in this work. The advantage of this algorithm is that it automatically adjusts exploration of the agent by enforcing a maximum entropy target, in other words, a target of maximum "randomness" of the control policy, resulting in a control policy that is able to devise multiple routes to an objective. A version of this algorithm by the same author [37] goes further and automatically adjusts the reward scale of the entropy target, denominated the temperature. By using this method, not only does it become unnecessary to tune exploration, but we can also expect the agent to automatically seek improvement in low confidence subspaces. The enforced entropy objective further speeds up learning compared to previous actor-critic iterations, a crucial feature if one wishes to employ RL on a real-life quadrotor [13].

2.2.5.1 Maximum entropy objetive

The maximization of the entropy is a measurement of the "randomness" of a variable. Consider a random variable x with a density function P. We can compute its entropy $\mathcal{H}$ as [34]:

$$\mathcal{H}(P) = \mathop{\mathbb{E}}_{x \sim P}[-\log P(x)]. \tag{2-23}$$

To enforce entropy maximization, the reward term is augmented with an entropy term $\mathcal{H}(\pi_\theta(\cdot|s))$, which represents the entropy of the distribution

over actions $\pi_\theta(\cdot|s)$ of the stochastic control policy. The augmented reward becomes, at time t:

$$R_t = r(s_t, a_t, s_{t+1}) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t)), \tag{2-24}$$

where the temperature, α, dictates the scale of the effect of the entropy term in the value function and in the stochastic policy. Thus, the expected return following the stochastic policy π becomes:

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t, s_{t+1}) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t)) \right) \,\middle|\, s_0 = s \right], \tag{2-25}$$

for the state value function from a timestep $t = 0$, and:

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t \alpha\mathcal{H}(\pi_\theta(\cdot|s_t)) \,\middle|\, s_0 = s, a_0 = a \right], \tag{2-26}$$

for the state-action value function from a timestep $t = 0$, which includes the entropy bonuses from every timestep except the first [34]. With this, $V_\pi(s)$ can be written in terms of $Q_\pi(s, a)$ [13] by:

$$V_\pi(s) = \mathbb{E}_{a\sim\pi}[Q_\pi(s, a)] + \alpha\mathcal{H}(\pi_\theta(\cdot|s)), \tag{2-27}$$

and the Bellman equations (eq. (2-7) and eq. (2-8)) for Q_π becomes [34]:

$$Q_\pi(s, a) = \mathbb{E}_{s'\sim P, a\sim\pi} \left[r(s, a, s') + \gamma(Q_\pi(s', a') + \alpha\mathcal{H}(\pi_\theta(\cdot|s'))) \right], \tag{2-28}$$

which, by the definition given in eq. (2-27), becomes:

$$Q_\pi(s, a) = \mathbb{E}_{s'\sim P} \left[r(s, a, s') + \gamma V_\pi(s') \right]. \tag{2-29}$$

By the definition given in eq. (2-23), the expectation $\mathcal{H}(\pi_\theta(\cdot|s))$ can be estimated using samples from the policy, thus being rewritten as:

$$\mathcal{H}(\pi_\theta(\cdot|s)) = -\log \pi_\theta(a|s), \qquad a \sim \pi_\theta(\cdot|s). \tag{2-30}$$

By replacing this expectation in eq. (2-28), we can estimate the target of the MSBE loss (eq. (2-15)) of the Q-value function:

$$y(s', r) = r + \gamma \left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a') - \alpha \log \pi_\theta(a'|s') \right), \tag{2-31}$$

where this target value relates to the Q function not including the entropy of the first (current) state (eq. (2-27)), and where a' is sampled from the distribution $\pi_\theta(\cdot|s')$. Note that, in this implementation, target value networks are used for bootstrapping, but not target policy networks. The algorithm is

"soft", since it performs the policy evaluation and improvement steps using its own stochastic policy distribution over actions $\pi_\theta(\cdot|s)$ as a regularizer [34].

The control policy is updated in a way that maximizes the expected returns-plus-entropy over all the visited states, i.e., over a minibatch of states, it should maximize $V_\pi(s)$. Thus, for every update step, the policy is updated via one-step gradient ascent of:

$$\nabla_\theta \left(\min_{i=1,2} Q_{\phi_i}(s', \tilde{a}_\theta) - \alpha \log \pi_\theta(\tilde{a}_\theta|s') \right), \tag{2-32}$$

where $\tilde{a}_\theta$ is sampled from the distribution $\pi_\theta(\cdot|s')$ differentiable with respect to θ [34].

In the implementation of SAC used in this work, the entropy of the stochastic policy is estimated via a Gaussian log-likelihood function combined with an invertible squashing function [13]:

$$\begin{aligned} \mathcal{H}(\pi(\cdot|s_t)) = & \underbrace{-\frac{1}{2} \sum_{i=1}^{\text{len}(a)} \left(\frac{\pi(a_i|s_t) - \mu(a_i|s_t)}{\sigma(a_i|s_t)} \right)^2 + 2 \ln \sigma(a_i|s_t) + \ln 2\pi}_{\text{Gaussian log-likelihood}} \\ & \underbrace{- \sum_{i=1}^{\text{len}(a)} \ln \left(1 - \tanh^2 \pi(a_i|s_t) \right)}_{\text{Invertible squashing function}}, \end{aligned} \tag{2-33}$$

assuming that the underlying distribution of the action vector a is Gaussian and each action a_i is bounded to finite values.

2.2.5.2 Automated entropy adjustment

The temperature α defines the scale of the influence of the entropy term in the reward function. A high temperature causes the algorithm to approximate a high-entropy low-return control policy, while a low temperature causes the opposite. This complex trade-off has different effects with different training environments, usually requiring manual tuning of the temperature for each environment. Further, the temperature requirement depends on the policy, which changes during the learning process. A solution consists in formulating a different maximum entropy reinforcement learning objective, where the entropy is treated as a constraint [37].

In this formulation, the average entropy of the policy is set as a constraint, h, and the temperature α is adjusted by minimizing the loss function:

$$\mathcal{L}(\alpha) = -\ln \alpha (\mathcal{H}(\pi(\cdot|s_t)) + h) \tag{2-34}$$

after performing the value update and policy improvement steps.

As a result, the complex interplay between maximum entropy objective and maximum expected return from the environment is controlled in accordance to a minimum expected entropy constraint, adjusting the scale of "stochasticity" according to the needs of the environment and the exploration requirements [37]. In other words, if the policy entropy is below the constraint, α increases, emphasizing the maximum entropy objective. On the other hand, if the policy entropy is higher than the minimum entropy constraint, α decreases, emphasizing the maximum expected return.

2.3 Quadrotor dynamics

The quadrotor, also known as quadcopter, is a kind of unmanned aerial vehicle (UAV, also commonly known as **drone**) that generates lift using 4 rotors, one at the extremity of each of its arms. Most quadrotors have these rotors distributed symmetrically in a plus (+) shape or in a cross (×) shape. Figure 2.6 shows a cross-shaped quadrotor. Despite the latter being more common for commercial drones, the quadrotor used in the first part of this work is plus-shaped, as we found to be easier to model mathematically.

A rotor corresponds to a motor-propeller pair. It rotates in a single direction, generating a thrust (T) upwards and momentum (Q) around it, against the direction of rotation. In order to prevent the quadrotor spinning out of control due to the sum of momentums, 2 of the rotors rotate clockwise while the other 2 rotate counter-clockwise [38].

While capable of movement in 6 degrees of freedom (X, Y, Z and rotation around each of these axes), the quadrotor only actuates directly in 4 degrees of freedom: Z (thrust), roll (θ), pitch (ϕ) and yaw (ψ) torques. Roll is the rotation around the X axis, used to move along the Y axis. Pitch is the rotation around the Y axis, used to move along the X axis. Yaw is the rotation around the Z axis [38].

Model training and experiments are performed with a simulated quadro-

Figure 2.6: The Parrot Bebop is one of the most commonly known commercial drones.

tor based on the dynamic model of a simplified structure illustrated by figure 2.7. The central body of the quadrotor, or the hub, which houses the battery and all the electronic components of the vehicle, as well as any extra payload, is approximated by a sphere. Each of its arms is approximated by a thin rod with no mass and its rotors (not shown in the figure) approximated by point masses [39].

While the local and inertia reference frames most used for quadcopter dynamics modelling is NED (north-east-down), where the axes X, Y and Z point to, respectively, front, right and down, the formulation used in this work uses the NEU reference frame (north-east-up), so the local reference frame is X pointing to the front, Y pointing to the right and Z pointing up, as indicated by the diagram.

The thrust T_i produced by each rotor results from the relationship between rotor velocity and propeller parameters described by:

$$T_i = 4.392 \times 10^{-8}\ \Omega \frac{d^{3.5}}{\sqrt{\alpha}}\ (4.23 \times 10^{-4}\ \Omega\ \alpha), \tag{2-35}$$

based on the aerodynamics of a dual-blade propeller with diameter d and angle of attack defined by a "pitch" (α) calculation [40]. In this model, the propeller pitch is the theoretical travel distance of the propeller after **one full revolution**, assuming that the leading edge of the blades have an angle of attack of 0° with respect to the airflow (no slip), as indicated by the diagram shown in figure 2.8. Both propeller parameters are measured, in this case, in inches.

In the same equation, Ω is the desired rotor velocity (in RPM, revolutions-per-minute), the direct input of this model, assuming the electric motor subsystem dynamics are negligible with respect to the overall system.

The force F_p applied by the rotors to the quadrotor in the local frame is

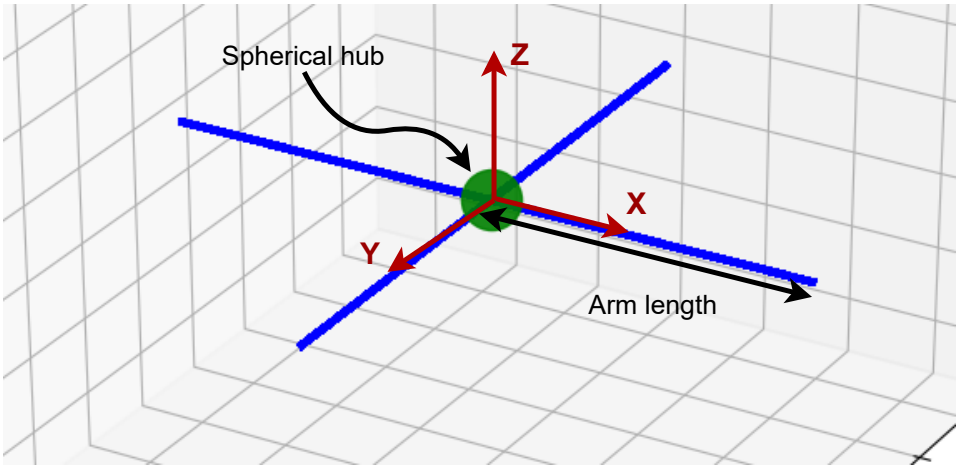


Figure 2.7: Simplified model of the quadrotor used in simulation.

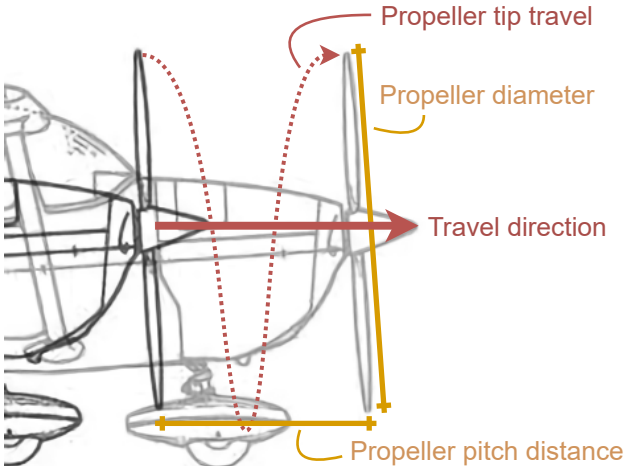


Figure 2.8: Diagram of propeller parameters: diameter and pitch, represented by a Cessna C-172M propeller [41].

given by:

$$F_p = \begin{bmatrix} 0 \\ 0 \\ \Sigma_{i=1}^{4} T_i \end{bmatrix}, \tag{2-36}$$

and the momentum M_p applied by the rotors is given by:

$$M_p = \begin{bmatrix} L(T_1 - T_3) \\ L(T_2 - T_4) \\ b(T_1 - T_2 + T_3 - T_4) \end{bmatrix}, \tag{2-37}$$

where each rotor is indexed as indicated by figure 2.9, and where L is the length of the quadrotor arms and b is an appropriately dimensioned constant that describes the rotor torque linearly with respect to the rotor thrust [42] (in this case, $b = 0.0245$).

The moment of inertia of the simplified quadrotor model is given by

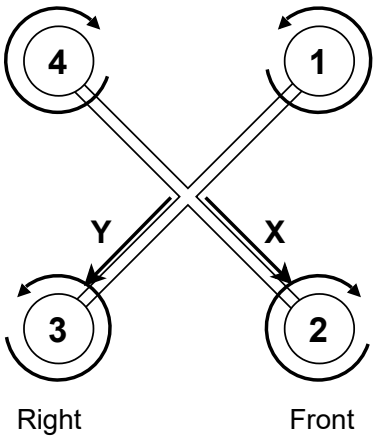


Figure 2.9: Indexation of each rotor for total momentum calculation with indicated direction of rotation.

matrix I. In this simplification, the quadrotor hub is represented by a dense sphere with mass M and radius r, surrounded by point masses m at a distance ℓ from the center, representing the motors [39]. Hence, I results in a diagonal 3×3 matrix given by:

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}, \tag{2-38}$$

where its diagonal entries given by:

$$\begin{aligned} I_{xx} = I_{yy} &= \frac{2}{5}Mr^2 + 2m\ell^2 \\ I_{zz} &= \frac{2}{5}Mr^2 + 4m\ell^2. \end{aligned} \tag{2-39}$$

The matrix R is the rotation matrix from the local reference frame to the inertial frame, which describes the relationship between the quadrotor velocities ν, in the local frame, and $\dot{p}$, in the inertial frame:

$$R = R_x R_y R_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\theta & -s_\theta \\ 0 & s_\theta & c_\theta \end{bmatrix} \begin{bmatrix} c_\phi & 0 & s_\phi \\ 0 & 1 & 0 \\ -s_\phi & 0 & c_\phi \end{bmatrix} \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2-40}$$

The matrix S is the representation Jacobian which describes the relationship between the quadrotor angular velocities ω, in the local frame, to the angular rates $\dot{\Phi}$, in the inertial frame:

$$S = \begin{bmatrix} 1 & s_\phi t_\theta & c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi / c_\phi & c_\phi / c_\theta \end{bmatrix}. \tag{2-41}$$

where s_*, c_* and t_* are compact notations for, respectively, $\sin(*)$, $\cos(*)$ and $\tan(*)$.

Finally, the set of differential equations that describe the dynamics of the quadrotor in the local and inertial frames are described by the set of equations [42]:

$$\begin{aligned} \dot{\nu} &= -\omega \times \nu + R^\intercal g + F_p/m \\ \dot{\omega} &= -I^{-1}\omega \times I\omega + I^{-1}M_p \\ \dot{p} &= R\nu \\ \dot{\Phi} &= S\omega. \end{aligned} \tag{2-42}$$

The observation array used as input to the reinforcement learning controllers is the same state space array $[\nu, \omega, p, \Phi]$, where:

– ν and ω are, respectively, the linear and angular velocities of the quadrotor in the local frame, and
– p and Φ are, respectively, the quadrotor position in the setpoint frame, also referred as X, Y and Z positions, and its attitude in the setpoint frame, which is its angular position around axes X (roll), Y (pitch) and Z (yaw).

2.3.1 Linear adaptation

PID controllers are originally designed for controlling linear time-invariant systems. Because most practical systems are non-linear, such as the quadrotor, some adaptations are needed for the controller and/or for the plant. A straightforward adaptation commonly used is a linear approximation by first order Taylor series, where the system is modelled as a linear function around a reference point. For quadrotors, this reference is the hover point, where all velocities and attitude angles are zero [38].

In the linearized model of the quadrotor, movement in each of the 6 degrees of freedom become uncoupled double integrator subsystems, allowing to design a controller for each subsystem separately as a single input single output (SISO) plant [38]. Then, the desired control action, δ, calculated as throttle, roll, pitch and yaw torques, can be allocated directly to individual motor velocities Ω, via a control allocation matrix Δ:

$$\Omega = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 1 \\ 1 & 0 & -1 & -1 \end{bmatrix} \delta, \tag{2-43}$$

specific for the +-shaped quadrotor.

2.4 Related work

In this section, relevant work in the field is reviewed, that motivate the development of this project.

2.4.1 Deep reinforcement learning for control applications

Deep reinforcement learning (DRL) control applications aim to develop a controller that fills in the same positions as classic PID and similar controllers. Control tasks are sub-divided in two levels: high-level control and low-level control, in which a controller can fulfill either or both.

High-level controllers fit into the external loop of a PID controller, so that they do not interact directly with actuators, as the underlying dynamics of quadrotors are notoriously difficult to control, especially if its parameters are unknown or difficult to accurately measure. High-level control tasks often rely on a pre-existing stable underlying on-board controller that manages the low-level control loop [7].

Low-level controllers map the decisions taken by high-level controllers to actual commands to the quadrotor actuators, as throttle-roll-pitch-yaw (throttle-RPY) forces and torques, which are allocated to each motor, or controlling motors directly. As a control task traditionally fulfilled by PID controllers, some work has been developed in order to bring RL-trained controllers to fit into this low-level control gap.

In our work, and in [27, 28, 43], the developed RL-controller fulfills both low and high level control loops, as a learning-based controller is able to grasp the full dynamics of the quadrotor.

Model-based RL (MBRL) methods have been employed in training with real vehicles in the mentioned references. However, when working on real hardware, sensors often present noise and drift in their measurements. The controllers found in these works have shown to be able to handle this issue by constructing a latent state-space model, from observational data [43], and a model predictive controller (MPC) [7]. The controller developed in [43] tracks a randomly placed waypoint in an enclosed space, actuating directly in unallocated thrust forces and RPY torques, while the controller developed in [7] aims to perform hover control, actuating directly in motor PWM commands.

These approaches are shown to be very sample efficient, by generating their own simulated data based on real dynamics, while also being able, with proper modifications, to account for sensor noise and disturbances. The resulting predictive model, however, is fine-tuned for use with a particular quadrotor configuration, with little known flexibility of use with different configurations, which a robust controller aims to achieve.

Model-free methods have also been investigated for the presented task. Previous works, such as [28], have successfully been able to develop a low-level controller using a variation of classic RL algorithms within simulated environments, capable of waypoint tracking and fast vehicle stabilization. The RL algorithm developed in [28] is an adaptation of deterministic policy optimization (or gradient, such as DDPG) with optimizations assuming the training environment is deterministic. This implies the quadrotor parameters are always constant, e.g. the same torques applied to the airframe produce the same accelerations. This, however, is not the case for the environments explored

in this work, which are stochastic with respect to quadrotor parameters for the first part (waypoint guidance and payload pickup and drop) and with respect to sensor disturbances for the second part (visual navigation). In such situations, stochastic gradient methods seem to display better performance than deterministic gradient methods [28].

More recently, actor-critic algorithms have been showing good learning performance, while being easier to implement and requiring fewer hyperparameter adjustments. Some examples of recent work using such algorithms are [27], which investigates the use of TD3 for waypoint tracking, and [26], which investigates the use of DDPG, TRPO and PPO for attitude control.

In [26], the author describes three open challenges in RL for attitude control (but applicable for many RL robot control tasks): precision and accuracy, robustness and adaptation, and reward engineering. The first challenge is the main focus of that work and [27], the development of a RL controller that works extremely well in a very specific task by a specific quadrotor configuration that aims to achieve a very strict control policy, which the author claims to be the appropriate approach to such a time-sensitive task as the attitude control.

Our work, however, aims at the second challenge when designing the robust waypoint tracker. We show that it is possible to design a low-level RL controller for a quadrotor that is, at the same time stable, accurate and robust to a range of parameters and time varying circumstances. This is achieved by leveraging the entropy enforcement introduced by the novel Soft Actor-Critic algorithm. As previously mentioned, the entropy enforcement allows the learning-based controller to devise multiple routes to an objective, achieving great adaptability when the quadrotor dynamics become unpredictable.

2.4.2 Deep reinforcement learning for visual navigation

A robot navigating through enclosed and cluttered environments usually requires high level sensors to complement the sensors that falter in these cases. This is because these environments present a series of challenges compared to their open-air counterparts: these environments are tight and crowded with obstacles; indoor environments are usually GPS-denied, requiring an internal odometry estimation; and cluttered environments have too much information to be captured by simple sensors, like LIDARs, requiring high level sensors that are able to extract richer information. Monocular, stereo and depth cameras provide high level information about the environment, while being usually cheaper and lighter than other available alternatives, becoming common for

navigation tasks in quadrotors and other mobile robots.

Modeling the value function as a CNN has been shown to yield satisfactory results [35], allowing the development of RL-based robot navigation controllers for the mentioned tasks.

DQN-based navigation controllers have been successfully developed for a visual obstacle avoidance controller for a quadrotor [22, 23]. The vehicles in these studies use a depth camera to generate observations of the environment, with [22] using a 4-timestep buffer of images. However, control policies used with DQN algorithms are discrete. In the mentioned works, the DQN navigation controllers are used together with low level classic controllers that interconnect the high level decisions taken by the DQN to motor commands. Additionally, these high level decisions are discretized, meaning that the control policy loses capability of exploiting control strategies outside the predefined discrete actions. A control policy that outputs decisions in the continuous domain can be more adequate for use with quadrotors. For example, actor-critic methods have been successfully employed for control tasks with visual inputs [44, 45].

In case of [44], DDPG has been used for teaching a car to drive around a simple track. Although it is not a quadrotor being controlled, the mentioned work provides relevant insight on speeding up the learning process for real time control tasks. The observations of the environment do not rely only on the camera images, but also on numerical sensor data of the vehicle. Additionally, it has been shown that the convolutional layers can be pre-trained on a dataset and transferred to the RL model for a major improvement in convergence speed, as these layers are able to abstract high level features of the environment from the first episode. The convolutional layers were pre-trained as a variational autoencoder (VAE) [46, 47], an unsupervised deep learning algorithm with the purpose of encoding an input into a latent space.

Regarding the specific race course used for visual navigation training in this dissertation, there is hardly any research available on proposed control solutions. The only solution that uses DRL is [45], which employs two cooperating models: a segmentation model and an RL control policy. The segmentation model is based on an object detection CNN, using monocular camera RGB images as input, which feeds the rough gate position to a pre-trained actor-critic model. Two controllers are trained with different action spaces: one that outputs RPY and throttle setpoints and other that outputs the parameters of a spline trajectory, the latter performing better than the former. Both controllers rely on underlying low level controllers, that translate these outputs into motor velocities. The mentioned work, however, provides little

insight on some of the strategies employed, such as the choice of a convoluted reward-driven training process for the segmentation model and the actor-critic pre-training process.

As previously mentioned, our work aims to develop a controller that outputs commands directly to the lowest level possible. From the cited works, the lowest level of control is attitude position setpoints [45]. Due to limitations of the visual navigation simulator, the lowest level of control are attitude velocity setpoints, which is still a "step" lower than position setpoints, used in the mentioned work. Additionally, even if not aiming for an agnostic controller for visual navigation, using SAC for learning this controller is advantageous for a racing course, considering that the quadrotor attitude, distance and velocity with respect to each gate can vary a lot.

2.4.3 Other control and navigation solutions

The works described in this subsection aim to undertake similar tasks using non-RL control solutions, however still using deep learning techniques.

2.4.3.1 Deep drone racing

Deep Drone Racing (DDR) [48–50] is a large scale project to develop a racing controller for quadrotors capable of challenging human pilots and was largely an inspiration for the development of the second part of this work. Racing tracks are usually cluttered, GPS-denied and partially-known. At the same time, fast-moving quadrotors often provides noisy and unreliable IMU readings. DDR aims to combine deep learning gate detection and position estimation with a state-of-the-art trajectory generation and tracking algorithm.

The gate detection system is designed with the sparsity of training data of a racing track in mind. The deep CNN in charge of estimating the gate positions also provide the controller with an uncertainty estimation, generating a coarse map of gate locations [49, 50]. The detection model is based on the DroNet architecture [51].

These predictions are incorporated into the trajectory controller using an extended Kalman filter. A minimum jerk trajectory is generated using the coarse map of gate locations and model predictive control, handling for gate displacement during the course [49, 50].

However, even if the proposed detection model is designed to be sample efficient, it still requires large amounts of labeled data for the specific racing track of interest [49].

2.4.3.2 DroNet

DroNet [51] is a CNN architecture, illustrated in figure 2.10, based on the ResNet architecture [32], employed throughout this work to perform visual localization estimation. This CNN is developed originally aiming to learn a visual navigation controller for collision avoidance, that integrates with other navigation controllers. This controller is trained in a supervised fashion with a dataset consisting of collision sequences and potentially dangerous situations, paired with desired steering angles for the quadrotor and the probability of collision. The aim is to trade-off detection performance for processing time, while still maintaining reliable collision avoidance [51].

In the DroNet architecture, an input image passes through 2 convolutional layers before being fed to the residual blocks. First, a convolutional layer with 32 filters of size 5×5 and a 2×2 stride with no activation. Then, a max-pooling layer with a filter of size 3×3 and a 2×2 stride, with ReLU activation.

The architecture has 3 residual blocks, each consisting of two convolutional layers with filters of size 3×3, the first layer with a 2×2 stride and ReLU activation and the second layer with normal striding and linear activation. The skip connection of each block consists of a max-pooling layer with filters of size 1×1 and a 2×2 stride, with no activation, essentially downsampling the input map to match the dimensions of the second layer output. Finally, this output is added to the output of the second convolutional layer and passes through a ReLU activation layer. Each residual block has, in order, 32, 64 and 128 convolutional filters for every convolutional layer and skip connection.

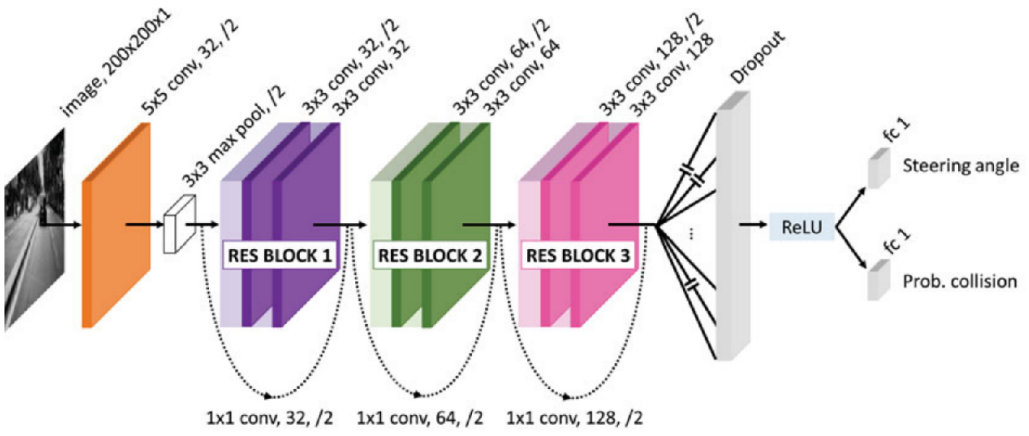


Figure 2.10: DroNet architecture from the original paper [51]. Each convolutional layer is annotated with its filter size, amount of filters and a "/2" notation indicating a 2 × 2 stride.

Finally, the output of the last residual block is flattened and fully connected to a 6 neuron feature layer with linear activation (contrary to the 2 neuron output of the original architecture depicted in figure 2.10). These connections have an 20% dropout rate during training and no dropout during evaluation.

To understand the decisions taken by the controller and which are the relevant characteristics of a frame for this decision, a technique based on class activation maps (CAM) called Grad-CAM [52] is used. This technique is based on a generalization of CAM, used as a way to provide visual explanations on the inner calculations of CNN-based architecture. This method uses the gradient flow into the convolutional layers to localize which parts of an input image have most relevance for the final decisions taken by the model.

3
Methods

In this work, we explore different kinds of controllers drawn from some of the tasks described in the previous chapter. First, we develop waypoint guidance controllers that fit in both high and low level control loops. Then, building upon the waypoint guidance, we integrate image processing techniques to obtain the waypoint visual features from the environment.

In this chapter, we describe the methodology of design, development and benchmarking of each of the controllers developed.

3.1
Low-level control

The first task of this work is to develop four different types of waypoint guidance controllers, that map the position error from sensor data to motor velocities. The first is a fully learned controller, trained in different environments where some quadrotor parameters, namely mass, hub size, arm length and propeller diameter, are randomized within predefined boundaries. This controller employs only reinforcement learning to find a control policy that directly maps the quadrotor velocities and positions to the desired motor velocities. The second is a cascade controller, that delegates the low-level control loop to a PID velocity controller and only learns the high-level position control, also trained in environments with randomized quadrotor parameters. Finally, we use two additional controllers to benchmark these control policies: a learned controller trained on a single environment with fixed quadrotor parameters, and a simple PID position controller with optimized gains. In this section, we describe how each of these controllers are designed and how they integrate with the overall system.

As mentioned, each controller should be able to control quadrotors with different sets of physical parameters. In this section, each controller is developed considering quadrotors with randomized parameters, as well as a set of fixed parameters for benchmarking. This randomization ensures the agnosticism of the controller with respect to quadrotor parameters, as it does not overfit to a single dynamic model, which happens without this randomization.

3.1.1 PID gains optimization

Usually, the PID gains for the quadrotor controller are found via manual tuning or via linear model analysis [53]. However, by employing these methods, it becomes increasingly difficult to find gains appropriate for a quadrotor with constant parameters, and even more difficult if the same controller is used for quadrotors with different sets, or varying, parameters, which is the final objective of the controllers described in this section. This occurs due to the curse of dimensionality: the quadrotor is able to move with six degrees of freedom (DoF), so that, to control movement in every DoF, with 6 uncoupled controllers with proportional, integral and derivative gains each, the final controller would have a total of 18 unique gains to be tuned. A faster, and optimized, way of tuning PID gains consists employing an automatic search strategy.

PID gains for both the pose and velocity PID controllers are found using a simple hill climbing search strategy [54]: starting from a manually estimated set of parameters, the search algorithm takes a step into a random direction in the parameter space (a vector of PID gains) and reevaluates the controller, taking the step if the performance is improved or returning to the previous value if not. This is done multiple times until the evaluation reaches an apparent local maximum, in this case, when performance is not improved after a number of search steps. Further details on the implementation of this search algorithm are laid out in Chapter 4.

Previous works have employed different search strategies for finding optimized PID gains shown to be more efficient, such as genetic algorithms [55]. RL could be used also for automatically tuning PID gains [53]. However, developing such methods is beyond the scope of this work.

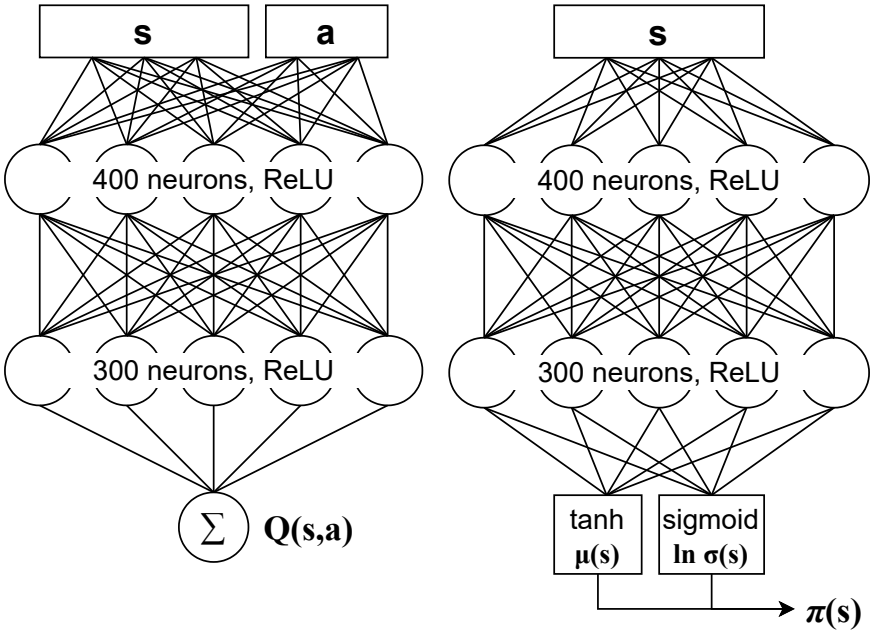
3.1.2 Learning algorithm

As previously mentioned, both learning-based controllers are obtained using the Soft Actor-Critic (SAC) algorithm, with automatic reward scale adjustment. This algorithm was chosen for a number of reasons, the most important being that, as an off-policy algorithm, with automatically adjusted exploration, it benefits from greater sample efficiency than previous actor-critic algorithms, leading to faster learning. Furthermore, it requires less hyperparameter tuning by the user, and we expect that the maximum entropy RL feature, presented by the algorithm, to help the control policy to easily adapt to different quadrotor parameters.

One of the features introduced by SAC is the automated adjustment of the exploration rate. The algorithm introduces this feature by learning the stochastic policy directly. Due to the entropy regularization present in the algorithm, the stochastic policy produces a different exploration rate on a per-state basis, stimulating exploration in regions with a larger flexibility of actions, while reducing exploration in states with less flexibility of actions.

There is a total of five neural networks implemented in this algorithm, which are the two Q-networks, for clipped double Q-learning, two target Q-networks, that track the other two Q-networks, and one for the control policy. All the networks are implemented with two hidden layers of 400 and 300 hidden neurons, using a rectified linear unit (ReLU) as activation function, as illustrated by the architectures in figure 3.1, which is a standard architecture for benchmarking RL algorithms, and is able to cope with a variety of problems with a single structure [28]. The critic networks have no activation function in their outputs and can be evaluated directly as $Q_i(s, a)$. The actor network outputs, however, the parameters for the control policy: $\mu(s)$, with a tanh activation function, and $\ln \sigma(s)$, with a *sigmoid* activation function.

In the actor network, $\mu(s)$ is the mean of the stochastic control policy. The $\ln \sigma(s)$ output is normalized between a pre-set range of $[-9; 2]$ and used to add a Gaussian exploration noise ϵ to $\mu(s)$, with mean 0 and variance equal to the exponential of the normalized output, resulting in the preliminary stochastic



(a) (b)

Figure 3.1: Network models that make up the Soft Actor-Critic, with hidden layer size and activation functions of each layer. (a) Critic network. (b) Actor network.

policy in:

$$\pi(s) = \mu(s) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma(s)). \tag{3-1}$$

The functions $\pi(s)$, $\mu(s)$ and $\sigma(s)$ are then used to estimate the entropy of the stochastic control policy as described by eq. (2-33). Finally, $\pi(s)$ is clipped using a tanh function and both it and $\mu(s)$ are scaled to the environment action range, resulting in, respectively, a stochastic and a deterministic control policy usable for training and testing.

In order to simulate the learning process that would take place with an actual quadrotor, the neural network training is done off-line, with a fixed amount of gradient steps, after an episode has ended, regardless of the episode duration, as each update would take too long to be fitted in between live control actions.

The pseudocode for the implementation of SAC used in this work is given by Algorithm 3, found in the Appendix A.

3.1.3 Controller integration

In the same fashion as a traditional PID controller, the learning-based controllers are guided by a waypoint by minimizing the position error of the quadrotor with respect to this waypoint. The diagram in figure 3.2 illustrates how the learned control policy ($\mu(s)$) fits in the full controller, which works in a simple feedback loop, where this $\mu(s)$ block is an implicit ANN structure, shown in detail in figure 3.1b.

In the diagram, the error tuple that we are aiming to minimize is the 3-dimensional position (x, y, z) and the vehicle yaw (ψ). At the same time, this controller receives the additional IMU data required to minimize attitude angles and better observe the current state of the agent. The IMU provides, additionally, the quadrotor roll (θ), pitch (ϕ), linear velocities $(\dot{x}, \dot{y}, \dot{z})$ and angular rates $(\dot{\theta}, \dot{\phi}, \dot{\psi})$. The fully learned controller outputs directly individual

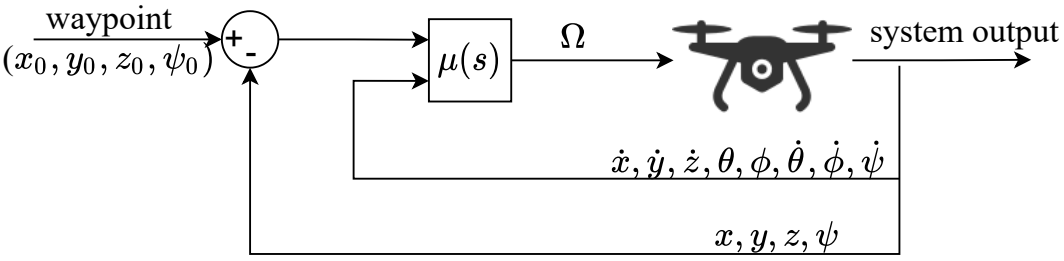


Figure 3.2: Control diagram for the fully learned controller. The $\mu(s)$ block represents the ANN trained using the SAC algorithm.

motor velocities (Ω), normalized between -1 and 1. Note that this controller does not require command allocation to the rotors via the allocation matrix Δ, since it outputs motor velocities directly.

Figure 3.3 shows the control loop of two cascade PID controllers, that compose the pose PID controller used in this work. This pose PID controller consists of two control loops, as it is customary for quadrotor controllers. The external loop calculates the pitch and roll targets (θ_t, ϕ_t) and throttle force (δ_{throttle}) of the vehicle necessary to navigate towards the waypoint. A faster internal loop calculates the lower level attitude commands δ_{att}, needed to track the attitude targets, consisting in roll, pitch and yaw torques. Then, these commands are concatenated with the throttle command, before being allocated to individual motor velocities Ω via the control allocation matrix Δ.

The cascade controller used in this work combines a learned controller $\mu(s)$ and a PID controller, as described by the diagram shown in figure 3.4. This controller receives the pose error as input, along with the additional IMU data, in the same way as the fully learned controller presented in figure 3.2.

The idea of this controller is to reproduce the control stack of commercially available UAVs. Attitude setpoints, used as output of the top-level controller of the pose PID stack (figure 3.3), are usually reserved for aerobatics and "sport settings" of these drones. Instead, existing RL control research opt for "higher" levels of control, such as a discrete displacement step used with DQN-based controllers [22, 23]. To make use of the continuous action space control capability of actor-critic methods, we opt for a control stack similar to the *Tello* drone, also employed in RL research [56], which takes, as inputs, the desired 3-dimensional linear velocities ($\dot{x}$, $\dot{y}$ and $\dot{z}$ summarized in the diagram as ν_t) and yaw rate for the quadrotor ($\dot{\psi}_t$), while an underlying internal controller calculates the low level commands needed to navigate towards the given targets.

The forementioned underlying controller is implemented as a velocity PID controller built with 2 control loops, equivalently to the strategy adopted

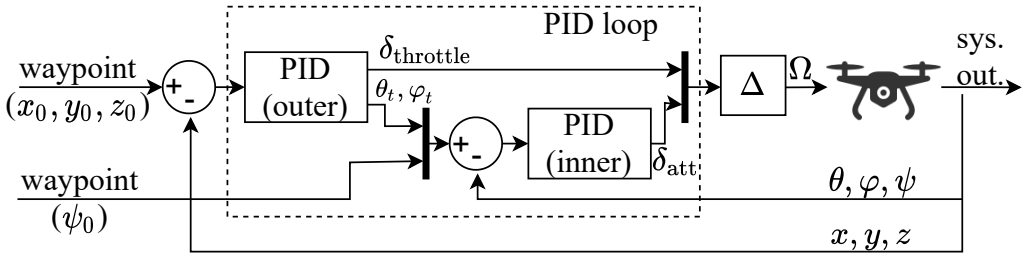


Figure 3.3: Control diagram for the pose PID controller. The $\mu(s)$ block represents the ANN trained using the SAC algorithm.

in the diagram show in figure 3.3, here summarized by a single "PID controller" block. This inner PID loop returns throttle force and roll, pitch and yaw torques as a command vector δ, allocated to individual motor velocities Ω via the control allocation matrix Δ.

All the proposed controller loops, during experimentation, operate at a 40 Hz sampling rate.

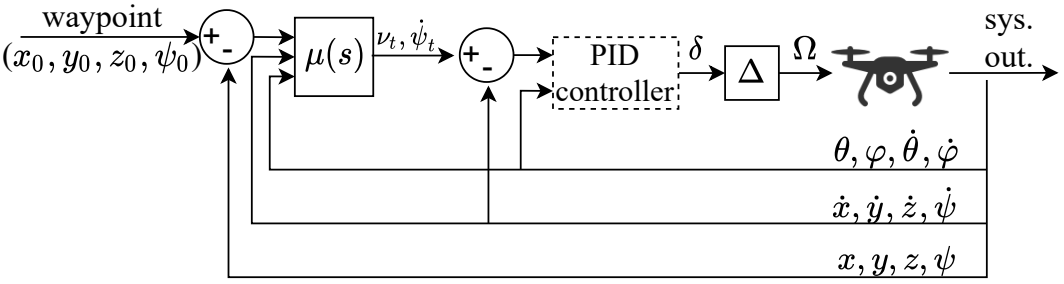


Figure 3.4: Control diagram for the cascade controller: a learned external high-level controller and a low-level PID loop for velocity control, with an implicit structure similar to the diagram in figure 3.3. The $\mu(s)$ block represents the ANN trained using the SAC algorithm.

3.2 Visual navigation

The second task of this work is to develop an RL controller that visually detects a square gate in the environment and travels in its direction. This controller is similar to the waypoint guidance controller developed in the previous section, since it tries to navigate towards the center of the gate based on sensor data, with the exception being the use of a $\times$-shaped quadrotor, which is the only shape provided by the simulator used in this part. The controller has access to the quadrotor IMU only for velocity and attitude readings. On the other hand, it needs to extract relative position data from an RGB-color camera and a depth camera. Thus, except for the addition of convolutional layers, the development process for this control model is the same as the fully learned controller.

Image processing is carried out via a convolutional neural network, which connects directly to the input of the underlying learned controller, with its weights adjustable with the same RL training method.

3.2.1 Baseline model

The baseline model serves as a standard of learning performance for an ideally accurate gate position estimator. For this model, the training is

performed with ground truth gate features together with IMU data. The feature vector of relevant gate position information are:

- Gate center position in the camera frame (X,Y);
- Gate distance from the camera;
- Gate rotation with respect to the camera plane; and
- Position in the camera frame of the subsequent gate (X,Y).

The position of the subsequent gate is added to the feature vector expecting that the quadrotor will face it as soon as the current focused gate is crossed. If the subsequent gate is not visible, this position is set to the origin of the camera frame, or $[0, 0]$, anticipating that the quadrotor will cross the current gate facing forward.

Figures 3.5 and 3.6 show the visualization of the 6 key gate features. In the leftmost picture of figure 3.6, the filled dot is the position of the next gate and the hollow dot is the position of the subsequent gate.

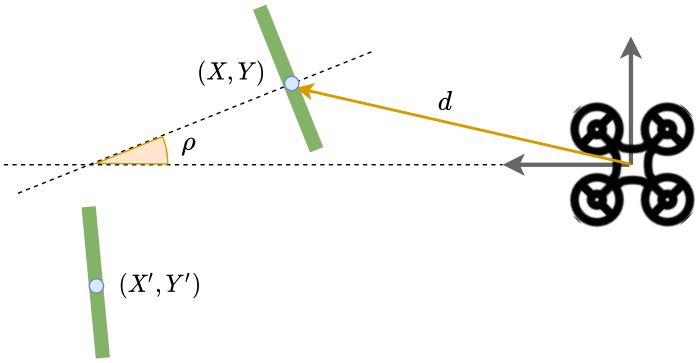


Figure 3.5: Top-down view diagram representing the gate features.

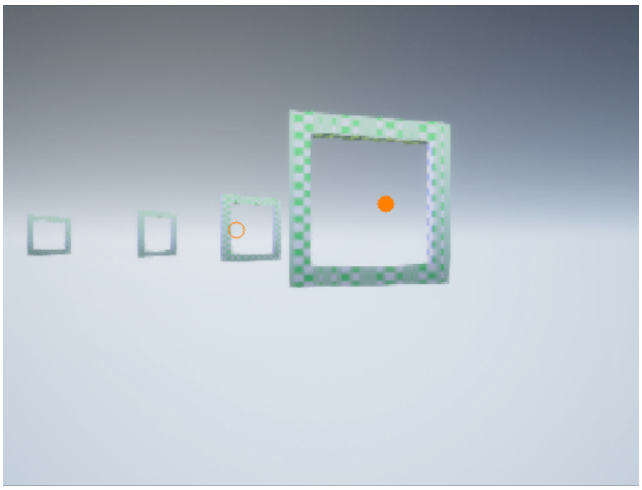


Figure 3.6: Visualization of the gate features overlaid with the image input.

The IMU provides the same data as in the waypoint guidance controller, which are the quadrotor roll (θ), pitch (ϕ), linear velocities ($\dot{x}, \dot{y}, \dot{z}$) and angular velocities ($\dot{\theta}, \dot{\phi}, \dot{\psi}$). Hereby, we can employ the same learning approach as the waypoint guidance controller developed in Section 3.1.2.

3.2.2 Image processing and gate detection

In order to perform visual navigation, a convolutional neural network was added to both value functions and control policy. The architecture for this task is the DroNet [51], an architecture based on the ResNet-8 with 3 residual blocks. This network receives, as input, an RGB-D[1] 320 pixels × 240 pixels image, obtained from the front camera of the quadrotor, providing, as output, the 6-feature vector of relevant gate information, used as part of the observation vector in the baseline model, connecting to the existing models as highlighted in figure 3.7.

Images as states occupy much more memory than simple numerical states. Thus, the replay buffer size is reduced, storing a maximum of 12000 samples.

Performing value and policy updates in CNNs is a very computationally demanding process, sometimes taking up to 15 minutes per training episode, depending on the hyperparameters used, with some models taking up to 10 days to complete the training process. This makes it exceedingly difficult to tune training hyperparameters and choosing an appropriate feature vector size.

To speed up the training process and to facilitate the design of some high-level hyperparameters, such as network architecture and the aforementioned feature vector size, the CNN part of the model was trained separately, in

[1] 3 channels for color and one for depth.

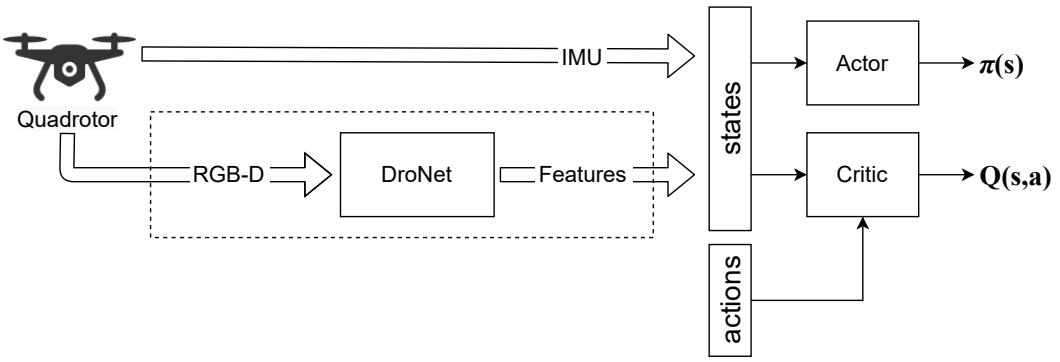


Figure 3.7: The DroNet network, highlighted by the dashed block, translates the input image into a feature vector, that is concatenated with the IMU readings from the vehicle.

a supervised manner, using ground truth data obtained from a simulation. This change of strategy significantly accelerates the model convergence to a satisfactory control policy.

3.2.2.1 Pre-trained model

Transfer learning is a common technique in machine learning, where the weights of a previously-trained network are transferred to a portion of another network for fine-tuning [57]. In this case, we pre-train a CNN that estimates the gate feature vector based on the RGB-D image input. This way, we are able to direct training for optimized results and easily evaluate the prediction accuracy. Ground truth data is generated from the same simulation used for training the RL model, with samples obtained from trajectories generated by the baseline control policy. The pre-trained portion of the overall model of figure 3.7 is highlighted by the dashed bounding box.

To increase the generalization capability of the network, an active learning-based mechanism was employed [58]. After each network test step, instead of selecting data based on uncertainty of prediction, the selection was based on the prediction accuracy, since data is already labeled automatically. With this, the 50% worst performing testing samples are added to the training set, randomly substituting old samples, and resample a new test set from the simulation. This refeeding loop is illustrated by the diagram in figure 3.8. This approach seeks to improve prediction accuracy in a limited number of training epochs by speeding up the learning process.

The feature vector has entries with largely differing magnitudes. For example, the gate center position in the camera frame is given as an angle, in radians, while the distance between the gate and the camera can reach up to 30 m. Thus, the loss function $\mathcal{L}(\theta)$ is weighted according to these differences

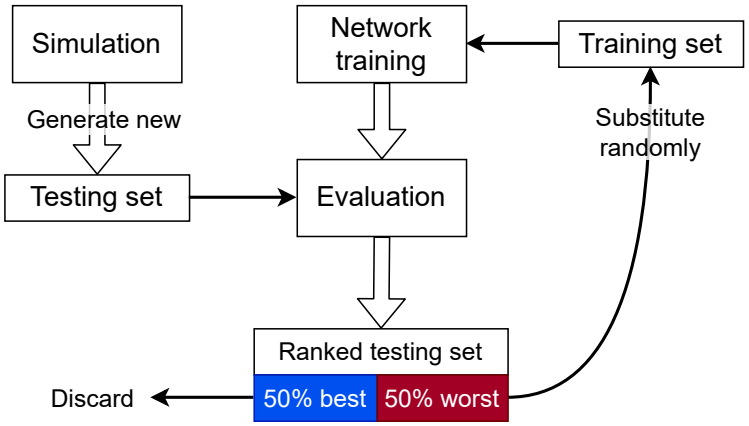


Figure 3.8: Training data resampling loop.

in scale and according to the importance of each feature for accurate detection (e.g. the center of the next gate is more relevant than other features), with a weight vector λ:

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{D}} (\lambda y_i - \lambda \hat{y}_i)^2, \tag{3-2}$$

where, out of a minibatch $\mathcal{D}$, y_i are the ground truth features and $\hat{y}_i$ are the predicted features. The weights correspond to the gate features, in order: horizontal position, vertical position, distance, rotation, subsequent gate horizontal position and subsequent gate vertical position, being $\lambda = [5, 5, 0.5, 1, 1, 1]$ if two or more gates are visible in the frame, and $\lambda = [5, 5, 0.5, 1, 0, 0]$ otherwise.

The pre-trained CNN is integrated with the existing baseline model in two ways: with trainable and with frozen weights. With trainable weights, the CNN weights, including its final dense layer, are frozen for the first 200 episodes, allowing the baseline weights to adapt to the new input, then, all trainable variables are unfrozen and fine-tuned. When integrated by the latter manner, the CNN weights are frozen for the entire training process, allowing only the baseline weights to be fine-tuned.

The prediction accuracy of a deep learning model with transferred weights depends on the similarity between the original environment where training takes place and the environment where the model performs prediction after receiving the transferred weights [57]. Since the CNN pre-training is done in the same racing courses as the control tasks, its prediction accuracy are not expected to reduce significantly, allowing further training to take place with is weights frozen. On the other hand, visual estimation introduces noise and disturbances to the observation vector. Thus, even if both control tasks, using real and estimated gate features, take place in the same racing courses, the introduced disturbances throw off the pre-trained baseline control model (actor and critic blocks in figure 3.7), making further training (fine-tuning) of these layers mandatory, both with and without trainable CNN weights.

4
Simulations

In this chapter we describe the details of implementation of the learning routine, learning environments, network architecture and hyperparameters. Additionally, we describe in detail the implementation of other algorithms, such as the PID gains optimization, quadrotor dynamics and the AirSim simulator.

The deep learning models developed are processed at the institution's remote GPU cluster, with 4× *Nvidia GTX 1080Ti* 12 GB VRAM GPUs, 2× *Intel Xeon E5-2669 v3* CPUs and 128 GB RAM. Some simpler models were processed, in parallel, in a personal computer, with an *Nvidia GTX 1660Ti* 6 GB VRAM GPU, an *AMD Ryzen 5 2600* CPU and 16 GB RAM.

4.1
Low-level control

We consider two low-level control tasks performed by the quadrotor: a waypoint guidance task and payload pickup course. The former is mainly used for training the controllers, while the latter is used for evaluation of these controllers, presenting a sudden mass change during payload pickup and drop.

The controllers are trained and tested in a simulated environment, consisting of the dynamic model of a quadrotor, without accounting for aerodynamic effects and sensor disturbances. This dynamic model takes as input the desired angular velocities of each rotor, while providing, as observations, the linear and angular velocities of the quadrotor in the local frame, and the linear and angular positions in the setpoint frame. Additionally, the dynamic model allows, at any time, the adjustment of five key quadrotor parameters: propeller diameter, propeller pitch, arm length, hub size and vehicle mass.

4.1.1
Training environment

The task for the quadrotor during training is to navigate towards the origin starting from a random point. For different training episodes, the environment parameters are randomized, following the bounds and constraints described by Table 4.1. The table includes, as well, the description of the fixed environment parameters for the benchmark learned controller. Arm length

is constrained with respect to the propeller diameter in order to properly accommodate it, while vehicle mass is constrained with respect to the same parameter in order to maintain a reasonable thrust-weight ratio, particularly around a mean rotor velocity of 5323 RPM, which was found to be the hover point of the quadrotor with fixed parameters described in Table 4.1.

Table 4.1: Upper and lower bounds of randomized quadrotor parameters used for waypoint controller development and PID gains optimization. Fixed parameters are used for training the single-environment benchmark controller.

Parameter	Lower bound	Upper bound	Fixed values
Propeller diameter (d) (in)	6.0	12.0	10.0
Propeller pitch (in)	4.5	4.5	4.5
Hub size (m)	0.05	0.15	0.10
Arm length (m)	$0.0167d + 0.05$	$0.0334d + 0.05$	0.3
Vehicle mass (kg)	$0.1d - 0.3$	$0.265d - 0.9$	1.2

The training hyperparameters used for the learned controllers are described by Table 4.2. Optimizer, replay buffer size, initial training steps and target update interval were chosen based on existing literature. The other hyperparameters were found by manual tuning, aiming to strike a balance between steady state guidance error and computational time, tested in ranges approximately one order of magnitude ($\times 10$) below and above the values described by the same table. In our implementation, different target entropies (h) did not significantly affect the learning performance, therefore this parameter is set at -4, as suggested by the original author of the algorithm for 4-dimensional action spaces [37].

Table 4.2: Hyperparameter and experiment configuration for the control tests.

Parameter	Value
Optimizer	Adam
Learning rate (Actor)	$3 \cdot 10^{-5}$
Learning rate (Critic)	$3 \cdot 10^{-4}$
Discount (γ)	0.99
Replay buffer size	10^7
Initial training steps	10^4
Number of samples per minibatch	256
Entropy target (h)	-4
Target smoothing coefficient (τ)	0.005
Target update interval	1
Gradient steps per episode	128

The ANNs are structured as described by figure 3.1, where the critic networks receive, as input, the state-action pair, for which they output its

predicted value. The actor network receives, as input, the state for which it predicts an appropriate action. As output, the network provides the deterministic action prediction and the natural logarithm of the appropriate exploration rate, with which a stochastic control policy can be computed. Further details of the training process are laid out in the SAC pseudocode in the Appendix A.

4.1.1.1 PID gains optimization

Here we consider the 18 different PID gains for a quadrotor controller. These gains are classified in categories with respect to which variable it controls (linear space: x, y, z, angular space: θ, ϕ, ψ) and to which term it refers to (proportional: **p**, integral: **i**, derivative: **d**). For example, the pitch angle proportional controller is denoted as $k_{\mathrm{p},\phi}$.

Prior to initiating the search algorithm, 100 evaluation environments are generated using randomized parameters within the bounds and constraints described in Table 4.1. This is done to maintain consistency between evaluations of different sets of PID gains.

For each search step, a random Gaussian noise with mean 0 and variance 1 is added to the gains vector. This vector is then clipped between a lower bound and an upper bound. The lower bounds are $k_{\mathrm{p},x}, k_{\mathrm{p},y}$:0.1, $k_{\mathrm{p},z}$:0.5, $k_{\mathrm{p},\theta}, k_{\mathrm{p},\phi}$:1.0, $k_{\mathrm{p},\psi}$:0.5 and 0 for all other gains. The upper bound is set to 10 for all gains. Lower bounds were initially chosen as 0 for all gains. However, the search would result in 0 for some of the gains, including some proportional gains and, sometimes, eliminating completely the control feedback loop. This happened especially with the yaw gains, due to its uncoupled nature not affecting overall stability. Because of this, the lower bounds were raised to adequate values for the indispensable proportional gains. Upper bounds were chosen arbitrarily as 10. Since gains were not clipped at 10 during search, it was left as is.

Finally, after a new set of gains is found, it is evaluated. If performance is improved, this set becomes the "current" set of gains, else it is discarded. The search is terminated if after 200 steps there is no improvement in performance.

Evaluation - position controller: The performance of a set of controller gains is evaluated on how many tests end with the quadrotor hovering at the waypoint. For consistency, this waypoint is always set to the origin, with a starting point set to a fixed distance of 1 m, with initial quadrotor attitude angles ranging from -0.064π to 0.064π radians (approximately 11.5°) for pitch and roll, which are reasonable ranges for non-acrobatic quadrotors, and from -0.3π to 0.3π radians (approximately 54°) for yaw, which is a reasonable range

to account any need for turning left or right. Starting points are randomized in advance and are the same for every iteration of the search algorithm.

Evaluation - velocity controller: Starting from the origin, the quadrotor follows a predefined trajectory, acting in all four action dimensions: roll, pitch, yaw and throttle. The performance is evaluated as the sum of the error signal along this trajectory.

Search dimensionality reduction: To further speed up the search time, we employ a number of optimizations to trim the search space from the 18 aforementioned PID gains. As the quadrotor is symmetric, the gains for pitch and roll angles are the same, as well as gains for the movement along the X and Y axes. Furthermore, as the linear approximation of the quadrotor is a double integrator [38], we can forego the integral gains, except for the vertical movement along the Z axis. With these optimizations, the search space is effectively reduced to only 9 parameters: $k_{\mathrm{p},xy}$, $k_{\mathrm{p},z}$, $k_{\mathrm{i},z}$, $k_{\mathrm{d},xy}$, $k_{\mathrm{d},z}$, $k_{\mathrm{p},\theta\phi}$, $k_{\mathrm{p},\psi}$, $k_{\mathrm{d},\theta\phi}$ and $k_{\mathrm{d},\psi}$.

4.1.1.2 Integrating dynamics with the environment

For the training phase, the quadrotor is subject to a 20 Hz control rate and a maximum episode time of 60 seconds (1200 time steps). For testing and experimentation, the same controller is subject to a 40 Hz control rate and maximum episode time of 60 seconds (2400 time steps). This is done to prevent a jittering behaviour we found during experiments, that occurs if the control rate is too low, likely caused by overfitting in some of the models trained.

The learning-based controller takes as input the pose error of the quadrotor with respect to the setpoint in terms of 3-dimesional errors (x, y, z) and yaw (ψ), as well as IMU data, such as roll, pitch, attitude rates and linear velocities. As output, the controller yields a normalized motor velocity vector, mapped to the range [1000 ; 9500] RPM. Due to the variability of vehicle parameters, the action is not always centered around the hover point. Consequently, the controller must learn this offset, an especially useful ability for handling the picking up and dropping of the payload.

The equations of motion in the local and inertial frames are, then, evaluated in the discrete-time environment using the ODE integrator included in *Python*'s *scipy*[1] library, employing the Real-valued Variable-coefficient Ordinary Differential Equation solver (VODE) and a method based on backward

[1] <`https://www.scipy.org/`> Accessed: 12 Mar. 2021

differentiation formulas (BDF). The system is integrated in up to 500 steps per control step, but sampled at the desired control rate for training or testing.

4.1.1.3 Starting conditions

For the origin guidance task, for both training and testing, the starting points are chosen in the same fashion as in the PID gains search, described in the paragraph "**Evaluation - position controller**", but sampled randomly at every episode, instead of being chosen from a set of predefined parameters.

4.1.1.4 Reward function

To complete this task, we define a reward function which is densely distributed through states, unless a state is terminal, in other words, if the quadrotor crosses the environment boundaries, in which then a reward of -80 is given. The reward function thus is described by:

$$r(s,a,s') = \begin{cases} -80 & \text{if the state is terminal;} \\ -1.0\sqrt{p_x^2 + p_y^2 + p_z^2} - 0.1\sqrt{\theta^2 + \phi^2} - 0.5\sqrt{\psi^2} & \text{otherwise,} \end{cases} \quad (4\text{-}1)$$

based on the position errors (p_x, p_y, p_z) and the attitude angles: roll (θ), pitch (ϕ) and yaw error (ψ). Defining the reward function in terms of the inverted quadratic error is customary in control problems [59]. However, we found in preliminary experiments that, for this environment, the root-squared error produces a more accurate control policy, when compared to the quadratic error. The reward term for roll and pitch angles aims to improve stability by minimizing the pitch and roll of the quadrotor during training. The weight factors for the positions and angles terms were found by manual tuning, where we did not want the angle terms to dominate the final reward too much.

4.1.2 Testing environment and experiment setup

The payload pickup course consists in navigating towards 2 waypoints in 3D space. This task is designed to take advantage of the parameter-agnostic capability of the controller by changing the vehicle size and mass mid-flight in the following fashion: upon arriving at the first waypoint, the quadrotor receives an additional mass corresponding to 30% of its current mass and an increased hub size of an extra 10 centimeters radius to simulate the extra size of the payload, which is a reasonable payload-mass ratio considering real

life payload-carrying multi-rotor (non-winged) drones. Then, it proceeds to the second waypoint, where this additional mass is removed, and, finally, flies back to the first waypoint. Each waypoint represents the origin of the setpoint frame, which can be moved within the inertial frame to the desired waypoint.

The controller is not guided by the payload pickup and drop waypoints directly, instead it is guided by a vector that points towards the desired waypoint, but with norm clipped to the maximum length of 1 m, as shown in figure 4.1. This is done to ensure the controller operates within the boundaries used for training or searching, especially because the output of neural networks outside the planned boundaries can be unpredictable.

The quadrotor is considered to have reached a waypoint when it is at most 0.15 m from its destination, which is set considering the steady state errors of the controllers, and is considered stable. The stability requirement is that the squared root sum of squared angular positions and rates errors is below a threshold which is defined by manual tuning, in this case, 0.15, where angular positions are measured in radians and angular rates are measured in rad/s.

For ease of visualization and evaluation, the experiments are performed in environments varying with respect to only propeller diameter and vehicle mass, with the arm length adjusted accordingly to a mean between the upper and lower bounds set for the training phase, with no randomization. This way, the controller performance can be evaluated in a 2-dimensional visualization. The ranges and constraints of these three parameters are described in Table 4.3.

For the payload pickup task, the vehicle mass upper bound is reduced to approximately 76% of the value used for the training task, in order to accommodate the extra 30% payload weight, without burdening the vehicle to a point it cannot hover. When the extra mass is added, the total mass equals the upper bound set for the training task.

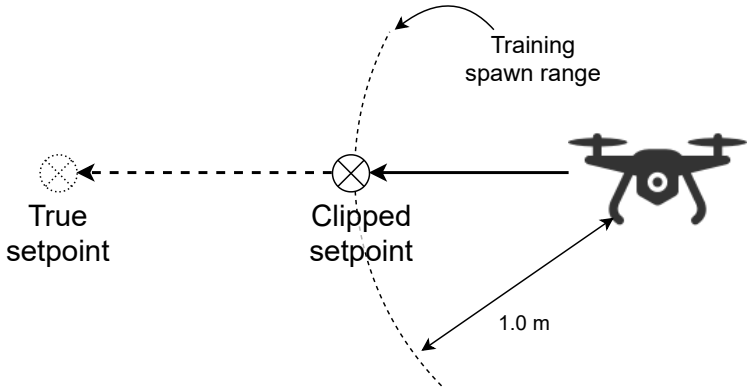


Figure 4.1: Diagram showing how a setpoint is clipped if it is >1 m away from the quadrotor.

Table 4.3: Range of quadrotor parameters used in experiments.

Parameter	Lower bound	Upper bound (waypoint tracking)	Upper bound (payload pickup)
Propeller diameter (d) (in)	6.0	12.0	12.0
Arm length (m)	$0.025d + 0.05$	$0.025d + 0.05$	$0.025d + 0.05$
Vehicle mass (kg)	$0.1d - 0.3$	$0.265d - 0.9$	$0.2d - 0.66$

The waypoints for the payload pickup task are the same for all experiments for consistency. The pickup point is $[-0.8, -0.3, -0.5]m$, in the lower half of the environment, whereas the drop point is $[0.8, 0.3, 0.5]m$, in the upper half, requiring the quadrotor to climb while carrying the extra weight, posing an interesting challenge for the designed controllers.

The quadrotor spawns close to the pickup point, anywhere within a 0.5 m cube centered on it and with the same initial attitude angles used in training, which range from -0.064π to 0.064π radians for pitch and roll and from -0.3π to 0.3π radians for yaw.

For each controller, enough tests are performed until it reaches 100 successful samples. A payload pickup test is considered successful if the quadrotor manages to reach all the waypoints defined in the trajectory within a time limit, set to 500 time steps (12.5 seconds at 40 Hz) for each waypoint.

4.2 Visual navigation: AirSim

For the visual navigation controller, we use an *Unreal Engine*-based simulator, AirSim[2] [60]. This simulator generates a 3D world where the drone navigates and allows spawning a camera, which creates a rendering of the environment for visual processing tasks. The simulator also models the drone flight and collision physics, for accurate real-world representation. For the racing courses used, the simulator does not allow the exchange of quadrotor models. Thus, in this part, the quadrotor parameters are left as default and do not change.

The main objective of the visual navigation controller is to complete a racing course by crossing square gates laid along the race path, in sequence, without colliding with them. In order to teach the agent to cross a gate, we define a sub-task that consists in navigating towards the center position of the gate, similarly to how the low-level controller developed in this work navigates towards a waypoint in the environment.

[2] <https://microsoft.github.io/AirSim/> Accessed: 12 Mar. 2021

In this sub-task, the controller has no access to externally available information on the racing course, such as in which section of the course the quadrotor is, how many gates have been traversed, etc. The controller also does not know the shape and size of each gate beforehand, being required to learn these by itself as it trains. Aside from navigating towards the gate 3D position, the quadrotor is taught to cross it facing the next gate to be crossed.

The training hyperparameters used for this environment are the same used for the low-level control environment, described in Table 4.2, and the same network architecture, except the CNN prior to the feature vector input, is used for the actor and critic networks. The only exception is the maximum buffer size, which is reduced to $1.2 \cdot 10^4$ if it is storing images, since they demand a larger memory space. Carrying over these characteristics from the previous training algorithms yields satisfactory results, especially considering that manual tuning is unmanageable for this environment, as training takes a notoriously long time to complete.

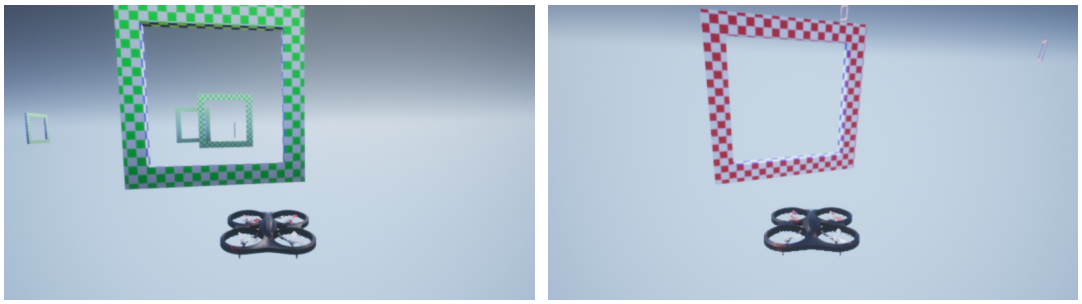
The actions are sent to the simulation via a *moveByAngleRatesThrottleAsync* method, the lowest-level command available for this simulator, which sends angular velocities setpoints in rad/s, as well as a throttle command between 0 and 1. These angular velocities are limited by the simulator to the range $[-1\,;\,1]$, the same output range used by the control policy network, so there is no need to adjust these commands. On the other hand, for throttle commands, the control policy output is normalized to the range $[0.2\,;\,1]$, since its midpoint, 0.6, is the hover point of the quadrotor in the simulation, so that a zero throttle output by the control policy corresponds to a hover behavior.

4.2.1 Racing courses

The racing courses are based on the *NeurIPS2019 Drone Racing*[3] competition, more specifically, the *Soccer_Field_Training* tracks. In these tracks, the racing path is marked with large square gates (figure 4.2). The colored checkered sides indicate the direction of crossing.

The *Soccer_Field_Easy* course is simply circular, with gates vertically aligned and equally spaced. The course map is shown in figure 4.3. On the other hand, the *Soccer_Field_Medium* course is more complex. Gates are both sparsely and tightly spaced, with tight turns, also requiring climbing and descending. The course map is shown in figure 4.4.

[3] <https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/> Accessed: 12 Mar. 2021



(a) (b)

Figure 4.2: Gate texture indicating front/back side and correct crossing direction. (a) Front side. (b) Back side.

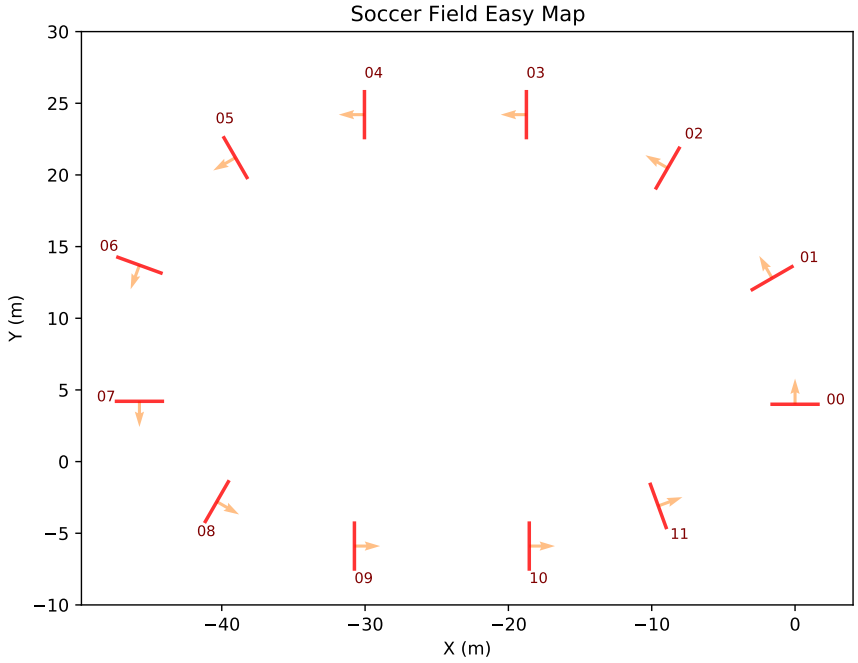


Figure 4.3: "Easy" race course top-down view. Gates are numbered according to the order of crossing and crossing direction is indicated.

The medium course presents a few choke points, that pose a challenge for the navigation controllers. When navigating through the sparse sections, the depth camera will not be able to pick up the gates reliably. On the other hand, in the crowded sections, the camera will not be able to capture the next gate entirely, especially during ascent/descent or tight curves. Figure 4.5 shows a crowded descent section and figure 4.6 shows an ascent section, both of which presenting a challenge to the developed navigation controllers. The visual estimator is unable to tackle the challenges present in these sections reliably without an external aiding mechanism. Thus, the baseline RL controller needs to learn how to manage the estimation disturbances during the fine-tuning phase.

Training takes place in both racing courses. For each training or testing

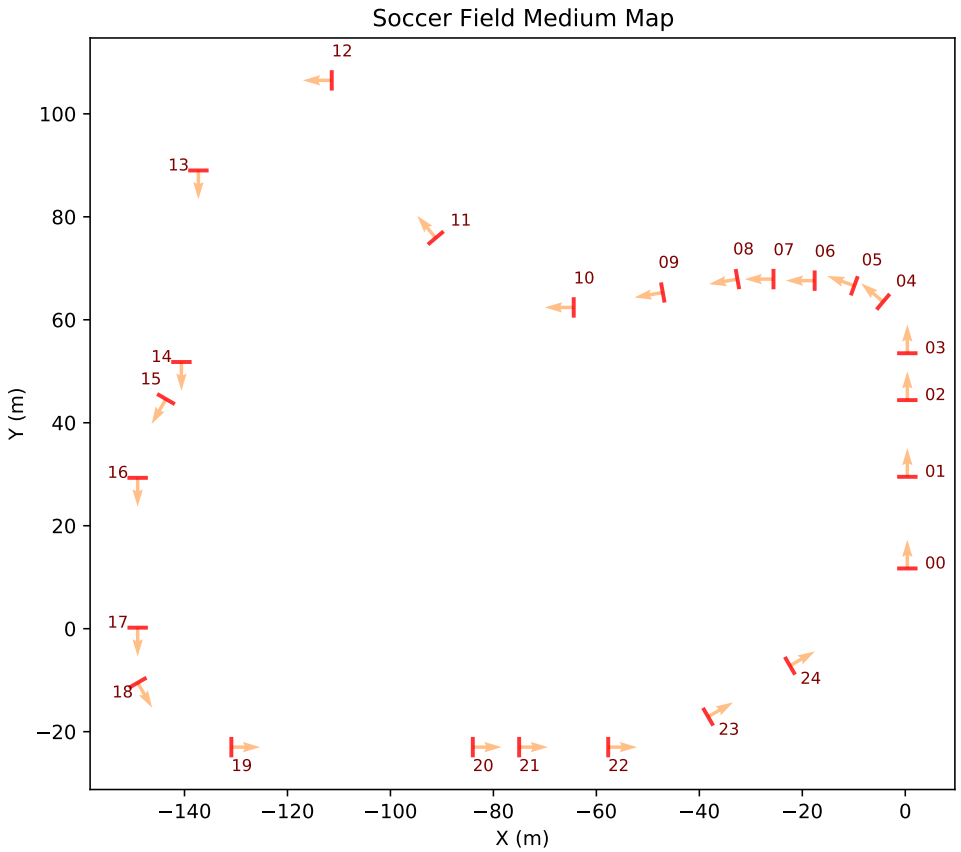


Figure 4.4: "Medium" race course top-down view. Gates are numbered according to the order of crossing and crossing direction is indicated.

episode, one of the courses is chosen randomly for spawning the quadrotor. After loading and prior to training or testing, the simulation environment is cleared of any objects except the gates, the quadrotor and the *skybox*.

4.2.2 Training environment

The training environment is an interface between the simulation and the training algorithm. This interface processes the state of the simulation in order to issue observations and rewards to the learning algorithm, as well as to serve as mission control (i.e. detect gate crossings and early termination conditions).

While the simulation operates with real-time control actions, the environment operates through a *lockstep* logic, in a sample rate of 10 Hz. In other words, the simulation remains paused until a control action is issued. Then, it runs for 100 ms until it is paused again and the new simulation state is observed along the reward signal. In this way, we guarantee the action is issued at the same time, and same state s, the control policy is evaluated. Additionally, any further computations can be done while the simulation is paused, avoiding disruption of this cycle.

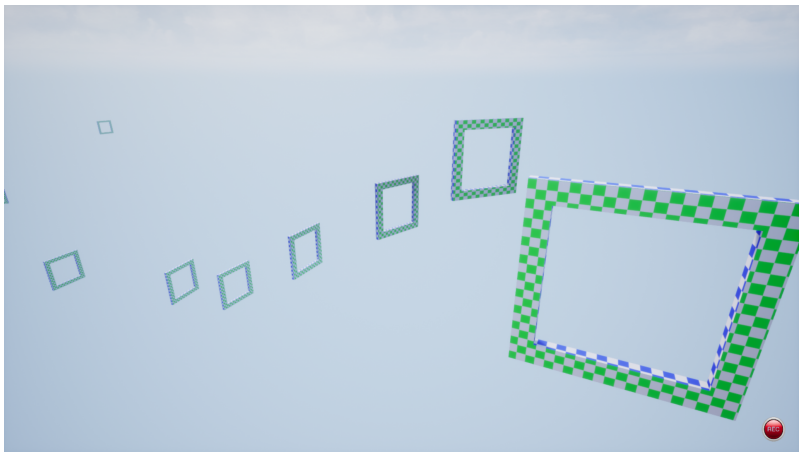


Figure 4.5: Steep descent section of the medium course.

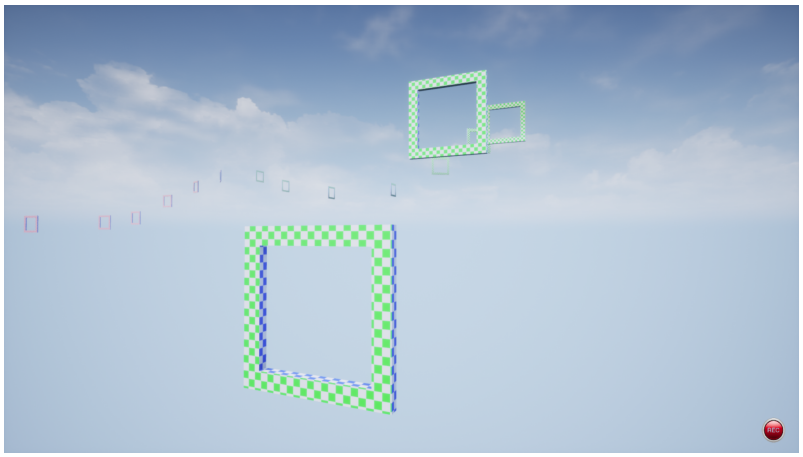


Figure 4.6: Steep ascent section of the medium course.

The baseline layers, denoted by the Actor and Critic blocks in the diagram in figure 3.7, are ANNs structured as described by figure 3.1, where the critic networks receive, as input, the state-action pair, for which they output its predicted value. The actor network receives, as input, the state for which it predicts an appropriate action. As output, the network provides the deterministic action prediction and the natural logarithm of the appropriate exploration rate, with which a stochastic control policy can be computed. For both networks, the state vector contains true gate features during the "baseline" training phase of this controller, while it contains estimated gate features output by the DroNet during the "visual" training phase. During the latter phase, the DroNet receives, as input, the FPV camera feed, and its output is connected directly to the first 6 entries of the state vector.

4.2.2.1 Starting conditions

To generalize detection for the different gates, especially during early steps of training, the quadrotor is spawned in a random position along the course. This position is chosen as follows:

1. A random gate from the course (except the last) is chosen to be the first the quadrotor attempts to cross;
2. Find the previous gate, or a fixed point in case the quadrotor starts at the first gate, and find the line that connects this gate to the next;
3. Begin at any point along this line between the previous gate and middle point; and
4. If the length of the line is larger than 10 m, shift the quadrotor position 2 m towards the next gate and add a random position noise along axes X and Y of 1.5 m.

4.2.2.2 Reward function

The reward is densely distributed through states and corresponds to the absolute distance between the state and the center of the gate, or:

$$r(s, a, s') = -0.1d = -0.1\sqrt{p_x^2 + p_y^2 + p_z^2}, \tag{4-2}$$

which is, as seen by the guidance controller, the position error between the quadrotor and the setpoint.

In certain states, the distance-based reward is substituted by a more substantial value that indicates the desirability of that state. In case a gate is crossed, the agent receives, instead, a reward of 100, the only positive reward possible for this environment. In case a state is terminal, the agent receives, instead, a negative reward based on the cause of termination.

- If the quadrotor collides with a gate, the episode ends with a terminal reward of -5;
- If the next gate to be crossed is not visible, the episode ends with a terminal reward of -20, even if other gates are still visible (otherwise, the agent learns to follow the course while ignoring the gates).

The completion of the course is not considered a terminal state, even if it is a termination condition, as we wish the learning algorithm to understand the course as infinite, at least theoretically.

There are other termination conditions that do not entail extra rewards and are not considered terminal states. These are included to limit the episode length, when termination by other means seem unlikely. One condition is a maximum distance towards the gate to be crossed, if the quadrotor moves past 50 m from the gate. Another condition is a maximum amount of time for a gate to be crossed (timeout) of 600 time steps, or 60 seconds at 10 Hz.

4.2.2.3 Camera properties

The visual observations are provided by the simulation of an RGB-D camera, that provides a 3-channel color image (red, green and blue) with an extra channel for depth measurements. The image resolution is 320×240 pixels and the field of view angle is 90°.

The color images are requested from the simulator as uncompressed matrices with each pixel encoded as an unsigned 8-bit integer (values ranging from 0 to 255) and this image is normalized to values between -1 and 1 prior to being fed to the actor-critic networks.

The depth images are requested from the simulator as an uncompressed depth perspective, in other words, each pixel represents the length, in meters, of a projection ray between the camera and the object in that pixel. Measurements provided by the simulation are not capped, thus, as we wish to emulate a real depth camera, we clip the measurements to 20 m. Then, this image is normalized to values between -1 and 1, prior to being fed to the actor-critic networks[4].

Further, the images are stabilized with the help of a gimbal, which is a device that mechanically stabilizes the camera of a quadrotor, a common feature of commercial photography and racing UAVs[5]. This device maintains the camera pitch and roll at 0° with respect to the inertial coordinate frame.

4.2.3 CNN pre-training

The convolutional section of the model is trained separately in the same environment as the baseline model. In this training process, the dataset is generated on-the-fly by following the baseline control policy in a new simulation instance. Once the initial dataset, consisting of a replay buffer, is filled, a supervised training process begins. During this stage, the final dense layer of

[4]In fact, the normalized image is multiplied by -1, as the normalization function is the same used for visualization. Mathematically, there should be no difference in performance.

[5]All but one of the top-selling quadrotors in <`https://www.techradar.com/news/best-drones`> (Accessed: 15 Apr. 2021) feature a gimballed camera.

the DroNet architecture has a dropout rate of 0.2. This dropout is removed (factor set to 0) during testing and once the CNN is integrated with the existing actor-critic model, even when fine-tuned.

In this part of training, the CNN receives, as input, the FPV camera feed. As output, it provides a prediction of the features of the nearest gate. The network is trained by comparing these predicted features to the true features.

For the model testing, the samples are generated again from the simulation. In order to improve the model generalization, we select the testing samples which results in low accuracy predictions and add those to the replay buffer, substituting older samples at random.

Algorithm 2: CNN pre-training

```
Fill a replay buffer R with samples
for epoch = 1,N_epochs do
    Train the network for every minibatch from R
    if evaluation step then
        Generate N_t new samples from the simulation
        Evaluate the network on these new samples
        ↪ Store the prediction loss for each individual sample
        Find the median ỹ of these losses
        for each test sample do
            Store in R if loss > ỹ, replacing a random sample
        end
    end
end
```

The hyperparameters used for training this network are described in Table 4.4.

Table 4.4: Hyperparameters for the supervisioned training of the DroNet.

Parameter	Value
Optimizer	Adam
Learning rate	$3 \cdot 10^{-4}$
Replay buffer (dataset) size	12800
Batch size	128
Test interval (epochs)	100
Test samples	2000

4.2.3.1 Grad-CAM

Grad-CAM [52] is a special class activation map visualization tool used to verify how CNNs evaluate inputs and calculate their outputs. This tool

generates a heatmap highlighting the sections of an input mage with the most relevance for the model, as shown in figure 4.7. This tool lets one check if the model correctly identifies the next gate to be traversed and the gate that follows.

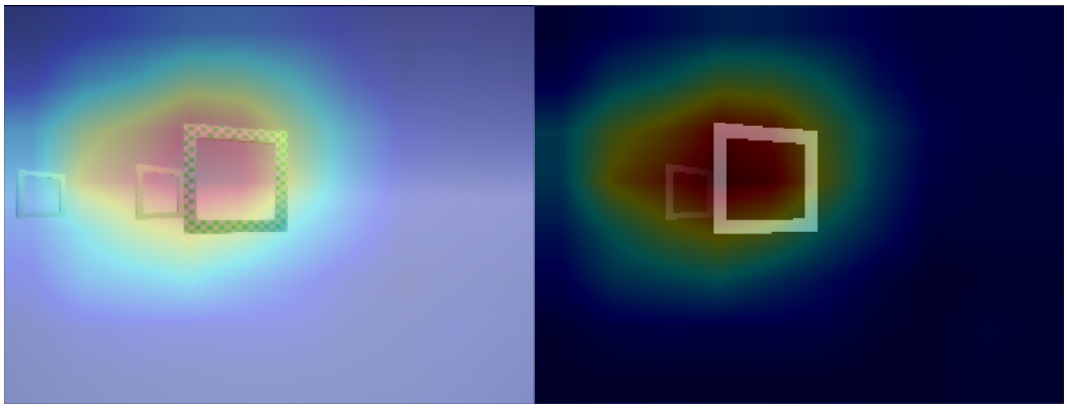


Figure 4.7: Grad-CAM overlay over RGB (left) and depth (right) camera. Red overlay indicates higher relevance regions, while blue overlay indicates less relevant regions.

5
Results

In this chapter, the results of the proposed experiments are laid out and commented in detail. Similarly to Chapters 3 and 4, this chapter is divided in two sections, one for each part of the dual objective, namely "low-level control" and "visual navigation".

For the low-level control section, experiments are carried out using the dynamic model of the quadrotor as the virtual environment. The tasks consist of navigating to setpoints in space while introducing variations in the vehicle parameters before and during experiments.

And, for the visual navigation section, experiments are carried out in the 3D simulator *AirSim*, containing one of the two predefined gate arrangements, namely *Soccer_Field_Easy* and *Soccer_Field_Medium*. The task of the quadrotor in this environment is to attempt to cross the gates in order. In this section, two controllers are evaluated: a "baseline" controller which takes, as part of its input, real position data of the gates, and a "visual" controller which estimates the gates' positions from an FPV camera feed.

5.1
Low-level control results

In this section, we lay out the results of development and testing of the proposed low-level controllers. The key experiments found in this section are:

- The resulting PID gains for the PID controllers;
- Learning performance of the RL controllers; and
- Performance in carrying out the proposed tasks, which are:
  - Simple waypoint guidance experiments; and
  - The payload pickup and drop course.

For the waypoint guidance experiments, starting conditions are the same used during training, described in Section 4.1.1. Success and failure conditions are also the same: in a successful test, the quadrotor is capable of remaining within the environment boundaries for 12.5 seconds. If these boundaries are crossed before this time, the experiment is considered a failure. Note that

this does not take into account the quadrotor stability, errors and oscillatory responses.

For the payload pickup experiments, starting conditions are also the same used during training, described in Section 4.1.1. Successful experiments consist of the quadrotor reaching the pickup point again after dropping the payload in the designed drop point. If the quadrotor crosses the environment boundaries, or is unable to reach any waypoint within 12.5 seconds from spawning or from the previous waypoint, the experiment is considered a failure. Further details on the criterions used for convergence to a waypoint can be found under Section 4.1.2.

One of the metrics used to evaluate the performance of each controller in carrying out the proposed tasks is its expectation ($\mathbb{E}[x]$) to do so, is estimated for different combinations of the quadrotors size (propeller diameter and arm length) and mass. The experiments for each controller are distributed in a 2D parameter space, composed by the propeller diameter and the vehicle mass, according to the parameter combination used for each test. Then, the expectation map is computed using a moving window filter of size [1×0.417] (propeller size×mass) in a 100×100 grid in the parameter space. For each point in the map, the expectation of success is estimated by the amount of successful tests under a window around this point, with respect to the total tests under this window, or $n_{\text{success}}/n_{\text{total}}$. Points outside the limits established by Table 4.3 and points in which there are no tests under its respective window are left blank.

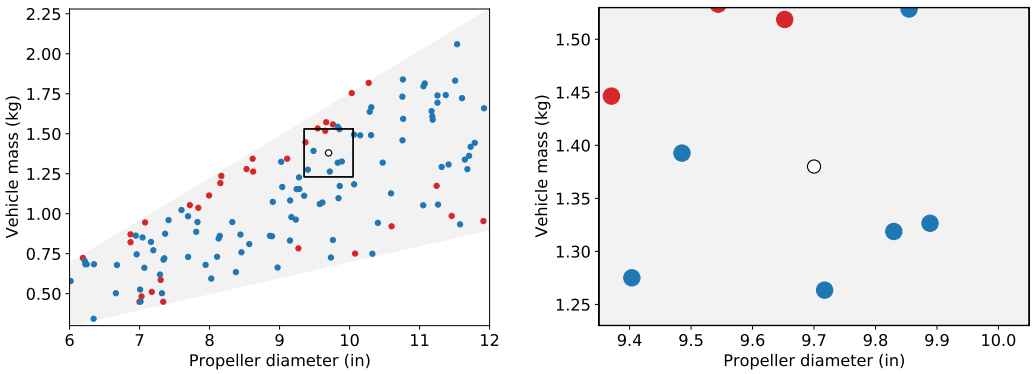
Figure 5.1 illustrates how this filter works in a dummy experiment distribution. In the figure, blue dots represent successful experiments, while red dots represent unsuccessful experiments. Under a window centered in parameters (m, p_d), the expectation of success for this parameter combination is computed by:

$$\mathbb{E}(m, p_d) = \frac{\text{successful tests}}{\text{total tests}} = 6/8 = 0.75, \tag{5-1}$$

and can be visually evaluated in figure 5.1b.

5.1.1 PID tuning and performance

The resulting gains for the controllers after the search is concluded are described by Table 5.1, for the position PID gains, and Table 5.2, for the velocity PID gains, the latter composing part of the cascade controller. These gains are obtained from an initial gain vector of [0.1, 3.0, 3.9, 0.0, 0.0, 10., 1.0, 0.71, 0.0] for, respectively, the gains $k_{\text{p},xy}$, $k_{\text{p},z}$, $k_{\text{i},z}$, $k_{\text{d},xy}$, $k_{\text{d},z}$, $k_{\text{p},\theta\phi}$, $k_{\text{p},\psi}$, $k_{\text{d},\theta\phi}$,



(a) (b)

Figure 5.1: Visual representation of the moving window filter for a dummy experiment success distribution. The winow size in this representation does not correspond to the actual window size used.

and $k_{\mathrm{d},\psi}$. The initial gains are obtained from manual tuning, set to values that allow the controller to complete the waypoint guidance task successfully at least once, in order to obtain a non-minimum final reward and to a "gradient" to be perceivable (i.e. a search step that results in clear improvement in evaluation). Otherwise, search steps need to rely on chance to find a set of gains that result in a different, non-minimum evaluation.

Table 5.1: PID gains for the quadrotor position control.

	P	I	D
Longitudinal position (X,Y)	0.125	0.0	0.247
Vertical position (Z)	3.707	0.877	0.841
Roll (θ), Pitch (ϕ)	6.107	0.0	2.828
Yaw (ψ)	0.521	0.0	1.683

Figure 5.2 shows the resulting performance of these gains for the pose PID in the waypoint guidance task. This controller obtained 100 successful tests out of 131. Figure 5.2b shows the position over time of all successful experiments broken down in each direction of movement, X, Y and Z. Note that the horizontal and vertical movements behave differently. Figure 5.2c shows the attitude over time of all successful experiments broken down in each direction of rotation, θ, ϕ and ψ.

The position axes have been symmetrically log-scaled[1] so that the quadrotor behaviour around the origin is highlighted. The red dashed line represents the extent of the linear scaling area, which is 0.05 m around 0,

[1]The plot is log-scaled (and $-$ log-scaled) for most of the plot area, except a small region around the axis origin, where the plot is linearly scaled. This scale is appropriate for data points with negative values in the scaled axis.

Table 5.2: PID gains for the quadrotor velocity control.

	P	I	D
Longitudinal velocity (X,Y)	0.379	0.0	0.0
Vertical velocity (Z)	2.872	4.237	0.0
Roll (θ), Pitch (ϕ)	9.493	0.0	1.148
Yaw (ψ)	0.5	0.0	0.0

where 95% of the initial distance has been covered. This visualization, though unrefined, contains abundant information for the behaviour of the controller.

A preliminary highlight in the success expectation of the pose PID controller is the red region in figure 5.2a. A larger concentration of failed experiments with high-mass vehicles show that this controller may have problems with saturating commands sent to the vehicle. Further inquiries into this aspect are contained in Section 5.1.3, in direct comparison to the results obtained from the main learning-based controllers.

5.1.2 Single environment training and performance

Figure 5.3 shows the performance of 10 different training runs of 3000 episodes using fixed environment parameters.

Despite the learning converging quickly to a satisfactory control policy with low steady state errors, training the waypoint guidance controller with fixed environment parameters put this controller at risk of overfitting to this particular environment. Without the randomization of training environment parameters, we can observe in figure 5.4 the performance metrics of the controller which attained the best success rate out of 10 training runs with the mentioned training conditions. In this case, the controller obtained 100 successful tests out of 109.

Figure 5.4a shows how this controller overfits to a certain range of quadrotor parameters, displaying poor performance for heavier vehicles. In figure 5.4b, which displays the position of the quadrotor during the waypoint guidance, we can observe how the position error, especially the height (Z) error, increases with respect to the error observed during training. This happens because the propeller diameter to mass ratio differs between environments, so does the base motor input required for the quadrotor to hover. This aspect is managed by the integral component of the vertical position PID controller (see Table 5.1), which, despite being slower than the horizontal position guidance, it clearly converges to the waypoint as time goes to infinity (see figure 5.2b). However, the learned benchmark controller does not converge. This is expected,

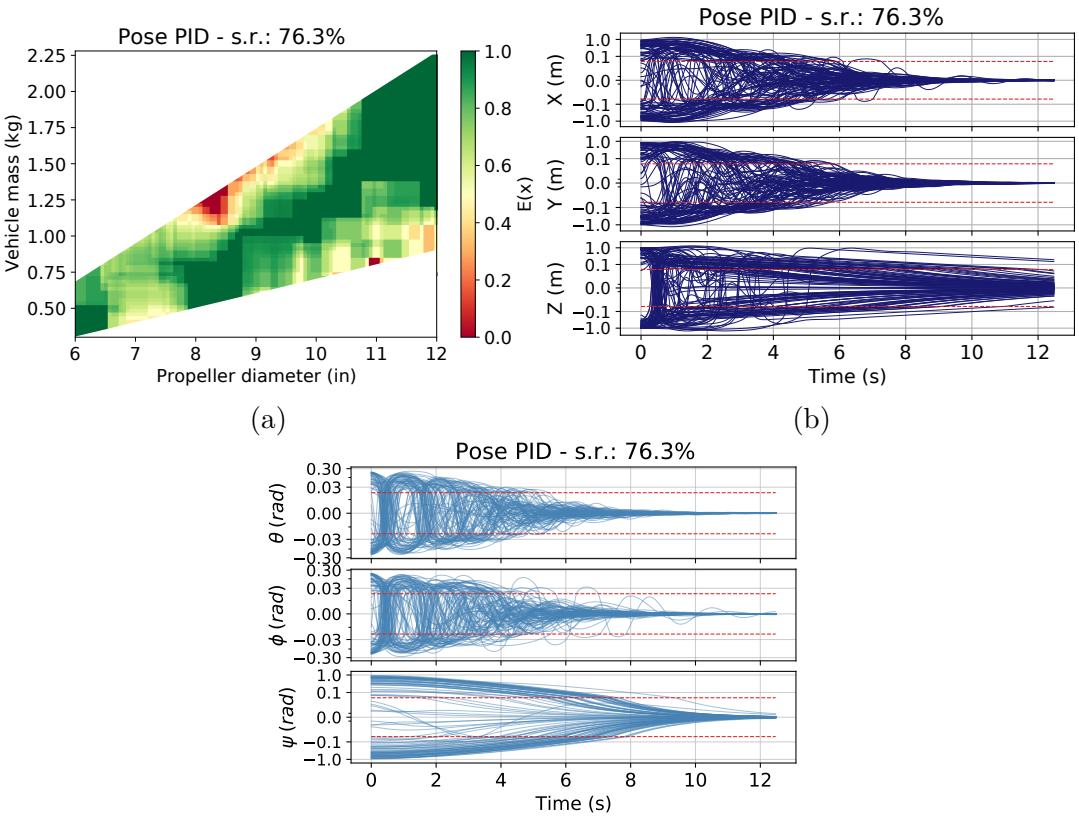


Figure 5.2: Statistics for the pose PID controller. (a) Expected success rate (s.r.) in the waypoint guidance task. (b) Position of the quadrotor over time. (c) Attitude of the quadrotor over time.

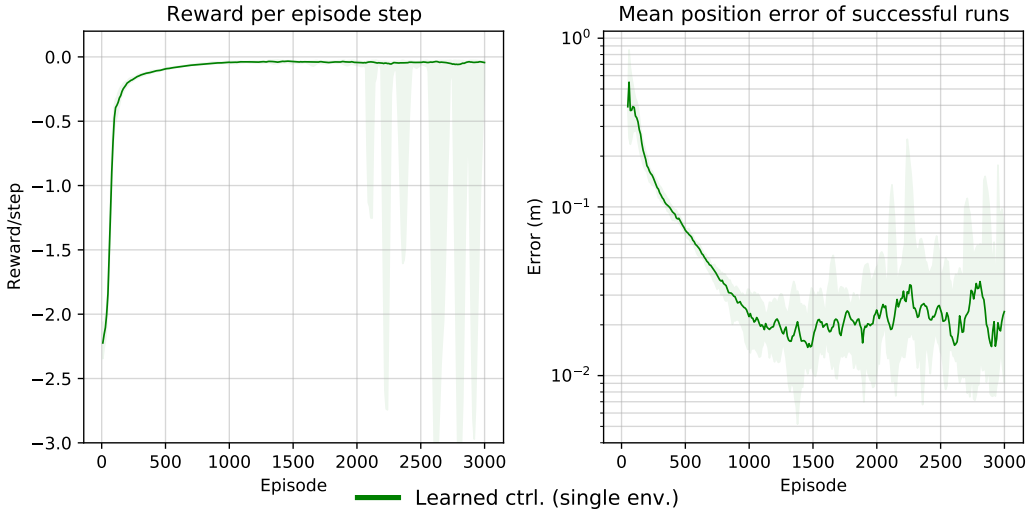


Figure 5.3: Training metrics for the fixed environment controller. The lines refer to the median metric from the 10 samples, while the shaded region denotes the minimum and maximum metrics from samples. Each data point represents the moving average of the 100 previous tests (50 previous episodes).

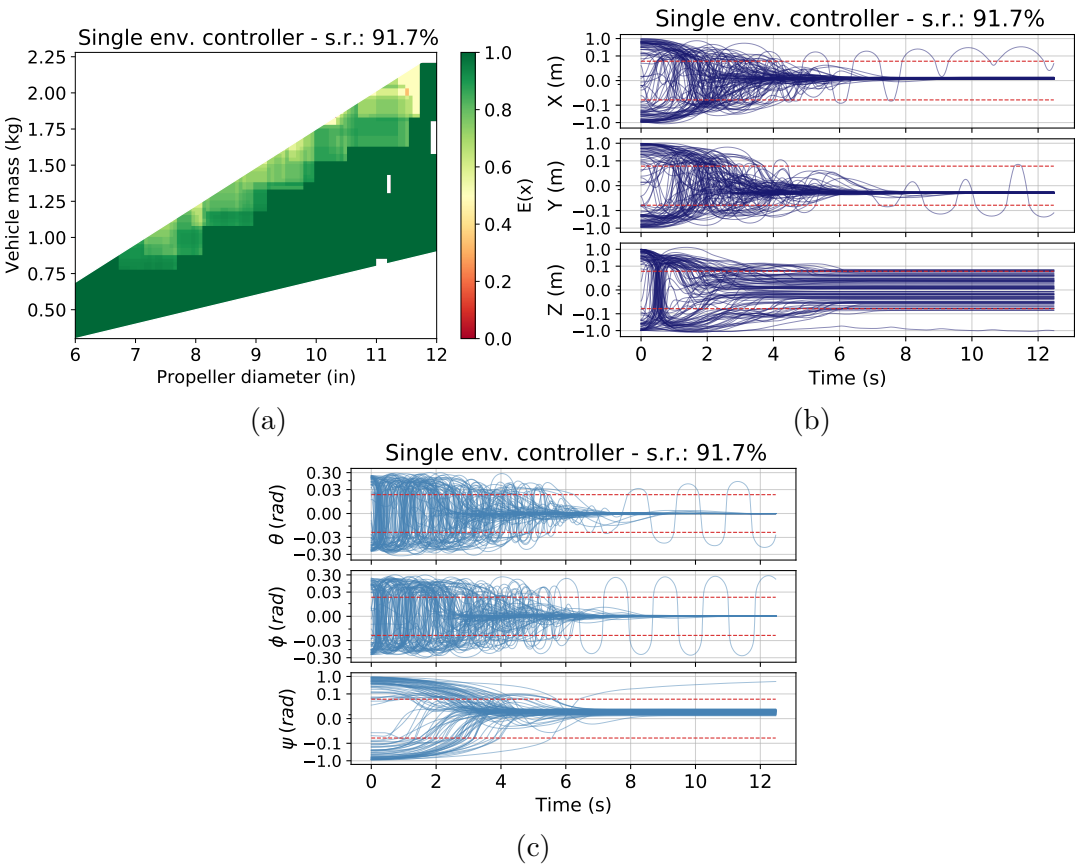


Figure 5.4: Statistics for the controller trained in a single set of environment parameters. (a) Expected success rate (s.r.) in the waypoint guidance task. (b) Position of the quadrotor over time. (c) Attitude of the quadrotor over time.

as the controller has overfitted to a limited set of environment parameters, thus being unable to generalize commands for quadrotors with different base actuations.

Notice that, even if these results indicate that this controller may have overfitted to a single set of environment parameters, it still attains superior performance with respect to the pose PID controller. However, as the development of the former is not the aim of this work, it is left out of further experiments.

5.1.3 Performance evaluation of the waypoint guidance task

Figure 5.5 shows the training performance of 10 different training runs of 3000 episodes for each of the main learned controllers, the fully learned and cascade controllers, compared to the training performance of the controller learned on a fixed environment.

In figure 5.5, while the learning performance of the fully learned controller (blue line) and the cascade controller (red line) are similar, the learned

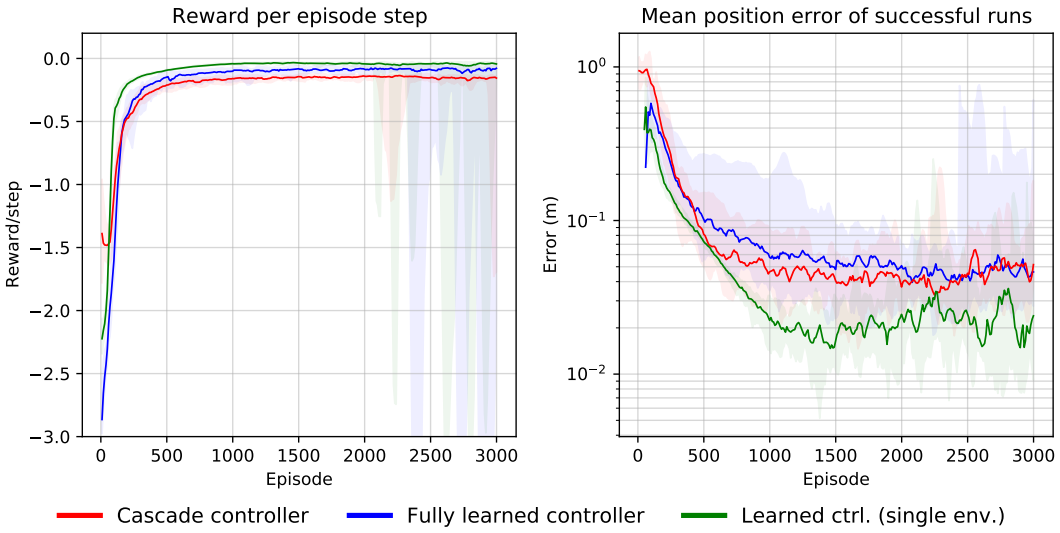


Figure 5.5: Training metrics for the learned controllers. The lines refer to the median metric from the 10 samples, while the shaded region denotes the minimum and maximum metrics from samples. Each data point represents the moving average of the 100 previous tests (50 previous episodes).

controller trained on a single environment parameter (green line) manages to achieve higher returns after convergence. However, as previously mentioned, these higher returns occur only within a limited set of quadrotor parameters, while other curves account for different parameters.

For the simulations performed in this work, the best controller of each kind is chosen. The chosen controller must have, among the other trained instances, the lowest steady-state error in navigating towards the origin, while maintaining a success rate of 100% over 100 tests specifically in the payload pickup task, which is the final goal of these controllers. Both the fully learned controller and the cascade controller take about the same amount of training episodes to converge to a locally optimal policy, both displaying very proximate position errors.

Figure 5.6 shows the expectation of success in completing the waypoint guidance task for the learned controllers and the baseline pose PID controller.

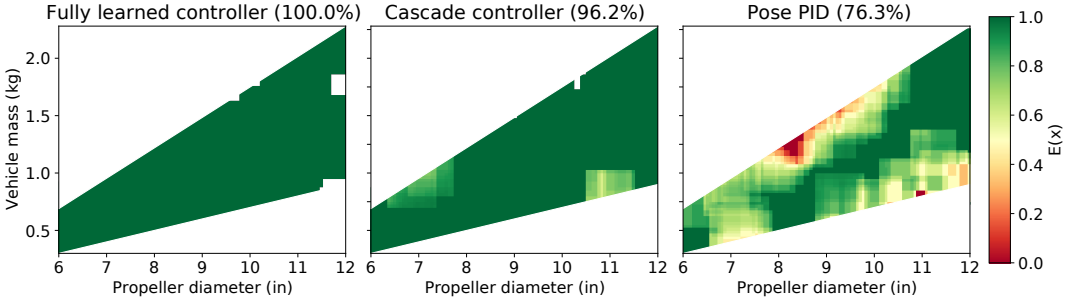


Figure 5.6: Expected success rate (s.r.) in the waypoint guidance task for each controller by combination of quadrotor parameters.

The experiments using the fully learned controller obtained full success out of 100 tests, while the experiments using the cascade controller obtained 100 successful tests out of 104 and the ones using the pose PID controller obtained 100 out of 131.

RL controllers trained on multiple environment parameters perform better in the waypoint guidance task with higher robustness to a variety of environment parameters, as observable in figure 5.6, than the controller trained on a single environment, in figure 5.4a.

Additionally, in figure 5.6, we observe another expected result: the pose PID controller, despite having gains optimized for the displayed range of quadrotor parameters, is not as robust as the learned, non-linear controllers, achieving lower success rate than an RL controller trained on a single set of parameters.

As a further inquiry into this task, for all 100 successful tests for each controller, over time, figure 5.7 shows the position of these tests, figure 5.8 shows the attitude and figure 5.9 shows motor commands sent.

The first highlight would be the settling time of the quadrotor. Both the fully learned controller and the pose PID have a smooth approach to the

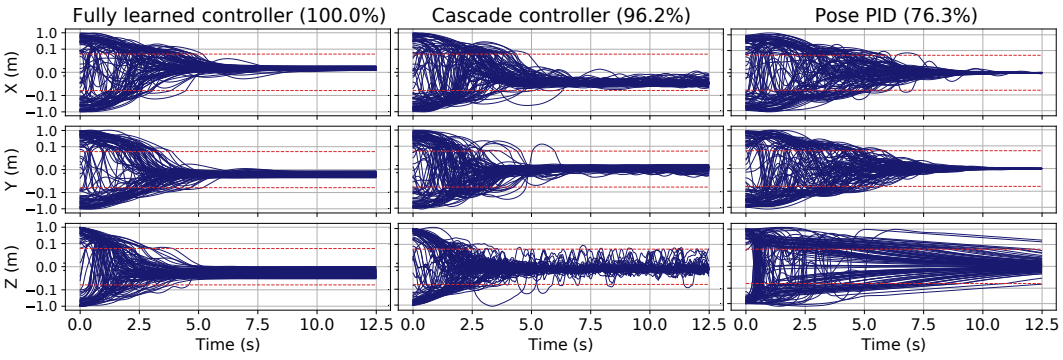


Figure 5.7: Position of the quadrotor over time for different controllers in the waypoint guidance task.

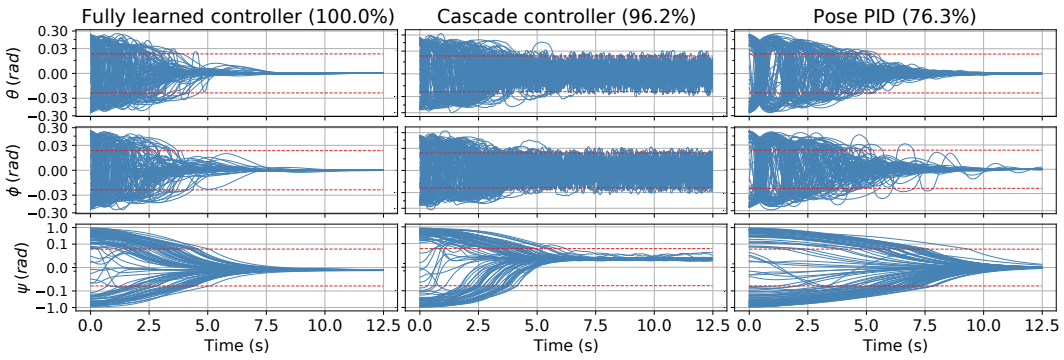


Figure 5.8: Attitude of the quadrotor over time for different controllers in the waypoint guidance task.

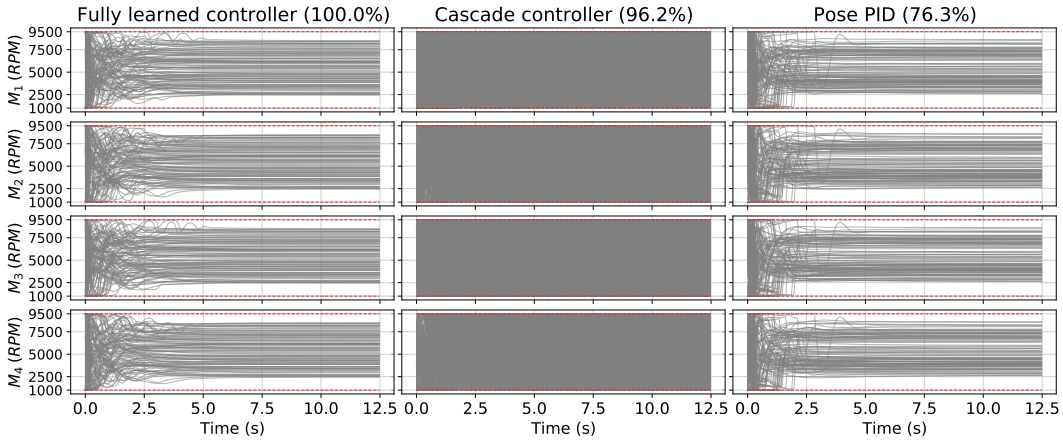


Figure 5.9: Motor commands of the quadrotor over time for different controllers in the waypoint guidance task. The dashed red lines here represent the minimum and maximum commands accepted by the motors.

origin, the former converging, in average, in 3.390 seconds and the latter in 5.182 seconds, taking slightly longer. The vehicle is considered to converge to any point when its velocity does not exceed 0.05 m/s for the remainder of the experiment. This does not happen to 44 out of the 100 successful tests for the cascade controller, which visibly oscillates around the origin. Note that the convergence requirement is different from the success requirement, which is less strict than the former. In view of the wide actuation swings seen in figure 5.6, we can assume that these oscillations result from the learned controller trying to "subdue" the velocity PID controller in environments where its performance is poor. This may happen because the velocity setpoint of the internal PID controller changes rapidly, causing an increase in amplitude of these changes.

The second point to consider would be the steady-state error. This metric, combined with the settling time, is numerically evaluated in Table 5.3. Notice that, for the cascade controller results, we consider the settling time of tests that did not actually settle to be the maximum test time, 12.5 seconds, resulting in a higher mean settling time.

Table 5.3: Mean and standard deviation metrics for the waypoint guidance task for 100 successful samples of each controller, including single environment (non-randomized) training.

Controller	Settling time (s)	Longitudinal position error (m)	Vertical position error (m)
Fully learned controller	3.390 ± 0.792	0.006 ± 0.002	0.004 ± 0.001
Cascade controller	11.799 ± 2.027	0.046 ± 0.017	0.030 ± 0.012
Pose PID	5.182 ± 1.359	0.007 ± 0.009	0.005 ± 0.006
Single env. training	3.853 ± 1.488	0.013 ± 0.044	0.009 ± 0.031

As expected, PID or PID-aided controllers, especially with an integral gain tuned, displays higher steady-state errors than the fully learned controller. However, the latter (PID-aided) could still improve this error with fine tuning of the training hyperparameters. These errors, among with the inability to stabilize the quadrotor with certain parameters might be caused by the saturation of PID commands (ANN controllers already have commands limited within bounds from the tanh activation function in their output layers.), which are, then, unable to quickly steer the quadrotor to stability. This becomes clear in figure 5.10, where the pose PID commands are registered without saturation (but still saturated when sent to the simulated motors). The figure shows that the PID controller is clearly hindered by the motor velocity saturation.

The reward function is a weighted measurement of error. Because of this, we can observe the reward distribution of experiments for a comparison of accumulated error through each experiment, in figure 5.11 and Table 5.4.

Table 5.4: Mean and standard deviation total rewards for the waypoint guidance task for 100 successful samples of each controller, including single environment (non-randomized) training.

Controller	Total reward
Fully learned controller	-78.715 ± 22.465
Cascade controller	-86.133 ± 23.822
Pose PID	-125.414 ± 55.156
Single env. training	-98.393 ± 59.741

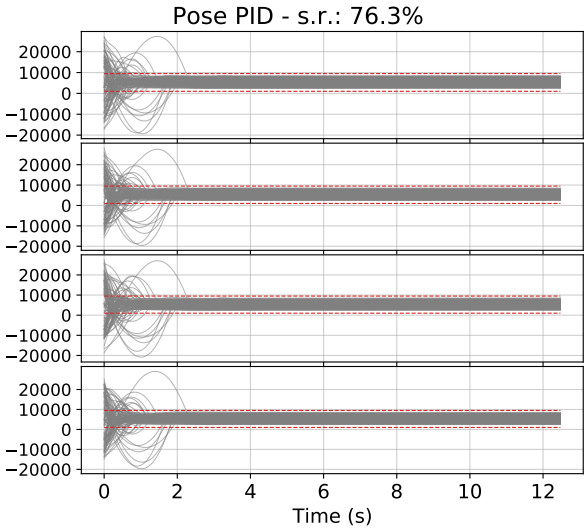


Figure 5.10: Unsaturated motor commands of the quadrotor over time for the pose PID controller in the waypoint guidance task. The red dashed lines indicate the velocity where commands are saturated when sent to the quadrotor.

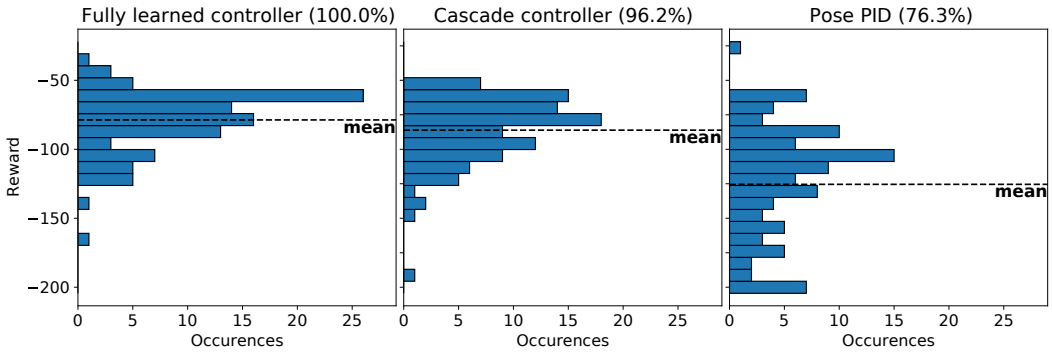


Figure 5.11: Distribution of total rewards attained per experiment to perform the waypoint guidance task (21 bins). For the pose PID, 6 of the experiments ended up with less than -200 total reward.

Despite the lower robustness overall, the pose PID controller, given enough time to settle, displays less steady-state errors than the controller learned on a single set of parameters, included in Tables 5.3 and 5.4 for comparison, further demonstrating the necessity of environment randomization in the training process of a controller robust to these variations. However, in terms of settling time and total reward, an RL controller is still preferable in comparison to the pose PID.

5.1.4 Performance evaluation of the payload pickup task

Similarly to the waypoint guidance task, figure 5.12 displays the expectation of a given parameter combination to perform the payload pickup and drop task successfully.

The experiments using the fully learned controller and the cascade controller obtained full success out of 100 tests, while the ones using the pose PID controller obtained 100 successful tests out of 160.

For these experiments, the environment parameters are resampled from the constraints specific for the payload pickup task. As previously mentioned,

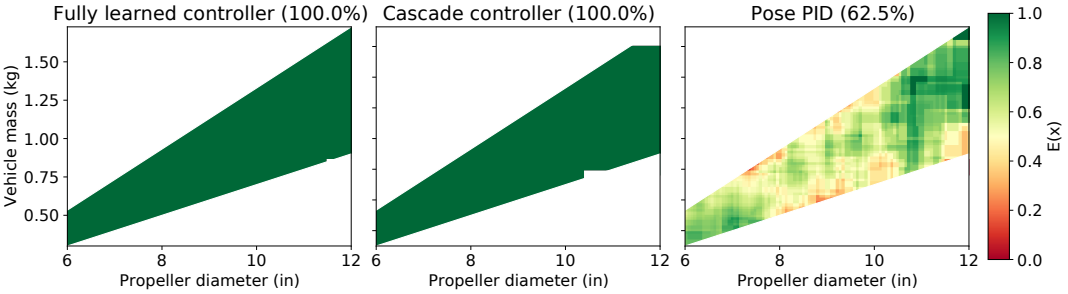


Figure 5.12: Expected success rate (s.r.) for each controller in the payload pickup and drop course, by combination of quadrotor parameters.

the criterion for selection of learned controllers out of the 10 training runs is that it attains 100% success rate in the mentioned task, displaying an expectation of 1 for every parameter combination.

From these results, as expected, from the previous results of waypoint guidance task, the pose PID controller has poor flexibility to the variation of quadrotor parameters, even if its gains are obtained using a search strategy on the same variety of vehicle parameters.

In the payload pickup experiments, due to the criteria of convergence to a waypoint, experiments with high position error or that take too long to reach a waypoint are filtered out (considered a failed test). A fitting metric to evaluate this task, further than the success rate, is the time taken to perform it, as it encompasses both the route optimality and the waypoint settling time. Figure 5.13 shows the histogram of this metric for each controller, where tests are allocated in 11 bins distributed between the minimum time taken among all 3 controllers, 6.750 seconds, and the maximum time, 24.6 seconds.

Learned controllers have proven to be clearly more time-efficient than the pose PID controller. Analyzing the resulting distribution, we calculate the mean and standard deviation of results in Table 5.5.

Table 5.5: Mean and standard deviation of the time taken to complete the payload pickup and drop course, for 100 successful samples of each controller.

Controller	Course completion time (s)
Fully learned controller	9.280 ± 1.492
Cascade controller	9.340 ± 1.515
Positional PID	14.499 ± 3.271

Figures 5.14 and 5.15 show the heatmap of the compound trajectories of the quadrotor in each successful experiment. In the former map, the trajectory

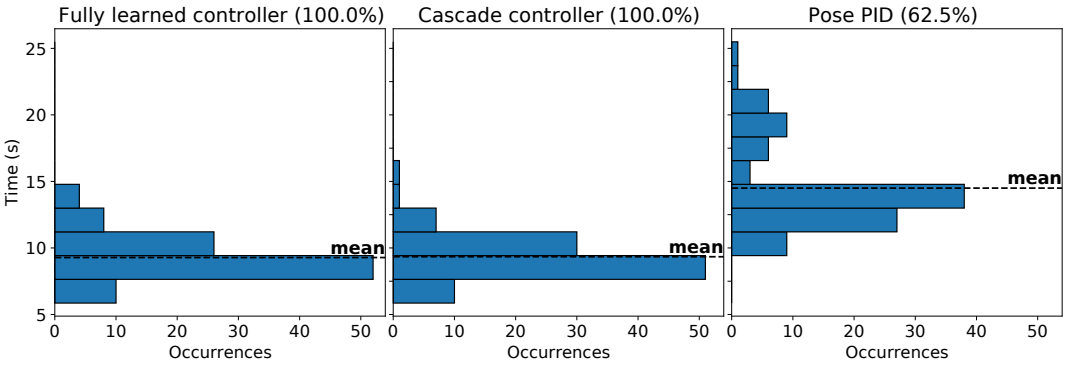


Figure 5.13: Distribution of total time taken per experiment to perform the payload pickup and drop course (11 bins).

is viewed from above, projected in the XY plane, while in the latter map, the trajectory is viewed from the side, projected in the XZ plane.

In figure 5.14, we can observe the effects of the maximum entropy RL of the SAC algorithm in action, as the trajectories taken by the quadrotor are more spread out through the space than the trajectories generated by the PID controller.

In figure 5.15, it becomes clear how the variation of mass heavily affects the trajectory generated by the pose PID controller. In this trajectory, the controller converges first horizontally, then vertically, to the waypoint, possibly due to the apparent slowness of the integral controller to adapt to the new mass, as evidenced in figure 5.7 by the difference in convergence speed and steady-state error between the horizontal (X,Y) and vertical (Z) components of the controller. In contrast, the trajectories generated by the learned controllers are smoother, possibly because there is a single coupled black-box controller acting in all directions, instead of the uncoupled nature of the PID controller.

In summary: The learned controllers are able to perform the guidance task with almost all variations of propeller diameters and vehicle mass (100%

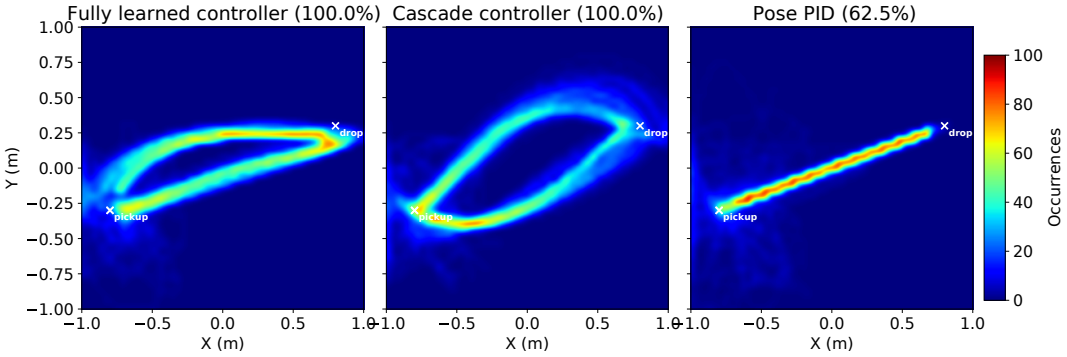


Figure 5.14: Heatmap of the route taken by each experiment in the payload pickup and drop course, viewed from above.

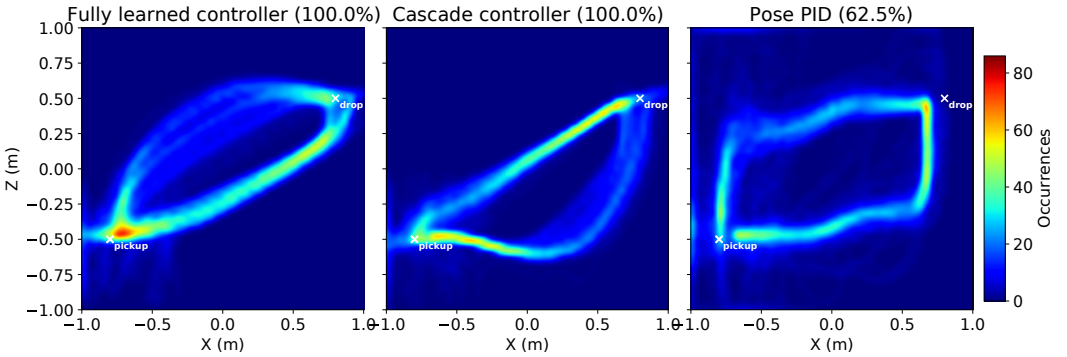


Figure 5.15: Heatmap of the route taken by each experiment in the payload pickup and drop course, viewed from the side.

for the fully learned controller and 96.2% for the cascade controller), while the pose PID only does so for 76.3% of these variations. Note that these variations are within the same parameter ranges used during training and hill-climbing parameter searches. Despite displaying similar success rate, the learned controllers behave differently when performing the present task. The cascade controller converges to the waypoint slower than the fully learned controller and oscillates significantly more in the steady-state. The pose PID controller does not display such oscillations, but it is still slower than the fully learned controller. This is caused by a saturation in motor commands, which slows down the convergence of the quadrotor with certain parameter combinations. Then, we employ the waypoint guidance controllers in the payload pickup and drop course. To complete this course, both learning-based controllers take about the same amount of time, in average 9.280 seconds for the fully learned controller and 9.340 seconds for the cascade controller, which is faster than the pose PID controller, that takes, in average, 14.499 seconds to perform the same course. This slower completion time is expected, as the waypoint guidance controller evaluation shows that the pose PID controller suffers from motor saturation and the lack of extra robustness mechanisms in its designs. Further, the pose PID controller is only able to perform this course for 62.5% of the analyzed cases, failing either due to destabilizing (the same reason it attains a poor success rate in navigating towards a waypoint) or to being unable to reach the distance threshold to one of the waypoints.

5.2 Visual navigation results

In this section, we lay out the results of development and testing of the learning-based visual navigation controller. The key experiments found in this section are:

- CNN pre-train and advantages of training data resampling;
- Learning performance of the navigation models; and
- Course navigation performance.

The navigation models evaluated are: the baseline controller, which navigates through the race course using real gate position data, it is, without visual data, and a visual navigation controller, which navigates through the race course by estimating gate positions via a CNN.

5.2.1 Pre-trained model

The CNN pre-training process leverages the capability of the environment to resample new data on demand, using an active learning-based method. This feature allows the training algorithm to populate the training dataset gradually with more relevant data, resampled at every test step. In other words, we direct the training process through samples that display low prediction accuracies in order to speed up the learning process. In 4000 training epochs, this method scores, in average, 0.492 in the weighted loss function (eq. (3-2)) for an independent dataset. On the other hand, a conventional training process with fixed training and test sets scores, in average, 0.775 in the weighted loss function, for the same independent dataset. The comparison between test losses during training for each approach can be seen in figure 5.16.

For the 2000 samples independent dataset, figure 5.17 shows the root-mean-squared prediction error for both training methods after 4000 epochs, further demonstrating that the active learning method results in a significant improvement in prediction accuracy.

The weights transferred to the actor-critic model are obtained after 12000 training epochs. In the independent dataset, the prediction model at this point scores, in average, 0.337 in the weighted loss function. For the same independent dataset, figure 5.18 shows how the prediction error at this point compares to the prediction error shown in figure 5.17, at 4000 training epochs.

The full metrics of the training process of this network are laid out in figure 5.19. Notice how the training loss rises when the dataset receives particularly bad training samples, but quickly adjusts to those.

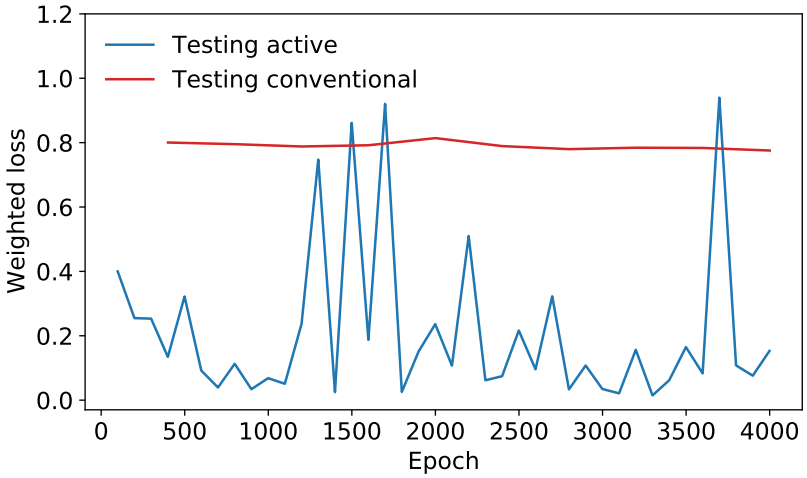


Figure 5.16: Test loss per training episodes of two different training strategies.

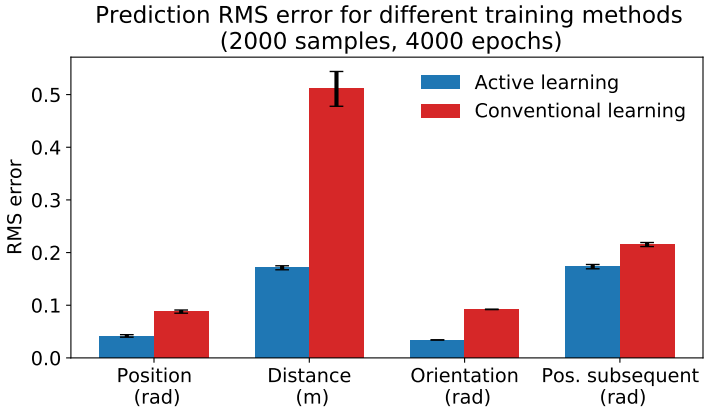


Figure 5.17: RMS error of prediction of two different training strategies (less is better), with 95% confidence interval bars. Note that the gate position is given by its angle with respect to the camera.

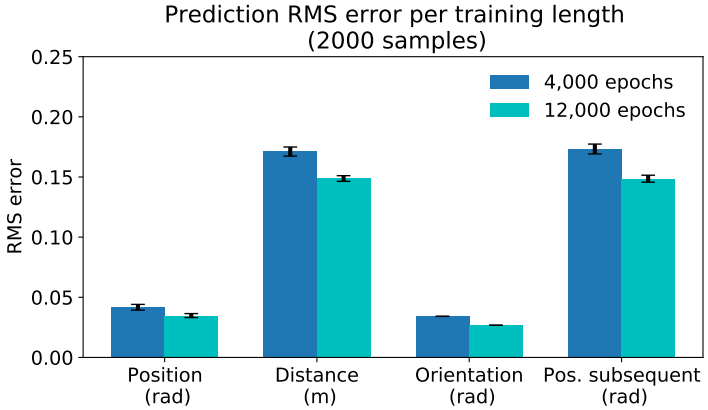


Figure 5.18: RMS error of prediction of the active learning method compared between training epochs (less is better), with 95% confidence interval bars. Note that the gate position is given by its angle with respect to the camera.

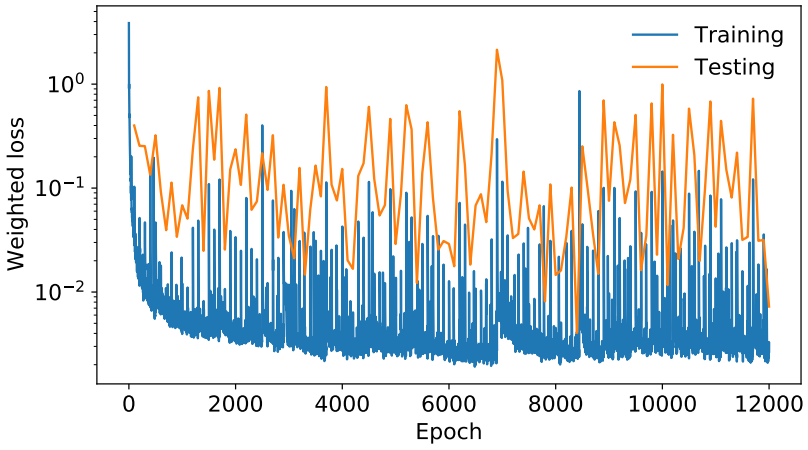


Figure 5.19: Training curves of the pre-trained network.

5.2.2 Learning performance

During the training process, the quadrotor is always spawned in a random location along the track, including during test episodes, which happen twice every 10 steps for every run, so that the total attained reward is not a good metric for measuring the training performance. Hence, we opt to measure the percentage of the course completion for each run, as the course length varies randomly for each test.

Figure 5.20 shows the course completion rate for the baseline navigation controller. This controller is trained in 10000 episodes, taking a total of 6 days and 20 hours to complete. Due to the steady rise in performance, it is possible that the course completion rate improves given enough time. However, we have chosen to limit the training time of the controllers to a week at most.

In order to further demonstrate the suitability of this training algorithm for this task, figure 5.21 shows the moving average of the course completion rate for four independent training runs up to episode 4000, one of them (the darker line) being the run of figure 5.20. All four, in average, attain similar performance throughout the training episodes.

Then, the trained weights from the baseline controller are transferred to the dense layers of the visual navigation model, along with the pre-trained CNN weights. The CNN weights are frozen for the first 200 training episodes in order to let the actor and critic dense layers adapt to this new input. The result is shown in figure 5.22.

Notice that the model learns to properly traverse a considerable portion of the course within the first 200 episodes, during which the CNN layers are frozen. Then, the training promptly diverges, as evidenced by the actor and

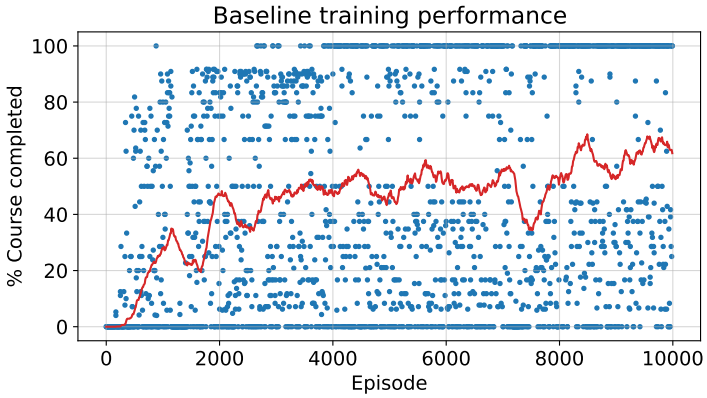


Figure 5.20: Percentage of the course completed during training of the baseline controller, with a 100-sample moving average.

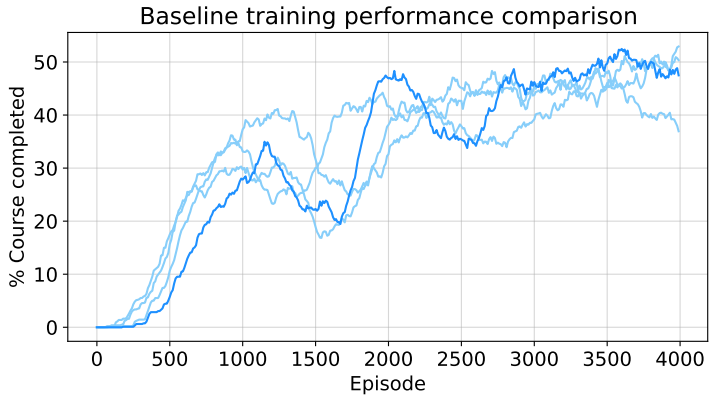
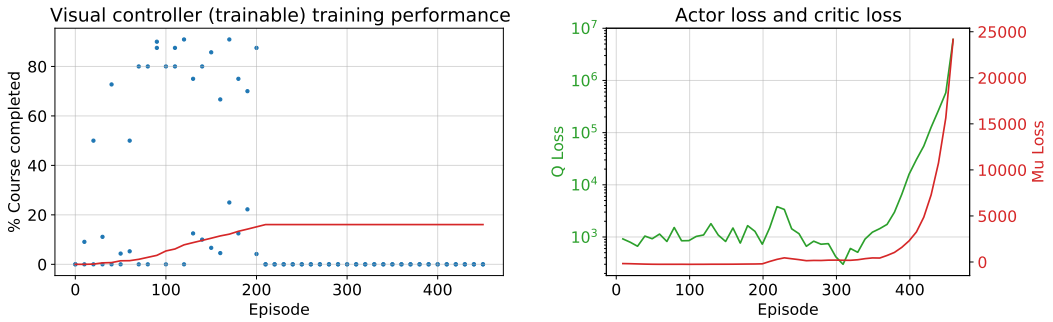


Figure 5.21: Percentage of the course completed (100-sample moving average) for 4 training runs of the baseline controller up to episode 4000.



(a) (b)

Figure 5.22: Training metrics for the visual controller. (a) Percentage of the course completed during training, with a 100-sample moving average. (b) Actor and critic losses.

critic losses in figure 5.22b.

In view of the satisfactory performance of the model while the CNN weights are frozen, we perform a training run with the CNN weights permanently frozen, resulting in the course completion rate of figure 5.23.

This model is trained in 8830 episodes, taking a total of 4 days and 12 hours, until interrupted[2]. Despite not displaying improvement in performance throughout training, convergence to locally optimal parameters is fast, as evidenced by the first sample reaching 100% course completion doing so within 70 training episodes. By episode 500, on a 100-sample average, the completion rate is 48.4%, while the baseline average is 61.8% in the last training episode. Hereby, we consider the loss of performance on the task at hand to be approximately 21.7%, when predicting the gate features by a neural network

[2]Due to an unexpected simulator crash. Training was not resumed since it was close to automatic termination at 10000 episodes and there was no apparent change in controller performance in the last few thousand episodes.

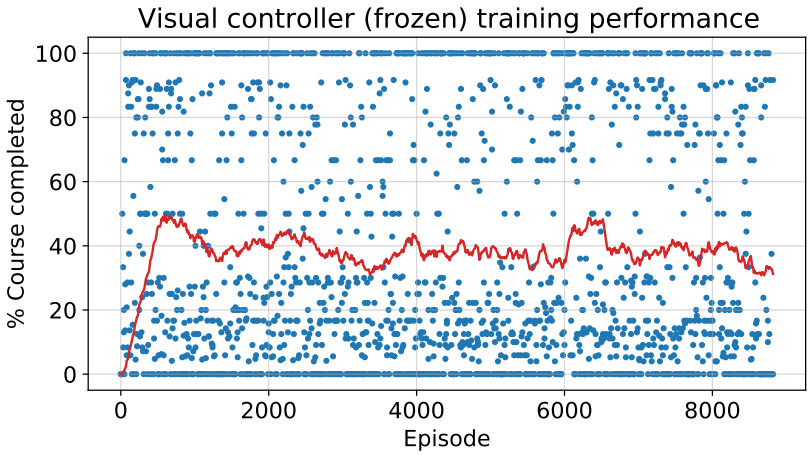


Figure 5.23: Percentage of the course completed during training of the visual controller with frozen CNN weights, with a 50-sample moving average.

compared to ground truth values.

5.2.3 Course navigation

In this task, the quadrotor relies on the trained controllers to traverse the racing courses of figures 4.3 and 4.4. Despite being trained for longer, the performance of the visual navigation controller is better at the 500th episode, so that, for this controller, we use the weights of this episode. On the other hand, for the baseline controller, we use the weights of the last episode.

For the *Soccer_Field_Easy* course, we perform 10 tests starting at a fixed point 2 meters above a *Start_Block* object near the origin of the 3D space. Figure 5.24 shows the trajectories and end results of the baseline controller and figure 5.25 shows the trajectories and end results of the visual navigation controller.

Each of the experiments using the baseline controller is able to successfully complete the course. Meanwhile, for the experiments using visual navigation, only 5 out of 10 managed to complete the course. Out of the 5 unsuccessful tests, one has collided with Gate 01, 2 have lost visual contact with Gate 08, after trying to cross it and missing, and another 2 have lost visual contact with Gate 09, soon after crossing Gate 08. In this way, the visual navigation controller manages to achieve 79.2% of the baseline performance, when starting from Gate 00.

For the *Soccer_Field_Medium* course, no controller managed to complete the task starting from the beginning (at the *Start_Block*). With this issue in mind, we perform 10 tests starting from the initial point 2 meters above the *Start_Block* and, wherever all these tests have failed, 10 additional tests are

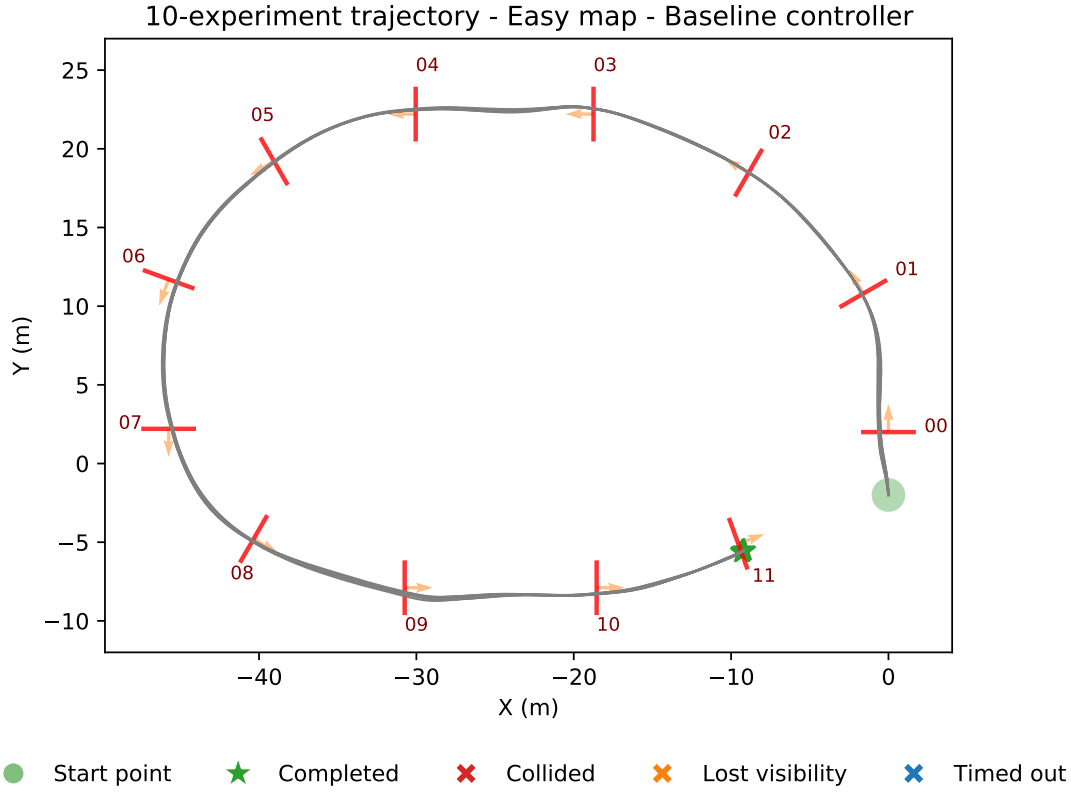


Figure 5.24: Baseline controller trajectories on the easy course (top-down view).

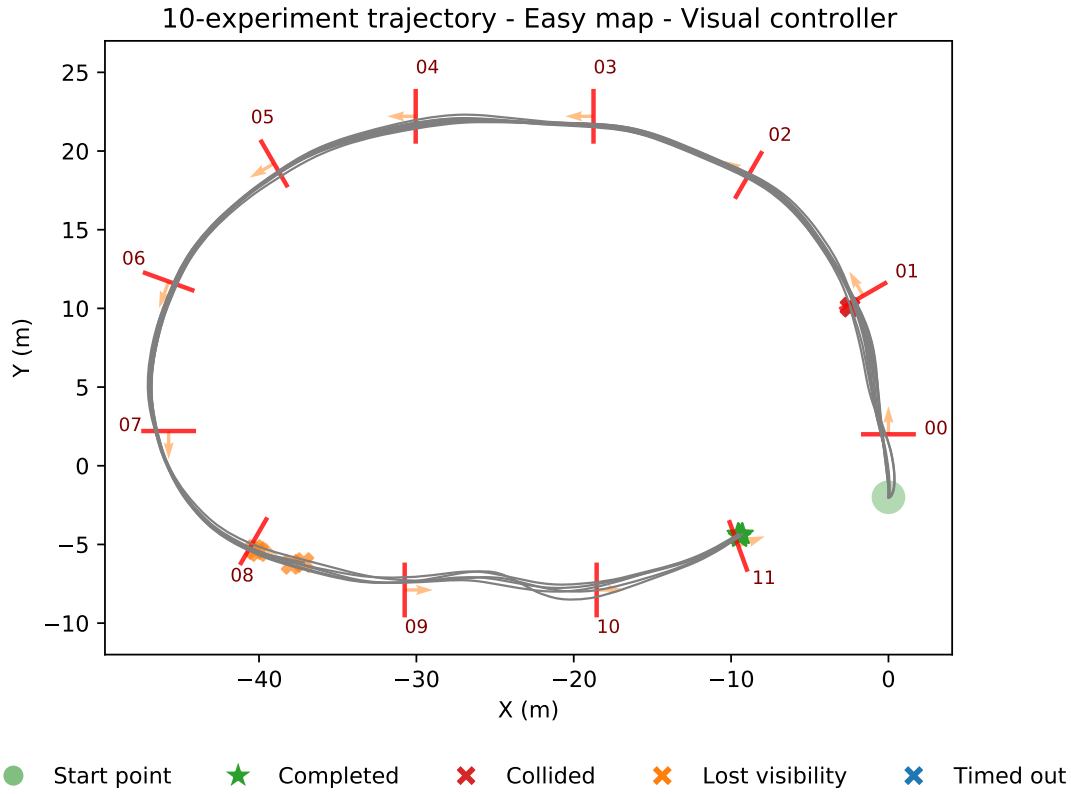


Figure 5.25: Visual controller trajectories on the easy course (top-down view).

respawned. The respawn point is the midpoint between the last gate to be successfully crossed and the next gate. In this way, the quadrotor attempts to cross all the gates in the course. Figure 5.26 shows the trajectories and end results of the baseline controller and figure 5.27 shows the trajectories and end results of the visual navigation controller.

In both experiments, we can observe three key struggle points in the medium course. Between Gates 04 and 08 there is a steep descent, with tightly spaced gates. The trajectories generated in this section are shown in figure 5.28, for the baseline controller (which was successful in this section), and figure 5.29, for the visual navigation controller.

Because of their placement, the gates are partially seen after a previous gate is crossed, posing a challenge to the position estimator used in the visual navigation controller. Notice how an important part of the trajectories terminate above the gate, unable to see it. The baseline does not struggle in this point, because it does not rely on the camera to be able to estimate gate positions. Another interesting observation is how, in some of the successful gate crossings, the quadrotor adjusts its position before attempting to cross. This behaviour is not found in the baseline controller, so we assume it was learned specifically for visual navigation in an attempt to localize the gate before crossing.

The section between Gates 10 and 14 has some of the largest spacings between gates in the course. From the baseline trajectories, it becomes clear that the controller struggles to navigate towards distant gates. Nevertheless, the quadrotor using the visual controller is able to fly farther, in average, between gates 13 and 14, than using the baseline controller.

Between Gates 19 and 22 there is a combination of the two aforementioned issues, Gates 19 and 20 are very far apart and there is a steep ascent between Gates 20 and 22. The trajectories generated in this section are shown in figure 5.30, for the baseline controller, and figure 5.31, for the visual navigation controller.

While all the collision incidents in this section happened using the baseline controller, the quadrotor using the visual navigation controller flew shorter distances in comparison, losing visual contact with the gate more frequently. After this section of the course, all tests that manage to cross Gate 22 were able to complete the course successfully.

Neither of the controllers managed to complete the medium course starting from Gate 00. Additionally, the visual navigation controller had to restart the race twice the time (12 restarts) compared to the baseline controller (6 restarts), which further evidences the difficulty of the visual estimator to

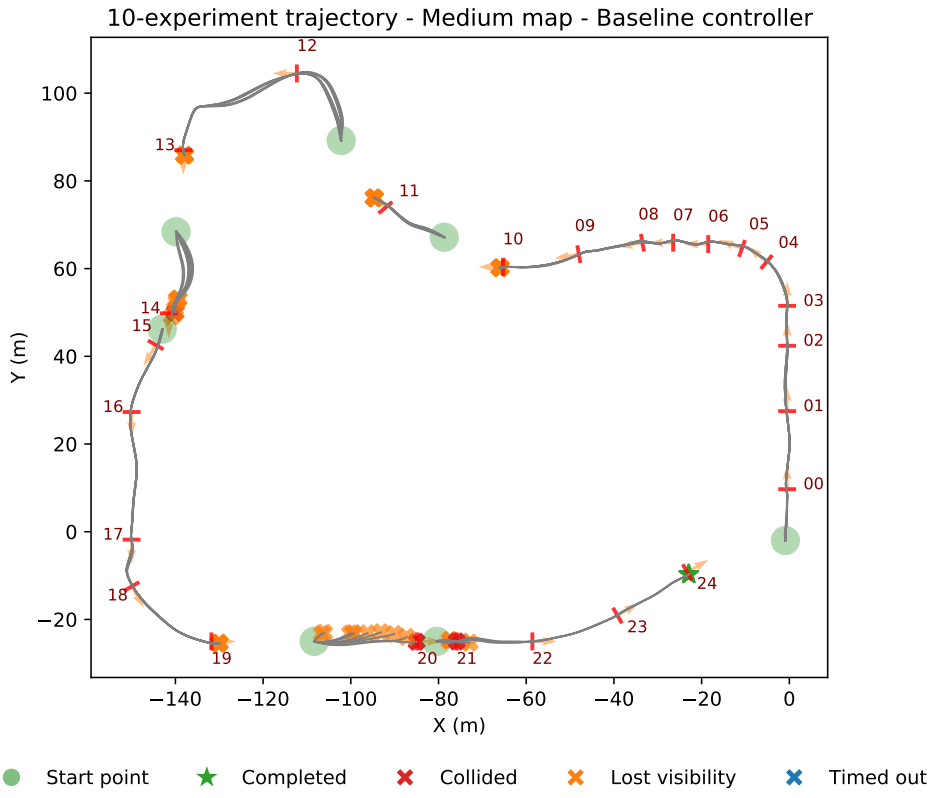


Figure 5.26: Baseline controller trajectories on the medium course (top-down view).

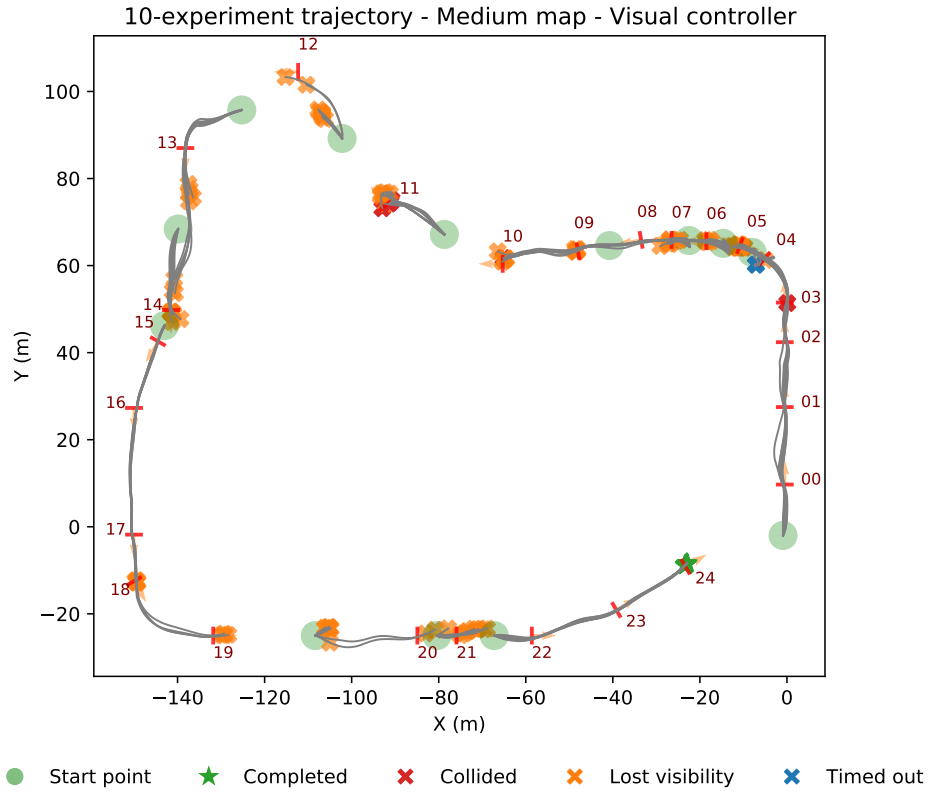


Figure 5.27: Visual controller trajectories on the medium course (top-down view).

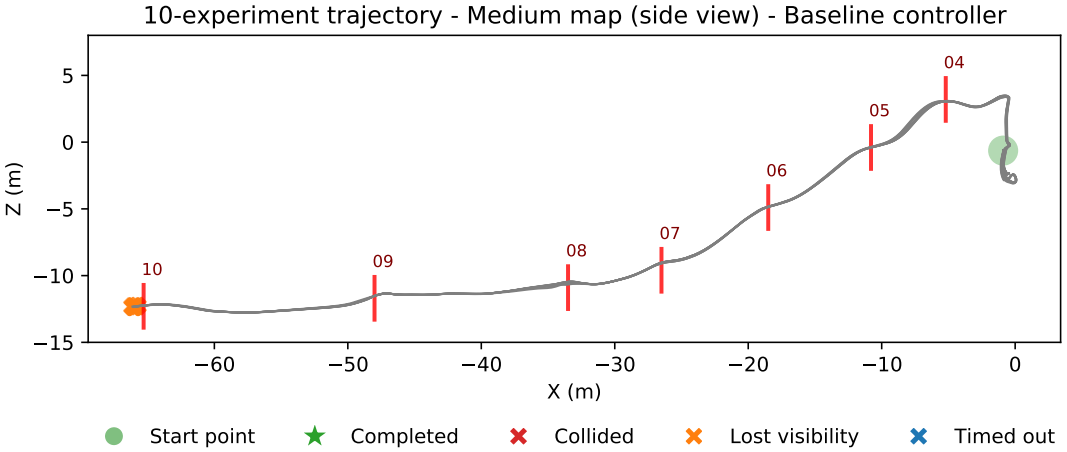


Figure 5.28: Baseline controller trajectories on the Gates 04-10 section of the medium course (side view).

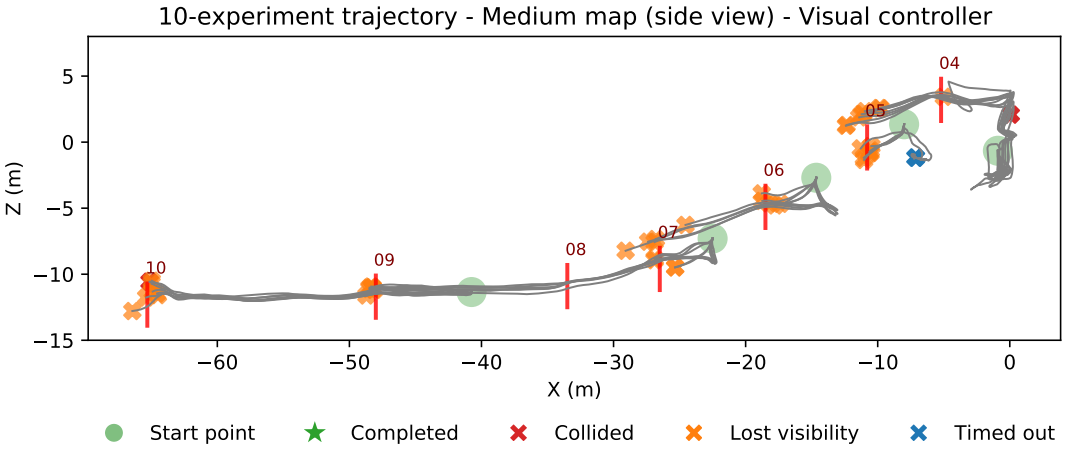


Figure 5.29: Visual controller trajectories on the Gates 04-10 section of the medium course (side view).

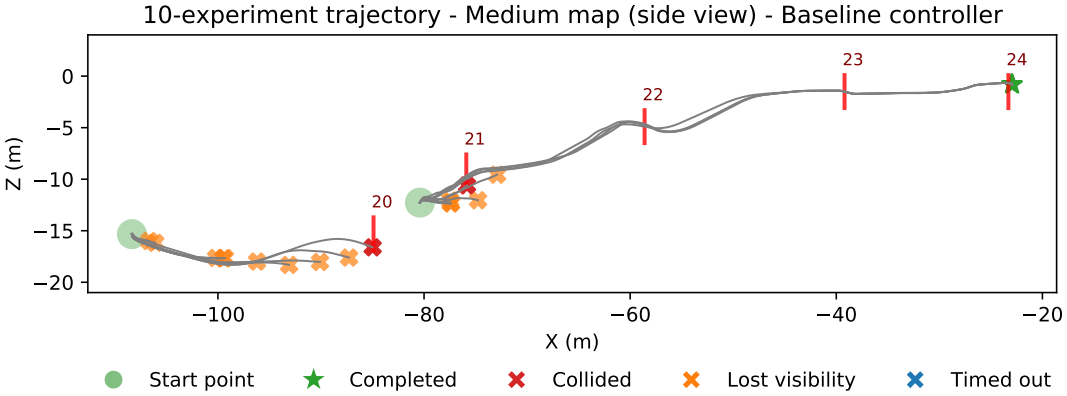


Figure 5.30: Baseline controller trajectories on the Gates 20-24 section of the medium course (side view).

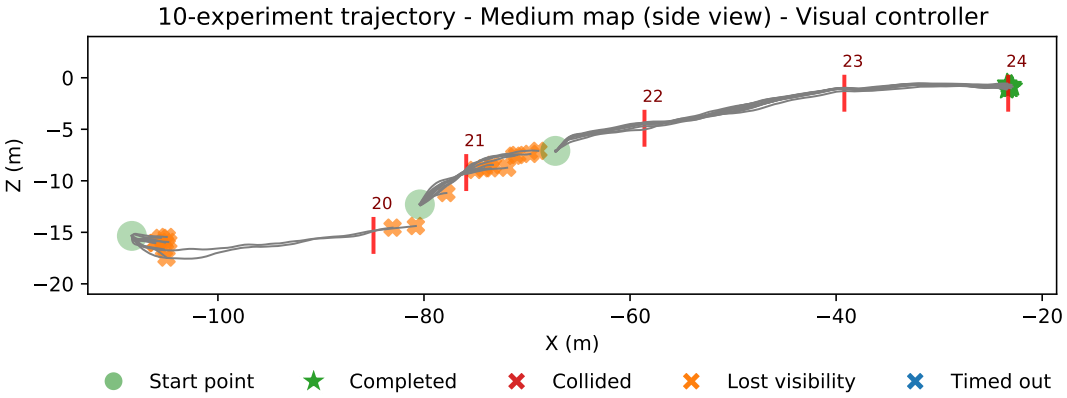


Figure 5.31: Visual controller trajectories on the Gates 20-24 section of the medium course (side view).

detect gates in the medium course. To equally evaluate the performance of each controller, we carry out 100 extra tests for each controller starting from random points along this course, with the starting points chosen by the same method used during training.

Out of 100 tests in the medium course, for the baseline controller:

- The quadrotor crossed a total of 359 gates out of 447 attempts (overall 80.3% success, approximately);
- The final gate was reached 12 times. 87 experiments ended with no visual contact with the gate and one ended with a collision.

Out of 100 tests in the medium course, for the visual navigation controller:

- The quadrotor crossed a total of 128 gates out of 225 attempts (overall 56.9% success, approximately);
- The final gate was reached 3 times. 87 experiments ended with no visual contact with the gate, 6 ended with a collision and 4 timed out;
- Considering the amount of gates attempted, compared to the baseline controller, the visual navigation controller had a relative performance of approximately 70.8%.

As mentioned previously, these experiments are performed using the visual navigation controller trained until episode 500. Despite the learning metrics, presented in figure 5.23, showing that there is no significative change in performance throughout further training episodes, there is indication that the model has overfitted at this point. Using the weights trained until episode 8500 for the visual navigation controller, we perform 10 tests for each course starting from the beginning (Gate 00). This controller achieves a performance similar

to the baseline controller in the section between Gates 04 and 08 (figures 5.28 and 5.29) for 3 out of 10 experiments. However, it fails every time at the easy course, colliding with Gates 00 and 01.

In summary: The baseline controller manages to avoid collisions and provides a control policy that can reliably complete the easy course and cross most of the gates in the medium course. Combining this baseline controller with a gate feature predictor network, we obtain the visual navigation controller. This controller is fine-tuned through the same SAC algorithm used for the baseline controller, with the same hyperparameters and reward function. Unexpectedly, allowing the CNN weights to be fine-tuned by this method leads the training process to diverge. In view of this, we fine-tune only the base actor-critic weights, maintaining the original CNN weights fixed, resulting in a satisfactory control policy, which is able to race through the easy course with 79.2% performance, compared to the baseline, and 70.8% performance, compared to the baseline, for the medium course. This drop in performance is caused by a struggle of the visual controller to navigate in cramped sections of the course. Eventually, with more training episodes, the controller manages to navigate properly in these sections, at the cost of reduced performance in other sections, highlighting a clear limitation in learning capacity of this model. Nonetheless, even the easy racing course is challenging enough for a visual navigation controller developed solely based on machine learning. Thus, acquiring a 79.2% performance compared to a perfect run can be considered a satisfactory result.

6
Conclusion

In this work, we evaluate the feasibility of employing reinforcement learning in the development of a low-level pose controller for a quadrotor, aiming at some specific tasks. Starting from a proof-of-concept unified waypoint guidance plus stability controller, we establish a development method that can be carried over for more challenging tasks. More specifically, a payload pickup and drop task and a racing course.

Low-level control: Development

Previous work often relies on building a dynamic model of the quadrotor (model-based reinforced learning) or use auxiliary controllers in the control stack. Our method, in contrast, seeks to train a controller using a Soft Actor-Critic (SAC) algorithm, that calculates motor commands directly from sensory input (end-to-end), using a dense 2-layer neural network, while also being agnostic, in other words, being applicable to a range of vehicle sizes and masses. This feature is also very useful in situations where the quadrotor parameters change mid-flight.

To compare this method to existing approaches to RL controllers, we train a second high-level controller which works in cascade with a low-level velocity PID controller, investigating the effectiveness of an independent, faster and conservative underlying control loop in learning and in task performance. For benchmarking, we develop two additional controllers. A full RL controller, trained in an environment with fixed parameters, and a full pose PID controller. Both PID controllers, velocity and position, use optimized gains calculated from a hill climbing search strategy in a randomized environment, similarly to the training procedure of the RL controllers. This ensures well-performing gains for fairness of comparison between experiments. For all learned controllers, we perform 10 training runs and pick the one that results in the highest success rate in the payload pickup task, using the lowest position error as a tiebreaker.

Low-level control: Waypoint guidance results

Within the same range of testing parameters, the RL controller trained in a single environment attained significantly higher success rate than the pose PID controller. The trajectories generated by both controllers indicate a struggle to climb to the height of the waypoint due to different propeller diameter×mass ratios, causing the base motor input required for hovering to vary between environments. While the only issue with the PID controller is its slowness to navigate towards the vertical position, the learned benchmark controller is unable to do so, ending episodes hovering with significant position errors. This evidence supports the speculation that the controller trained in a single environment is unable to generalize for different propeller diameter × mass ratios, thus indicating it has overfitted to its training parameters. Nevertheless, the RL controller is able to account for a wider range of quadrotor parameters than the PID controller, even if the gains of the latter have been optimized for the same range.

Both the fully learned controller and the cascade controller take about the same amount of training episodes to converge to a locally optimal policy, displaying very proximate position errors. The fully learned controller is able to perform the waypoint guidance task with all tested combinations of propeller diameter and vehicle mass, while the cascade controller managed almost all combinations, still attaining a substantially higher success rate than the pose PID controller. Despite displaying similar success rate, the learned controllers behave differently when performing the task considered. The cascade controller takes much longer to converge to the waypoint than the fully learned controller and oscillates considerably in the steady-state, causing it to not meet the convergence requirement (total velocity should not exceed 0.05 m/s) in almost half of successful tests. This response is the result of the internal velocity setpoint provided by the control policy changing too rapidly, which amplifies the oscillations of the control actions computed by the PID controller. The pose PID controller does not display such oscillations, despite being slower than the other controllers. This controller, however, suffers from motor saturation, which is believed to hinder its performance, as it is unable to steer the quadrotor using "full force". The learning-based controllers do not suffer from this hindrance for two reasons: the fully learned controller already provides bounded commands due to its bounded activation function in the output layer, and these controllers have the opportunity to learn a policy that circumvents this obstacle. Lastly, by the accumulated position error during the episode, the fully learned controller has the best overall performance, closely followed by the cascade controller. By

this metric, the PID controller performed poorly compared to other controllers.

Low-level control: Payload pickup results

Then, we employed the waypoint guidance controllers in the payload pickup and drop course. This course consists of two waypoints: the first in a low altitude where the quadrotor picks up a payload weighting 30% of its current mass and the second in a higher altitude where the payload is released, then prompting the quadrotor to return empty to the first waypoint. Both learned controllers, as a requirement for selection from the training runs, achieved 100% success rate in this task for all the tested environment parameters, and take about the same amount of time to complete it, being significantly faster than the pose PID controller. Further, the success of the pose PID controller in completing this task falls by 18.1% with respect to the waypoint guidance task, failing either due to destabilizing (the same reason it attains a poor success rate in navigating towards a waypoint) or to being unable to reach the distance threshold to one of the waypoints before timing out.

In summary, for the first objective of this work, the fully learned controller (trained in randomized environments) is not only able to handle a wider variety of quadrotor sizes and masses than the benchmark controllers, but it also performs better in the proposed tasks. This is expected, because the quadrotor model is non-linear, while the PID controller is designed for linear systems. Since the learned controllers are based on a neural network structure, they better address these non-linearities. The learning-based controllers also learned to circumvent other challenges present in the model, such as motor command saturation. Another advantage of the RL controller is its end-to-end learning capability, which allows, for example, to forego manual PID gains tuning. As for the payload pickup task, both the fully learned controller and the cascade controller take the same amount of time, in average, to complete it. The difference between these controllers lies in the mentioned advantage of foregoing PID tuning for the former and the oscillatory response of the latter, which might be undesirable in some applications.

Visual navigation: Development

With the development methodology for the controllers established, we move on to a more challenging task, a racing course, performing training with the same RL algorithm, SAC. For this task, the RL controllers learn to navigate towards gates in two different racing courses as if they were

the waypoints of the payload pickup task. Each course presents a different completion difficulty: easy and medium. The racing environment does not provide the 3-dimensional coordinates and yaw of the waypoint directly. Thus, we develop a controller that estimates the crossing direction of the next gate and the relative position of the two nearest gates, using visual data in order to devise a trajectory through them. To benchmark this controller, we also develop a baseline controller, that receives ground truth gate features as inputs.

A full end-to-end training process for the racing course was found to be excessively lengthy and too computationally demanding, for the available hardware. Therefore, we divided this process in two parts that are trained independently. The first is the actor-critic controller part, which has its weights initialized from the fully trained baseline controller. The second part is the visual estimator for gate features, using a DroNet architecture, which is trained in a supervised fashion using the samples from the baseline controller. The visual estimation model leverages the capability of the simulation to generate new data to employ an active learning based training method, that results in a higher prediction accuracy than a conventional learning method with static training and testing datasets. In the end, both parts are combined and put through training again in order to fine-tune their network weights.

In four independent training runs of 4000 episodes, the learning process of the baseline controller converges to a satisfactory course completion rate, indicating the reliability of this method to teach a quadrotor to navigate towards the gates in the environment. The final baseline controller is trained further for a total of 10000 episodes and displays, in average at the end, 61.8% course completion rate starting from random points in both courses.

After combining the baseline controller with the pre-trained visual estimator, we freeze the estimator weights for the first 200 training episodes to let the baseline network adapt. Unexpectedly, the controller becomes unable to cross any gates as soon as the estimator weights are unfrozen and the learning process completely diverges soon after.

Nevertheless, while the estimator weights are still frozen, the controller managed to attain a satisfactory course completion. In view of that, the estimator weights are left frozen for the remainder of the training process, attaining, in average, 48.4% completion rate by episode 500, representing a 21.7% drop in performance with respect to the last training episode of the baseline controller.

In posterior training episodes, the completion rate does not change significantly, but the controller displays a slight change of behaviour, favouring an improved performance in particular sections of the medium course at the

expense of a heavy performance loss in the easy course and in some other sections of the medium course. Thus, experiments were done with the model trained up to the 500^{th} episode mark.

Visual navigation: Results

Both controllers, baseline and visual, managed to complete the easy course starting from the beginning. The baseline controller attained only perfect runs out of 10 tests. The visual controller managed 5 perfect runs out of 10 tests, with 4 runs ending by losing visual contact with the gates, happening around the final stretch, and one ended by a collision with a gate near the starting point.

Both controllers struggled to complete the medium course. The 3 main choke points present one of two key characteristics: the gates are too far apart or the gates are too close together, in a steep ascent or descent. While both controllers have a drop in performance with respect to the easy course, the visual navigation controller sees a larger drop than the baseline controller. This is caused mainly by the disturbances of the visual feature estimations when the camera is too close to one of the gates, which is frequent in cramped sections of the medium course.

Nonetheless, the visual controller displays a learned behaviour which is not present in the baseline controller, possibly driven by the mentioned struggles of the visual estimator: in an early section of the medium course, where the gates are close together in a steep descent, the quadrotor has learned to back up before crossing the gate, possibly to get a better visualization of it and any subsequent gates.

In summary, despite the mentioned disturbances in the visual feature estimator, the visual navigation controller is still able to cross a large number of gates in succession, indicating it has learned to overcome the estimation noise and disturbances, most of the time, with just a few episodes of fine-tuning. Visual navigation displays about 70%∼80% efficiency compared to navigation using ground truth gate features.

6.1 Future work

The evident next step of this work would be to investigate these controllers in a real-life quadrotor. As discussed in some of the related works, the main challenge when taking this step, assuming an adequate testing platform is readily available, would be dealing with sensor noise and deviation, especially

if an inertial frame position sensor is required. Hopefully, the versatility of the learned controller for different vehicle parameters smooths out this transition, as it is very difficult to accurately model a real quadrotor in simulation.

In this work, we developed parameter-agnostic waypoint guidance controllers for plus-shaped quadrotors only. We suggest that such a controller could be modified to also account for geometrical variations of drones, such as quadrotors with movable arms, or picking up an off-centered payload, or even dealing with damaged parts mid-flight. This agnosticism can also prove useful for various multitasking applications, not only restricted to payload carrying, but any activity that imposes sudden changes in vehicle parameters.

The controller proposed in this work competes directly with the purpose of traditional robust and adaptive controllers. These controllers were not used for benchmarking, because of the limited time and manpower available for the development and debugging of such algorithms. Even with the advantage of end-to-end learning provided by RL methods contributing to a faster development and deployment, adaptive controllers may also be an alternative for at least the waypoint guidance and payload pickup tasks proposed in this work.

The solution presented in this work for drone racing employs only the visual estimation model and the RL control approach. Competition solutions and real-life applications commonly employ other devices to improve estimation accuracy and navigation reliability, such as model predictive control, gate feature estimation with a latent state space, Kalman filters, *et cetera*. These solutions also employ more than one visual estimation model, for simultaneous object detection and scene segmentation, in order to detect obstacles and race checkpoints of different shapes and colors. For a full completion of the medium racing course and other challenging drone racing courses, we must employ such devices to circumvent the limitations of the RL approach.

Bibliography

[1] TURING, A. M.. **I.—Computing Machinery and Intelligence**. Mind, LIX(236):433–460, 10 1950.

[2] SUTTON, R. S.; BARTO, A. G.. **Reinforcement Learning: An Introduction**. A Bradford Book, USA, 2018.

[3] **Robots and robotic devices**. Vocabulary – iso 8373:2000(e), International Organization for Standardization, Geneva, Switzerland, Mar. 2012.

[4] TRIPATHI, S.; DANE, G.; KANG, B.; BHASKARAN, V. ; NGUYEN, T.. **Lcdet: Low-complexity fully-convolutional neural networks for object detection in embedded systems**. In: 2017 IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION WORKSHOPS (CVPRW), p. 411–420, 2017.

[5] LILLICRAP, T. P.; HUNT, J. J.; PRITZEL, A.; HEESS, N.; EREZ, T.; TASSA, Y.; SILVER, D. ; WIERSTRA, D.. **Continuous control with deep reinforcement learning**. arXiv preprint arXiv:cs.LG/1509.02971, 2015.

[6] GARCÍA, R.; RUBIO, F. ; ORTEGA, M.. **Robust pid control of the quadrotor helicopter**. IFAC Proceedings Volumes, 45(3):229–234, 2012. 2nd IFAC Conference on Advances in PID Control.

[7] LAMBERT, N. O.; DREW, D. S.; YACONELLI, J.; LEVINE, S.; CALANDRA, R. ; PISTER, K. S. J.. **Low-level control of a quadrotor with deep model-based reinforcement learning**. IEEE Robotics and Automation Letters, 4(4):4224–4230, 2019.

[8] DYDEK, Z. T.; ANNASWAMY, A. M. ; LAVRETSKY, E.. **Adaptive control of quadrotor uavs: A design trade study with flight evaluations**. IEEE Transactions on Control Systems Technology, 21(4):1400–1406, 2013.

[9] OGATA, K.. **Modern Control Engineering Fourth Edition**. Prentice Hall, New Jersey, 2002.

[10] MONTAVON, G.; ORR, G. B. ; MÜLLER, K.-R.. **Neural Networks: Tricks of the Trade Second Edition**. Springer, 2012.

[11] GOODFELLOW, I.; BENGIO, Y. ; COURVILLE, A.. **Deep Learning**. MIT Press, 2016.

[12] FUJIMOTO, S.; VAN HOOF, H. ; MEGER, D.. **Addressing function approximation error in actor-critic methods**. In: Dy, J.; Krause, A., editors, PROCEEDINGS OF THE 35TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, volumen 80 de **Proceedings of Machine Learning Research**, p. 1587–1596. PMLR, 10–15 Jul 2018.

[13] HAARNOJA, T.; ZHOU, A.; ABBEEL, P. ; LEVINE, S.. **Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 1861–1870. PMLR, 2018.

[14] SILVER, D.; LEVER, G.; HEESS, N.; DEGRIS, T.; WIERSTRA, D. ; RIEDMILLER, M.. **Deterministic policy gradient algorithms**. In: Xing, E. P.; Jebara, T., editors, PROCEEDINGS OF THE 31ST INTERNATIONAL CONFERENCE ON MACHINE LEARNING, volumen 32 de **Proceedings of Machine Learning Research**, p. 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[15] FLOREANO, D.; WOOD, R. J.. **Science, technology and the future of small autonomous drones**. Nature International Journal of Science, 521:460—-466, 2015.

[16] BACHRACH, A.; PRENTICE, S.; HE, R.; HENRY, P.; HUANG, A. S.; KRAININ, M.; MATURANA, D.; FOX, D. ; ROY, N.. **Estimation, planning, and mapping for autonomous flight using an RGB-d camera in GPS-denied environments**. The International Journal of Robotics Research, 31(11):1320–1343, Sept. 2012.

[17] NETO, M. F. S.; EDUARDO, G. S.; SILVA, E. C. ; CAARLS, W.. **Computer vision based solutions for mav target detection and flight control**. In: 10TH INTERNATIONAL MICRO AIR VEHICLE COMPETITION AND CONFERENCE (IMAV2018), p. 309–314, 2018.

[18] AGGARWAL, C. C.. **Neural Networks and Deep Learning**. Springer International Publishing, 2018.

[19] CUI, Z.; HENG, L.; YEO, Y. C.; GEIGER, A.; POLLEFEYS, M. ; SATTLER, T.. **Real-time dense mapping for self-driving vehicles using fisheye cameras**. In: 2019 INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (ICRA), p. 6087–6093, 2019.

[20] KALLIOMÄKI, R.. **Real-time object detection for autonomous vehicles using deep learning**. Master's thesis, Uppsala University, Sweden, 2019.

[21] HIRZ, M.; WALZEL, B.. **Sensor and object recognition technologies for self-driving cars**. Computer-Aided Design and Applications, 15(4):501–508, 2018.

[22] WU, T.; TSENG, S.; LAI, C.; HO, C. ; LAI, Y.. **Navigating assistance system for quadcopter with deep reinforcement learning**. In: 2018 1ST INTERNATIONAL COGNITIVE CITIES CONFERENCE (IC3), p. 16–19, 2018.

[23] WALVEKAR, A.; GOEL, Y.; JAIN, A.; CHAKRABARTY, S. ; KUMAR, A.. **Vision based autonomous navigation of quadcopter using reinforcement learning**. In: 2019 IEEE 2ND INTERNATIONAL CONFERENCE ON AUTOMATION, ELECTRONICS AND ELECTRICAL ENGINEERING (AUTEEE), p. 160–165, 2019.

[24] ILG, E.; MAYER, N.; SAIKIA, T.; KEUPER, M.; DOSOVITSKIY, A. ; BROX, T.. **Flownet 2.0: Evolution of optical flow estimation with deep networks**. In: PROCEEDINGS OF THE IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), July 2017.

[25] CAI, S.; LIANG, J.; GAO, Q.; XU, C. ; WEI, R.. **Particle image velocimetry based on a deep learning motion estimator**. IEEE Transactions on Instrumentation and Measurement, 69(6):3538–3554, 2020.

[26] KOCH, W.; MANCUSO, R.; WEST, R. ; BESTAVROS, A.. **Reinforcement learning for uav attitude control**. ACM Transactions on Cyber-Physical Systems, 3(2):1–21, 2019.

[27] KONING, T.. **Low level quadcopter control using Reinforcement Learning** . Master's thesis, Delft University of Technology, the Netherlands, 2020.

[28] HWANGBO, J.; SA, I.; SIEGWART, R. ; HUTTER, M.. **Control of a quadrotor with reinforcement learning**. IEEE Robotics and Automation Letters, 2(4):2096–2103, Oct. 2017.

[29] SALIAN, I.. **Nvidia blog: Supervised vs. unsupervised learning**, Aug 2019. <https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/> Accessed: 03 Apr. 2021.

[30] BOTTOU, L.. **Online algorithms and stochastic approximations**. In: Saad, D., editor, ONLINE LEARNING AND NEURAL NETWORKS. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[31] O'SHEA, K.; NASH, R.. **An introduction to convolutional neural networks**. arXiv preprint arXiv:1511.08458, 2015.

[32] HE, K.; ZHANG, X.; REN, S. ; SUN, J.. **Deep residual learning for image recognition**. In: PROCEEDINGS OF THE IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION, p. 770–778, 2016.

[33] BELLMAN, R.. **A markovian decision process**. Indiana Univ. Math. J., 6:679–684, 1957.

[34] OPENAI. **Spinning up in deep rl**, 2018. <https://spinningup.openai.com> Accessed: 18 Jan. 2021.

[35] MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S. ; HASSABIS, D.. **Human-level control through deep reinforcement learning**. Nature, 518(7540):529–533, Feb. 2015.

[36] LIN, L.-J.. **Reinforcement Learning for Robots Using Neural Networks**. PhD thesis, USA, 1992. UMI Order No. GAX93-22750.

[37] HAARNOJA, T.; ZHOU, A.; HARTIKAINEN, K.; TUCKER, G.; HA, S.; TAN, J.; KUMAR, V.; ZHU, H.; GUPTA, A.; ABBEEL, P. ; LEVINE, S.. **Soft actor-critic algorithms and applications**. arXiv preprint arXiv:cs.LG/1812.05905, 2018.

[38] MOUTINHO, A.; AZINHEIRA, J. R.. **Simulação e Controlo de Drones**. Instituto Superior Técnico de Lisboa, May 2018.

[39] BEARD, R.. **Quadrotor dynamics and control rev 0.1**. Technical report, Ira A. Fulton College of Engineering and Technology, 2008. <http://hdl.lib.byu.edu/1877/624> Accessed: 12 Mar. 2021.

[40] STAPLES, G.. **Propeller static & dynamic thrust calculation**, 2014. <https://www.electricrcaircraftguy.com/2014/04/propeller-static-dynamic-thrust-equation-background.html> Accessed: 12 Mar. 2021.

[41] Cessna Aircraft Company, Wichita, KS. **Model 172 and Skyhawk Owner's Manual**, 1975.

[42] GIBIANSKY, A.. **Quadcopter dynamics, simulation, and control**, November 2012. <https://andrew.gibiansky.com/downloads/pdf/Quadcopter%20Dynamics,%20Simulation,%20and%20Control.pdf> Accessed: 12 Mar. 2021.

[43] BECKER-EHMCK, P.; KARL, M.; PETERS, J. ; VAN DER SMAGT, P.. **Learning to fly via deep model-based reinforcement learning**. arXiv preprint arXiv:cs.RO/2003.08876, 2020.

[44] KENDALL, A.; HAWKE, J.; JANZ, D.; MAZUR, P.; REDA, D.; ALLEN, J.-M.; LAM, V.-D.; BEWLEY, A. ; SHAH, A.. **Learning to drive in a day**. In: 2019 INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION (ICRA), p. 8248–8254. IEEE, 2019.

[45] SANG-YUN, S.; YONG-WON, K. ; YONG-GUK, K.. **Report for Game of Drones: A NeurIPS 2019 Competition**. 2019. <https://microsoft.github.io/AirSim-NeurIPS2019-Drone-Racing/_files/Sangyun.pdf> Accessed: 12 Mar. 2021.

[46] KINGMA, D. P.; WELLING, M.. **Auto-encoding variational bayes**. arXiv preprint arXiv:stat.ML/1312.6114, 2014.

[47] REZENDE, D. J.; MOHAMED, S. ; WIERSTRA, D.. **Stochastic backpropagation and approximate inference in deep generative models**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 1278–1286. PMLR, 2014.

[48] MOHTA, K.; WATTERSON, M.; MULGAONKAR, Y.; LIU, S.; QU, C.; MAKINENI, A.; SAULNIER, K.; SUN, K.; ZHU, A.; DELMERICO, J. ; ET AL.. **Fast, autonomous flight in gps-denied and cluttered environments**. Journal of Field Robotics, 35(1):101–120, Dec 2017.

[49] KAUFMANN, E.; GEHRIG, M.; FOEHN, P.; RANFTL, R.; DOSOVITSKIY, A.; KOLTUN, V. ; SCARAMUZZA, D.. **Beauty and the beast: Optimal methods meet learning for drone racing**. 2019 International Conference on Robotics and Automation (ICRA), p. 690–696, 2018.

[50] KAUFMANN, E.; LOQUERCIO, A.; RANFTL, R.; DOSOVITSKIY, A.; KOLTUN, V. ; SCARAMUZZA, D.. **Deep drone racing: Learning agile flight in dynamic environments**. In: CONFERENCE ON ROBOT LEARNING, p. 133–145. PMLR, 2018.

[51] LOQUERCIO, A.; MAQUEDA, A. I.; DEL-BLANCO, C. R. ; SCARAMUZZA, D.. **Dronet: Learning to fly by driving**. IEEE Robotics and Automation Letters, 3(2):1088–1095, 2018.

[52] SELVARAJU, R. R.; COGSWELL, M.; DAS, A.; VEDANTAM, R.; PARIKH, D. ; BATRA, D.. **Grad-cam: Visual explanations from deep networks via gradient-based localization**. International Journal of Computer Vision, 128(2):336–359, Oct 2019.

[53] PARK, D.; YU, H.; XUAN-MUNG, N.; LEE, J. ; HONG, S. K.. **Multicopter PID attitude controller gain auto-tuning through reinforcement learning neural networks**. In: PROCEEDINGS OF THE 2019 2ND INTERNATIONAL CONFERENCE ON CONTROL AND ROBOT TECHNOLOGY. ACM, Dec. 2019.

[54] SELMAN, B.; GOMES, C. P.. **Hill-climbing search**. Encyclopedia of cognitive science, 81:82, 2006.

[55] LIU FAN; ER MENG JOO. **Design for auto-tuning pid controller based on genetic algorithms**. In: 2009 4TH IEEE CONFERENCE ON INDUSTRIAL ELECTRONICS AND APPLICATIONS, p. 1924–1928, 2009.

[56] BELKHALE, S.; LI, R.; KAHN, G.; MCALLISTER, R.; CALANDRA, R. ; LEVINE, S.. **Model-based meta-reinforcement learning for flight with suspended payloads**. IEEE Robotics and Automation Letters, 6(2):1471–1478, 2021.

[57] PAN, S. J.; YANG, Q.. **A survey on transfer learning**. IEEE Transactions on knowledge and data engineering, 22(10):1345–1359, 2009.

[58] TUIA, D.; RATLE, F.; PACIFICI, F.; KANEVSKI, M. F. ; EMERY, W. J.. **Active learning methods for remote sensing image classification**. IEEE Transactions on Geoscience and Remote Sensing, 47(7):2218–2232, 2009.

[59] ENGEL, J.-M.; BABUŠKA, R.. **On-line reinforcement learning for nonlinear motion control: Quadratic and non-quadratic reward functions**. IFAC Proceedings Volumes, 47(3):7043–7048, 2014.

[60] MADAAN, R.; GYDE, N.; VEMPRALA, S.; BROWN, M.; NAGAMI, K.; TAUBNER, T.; CRISTOFALO, E.; SCARAMUZZA, D.; SCHWAGER, M. ; KAPOOR, A.. **Airsim drone racing lab**. In: NEURIPS 2019 COMPETITION AND DEMONSTRATION TRACK, p. 177–191. PMLR, 2020.

A
SAC implementation

Algorithm 3: Soft Actor-Critic

Initialize an empty replay buffer $\mathcal{R}$

Initialize the policy network with weights θ and two Q-networks with weights ϕ_1 and ϕ_2

Initialize two target Q-networks with weights $\phi_{\text{targ},1} \leftarrow \phi_1$ and $\phi_{\text{targ},2} \leftarrow \phi_2$

do

▷ Run an episode

for *t = 1,T* **do**

Observe s and sample an action $a \sim \pi_\theta(s)$

Input a to the quadrotor and observe s', $r(s, a, s')$

Store s, a, r, s' in $\mathcal{R}$

Break this loop if s' is terminal

end

▷ Perform N updates

for *i = 1,N*

Randomly sample a batch of transitions $\mathcal{D} \subset \mathcal{R}$

Perform one step ADAM from losses:

$$\mathcal{L}(\phi) = \frac{1}{|\mathcal{D}|} \sum_{(s,a,r,s') \in \mathcal{D}} \left(Q_{\phi_1} - y(r, s')\right)^2 + \left(Q_{\phi_2} - y(r, s')\right)^2,$$

where

$$y(r, s') \leftarrow \begin{cases} r, & \text{if } s' \text{ is terminal, otherwise:} \\ r + \gamma \left(\min_{i=1,2} Q_{\phi_{targ,i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right) \end{cases}$$

where $\tilde{a}' \sim \pi_\theta(\cdot|s')$

$$\mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(s,a,r,s')\in\mathcal{D}} - \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) + \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)$$

where $\tilde{a}_\theta(s) \sim \pi_\theta(\cdot|s)$

$$\mathcal{L}(\alpha) = \frac{1}{|\mathcal{D}|} \sum_{(s,a,r,s')\in\mathcal{D}} - \ln \alpha(\log \pi_\theta(\tilde{a}'|s) + h)$$

where $\tilde{a}' \sim \pi_\theta(\cdot|s')$
Update the target networks

$$\phi_{\text{targ},i} \leftarrow \tau\phi_i + (1-\tau)\phi_{\text{targ},i} \qquad \text{for } i = 1, 2$$

end

until *converged*;