POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Fully end-to-end deep learning policies for vision-based autonomous racing on ultra-low-power nano-drones

Supervisors
Prof. Daniele Jahier Pagliari
Prof. Alessio Burrello
Dr. Daniele Palossi
Elia Cereda
Beatrice Alessandra Motetti

Candidates
Florentin-Cristian Udrea

December 2024

Abstract

In recent years, drones have found applications in a variety of domains, such as aerial surveillance and rescue missions to precision agriculture and film-making. Advancements in drone miniaturization led to nano-drones: very compact drones with only 10 cm in diameter and a few tens of grams in weight, which have advantages, such as being highly maneuverable in confined areas and safe to operate around people, but also have limitations, such as battery lifetime lasting only a few minutes and a microcontroller unit (MCU) limited to under 100 mW of power, which restricts computational capacity.

Among possible tasks, over the last years autonomous drone racing (ADR) has become increasingly popular. In ADR competitions drones must autonomously navigate through gates and avoid obstacles at high speeds, thus requiring reactive perception and precise control. Because of these requirements, ADR competitions have become a proxy to improve autonomous drones' navigation capabilities, encouraging advancements in onboard perception and control algorithms. More recently, the research community started to put a lot of emphasis on fully end-to-end autonomous systems for ADR, in which a single deep learning system processes the sensor inputs and outputs the motor commands. While end-to-end policies became state-of-the-art for bigger drone systems, they have not been employed on nano-drones due to their computational constraints.

This thesis focuses on the Crazyflie 2.1, a nano-drone which, equipped with an ultra-low-power monochrome camera, state-estimation sensors, and a GAP8 MCU, can execute deep learning tasks onboard. The objective of the work is to develop a fully end-to-end vision-based deep learning policy in simulation, targeting the deployment on the Crazyflie 2.1 nano-drone.

The proposed method consists of multiple steps. First, we use a learning-by-cheating framework, in which a priviledged information policy (teacher policy) is used to teach a vision-based policy (student policy). Both the teacher and student policies are implemented as neural networks, enabling

them to learn complex behaviors through data-driven optimization. While the teacher policy can be trained directly with reinforcement learning (RL) due to its simpler state-based input space, the vision-based student policy has to be trained via imitation learning (IL), using the teacher policy as a dataset collector. Second, we leverage the dataset created in the previous step to apply neural architecture search (NAS) techniques in order to reduce the policy's computational cost and ensure its deployability. Finally, an RL fine-tuning through the asymmetric actor-critc framework is employed by using the pretrained post-NAS actor network and the teacher's pretrained critic network. This last step is essential to achieve a highly performing and stable vision-based student policy. In order to reduce the discrepancy between the simulator data and the real world data (known as 'reality gap') and to ensure deployability in unseen environments, multiple domain generalization techniques, such as visual domain randomization and pencil-filtering, were used during the training stages.

As a result, the teacher policy, taken as an upper bound in this work, it is able to finish successfully the tracks 99% of the time, but it requires priviledged information which is not obtainable in the real world. Our final policy, on the other hand, uses only information obtainable from the on-board sensors and it's able to complete successfully an unknown ADR track up to 70% of the time when in a already seen visual environment. Thanks to the NAS techniques applied, the final policy results deployable at 30Hz entirely onboard the GAP8 SoC, allowing it to process image frames in real-time at the native camera frame rate, requiring only 15M MAC, a 12x reduction w.r.t the pre-NAS model (183M MAC). The final student policy performances are obtainable thanks to the RL fine-tuning, which maximized the student network performances, going from a 3% success rate in known visual environment of the NAS-optimized network to the 70% cited previously. Thanks to the domain generalization techniques the final student policy is able to fly in an unknown track with a never-seen-before visual environment with a success rate of 51%, whereas without these techniques the agent would not be able to fly in an unknown visual environment.

Acknowledgements

This thesis marks the end of my five years long journey at the Politecnico di Torino. During this period of time, filled with ups and downs, many people supported me and this section of the thesis is dedicated to thank you all.

First and foremost, I want to give a heartfelt thank you to all my supervisors: Daniele Jahier Pagliari, Alessio Burrello, Daniele Palossi, Beatrice Alessandra Motetti and Elia Cereda. I want to thank them for the opportunity, their support and their guidance. I know it was as hard for them to deal with me, as it was for me to understand IsaacLab, so I am also really grateful for their patience. I'd also want to thank Lorenzo, Elia, Luca and all the IDSIA family for all the Mario Kart sessions and all the evenings at the foce, during my period abroad in Lugano.

A huge thank you goes to all my friends, both old and new, who have always supported me and with whom I've shared countless memories. Thank you to Vizzo who has been with me in this journey since day one. Thank you to Jack and Luca, for being my companions during the master degree. Thank you also to the Enri, Carlotta, Fede and Angelo, with who I have shared countless spritz o'clocks since the first year of the bachelor's. Thank you to Ilias who was always there for me when I was in need, even though we took separate paths. A special thank you goes to my girlfriend, Angi, who has been my no.1 supporter these last couple of years.

The biggest thank you of them all goes to my family. My mom and dad, who always supported me, were patient with me and pushed me to do better, and my little sister, who always knew how to cheer me up when I was stressed out. I hope I will continue to make you proud.

Finally, I think you, who are reading this thesis, probably didn't understand a thing, but that's okay. Thank you, too.

Florentin

Contents

List of Tables 9

List of Figures 10

1 Introduction 13

2 Background 19

2.1 Quadcopters . . . 20

2.1.1 Quadrotor Dynamics . . . 20

2.1.2 Bitcraze Crazyflie 2.1 . . . 24

2.1.3 PULP Architecture and GAP8 SoC . . . 25

2.2 Autonomous Drone Racing . . . 26

2.2.1 Classic Approach . . . 27

2.2.2 Learning-based Approaches . . . 29

2.2.3 End-to-End Learned Approaches . . . 32

2.3 Supervised Learning . . . 34

2.3.1 Deep Learning . . . 34

2.3.2 Neural Network Layers . . . 35

2.3.3 Domain Generalization . . . 38

2.4 Reinforcement Learning (RL) . . . 39

2.4.1 RL Theoretical Foundation . . . 39

2.4.2 Classic Model-Free Approaches . . . 43

2.4.3 Deep Reinforcement Learning . . . 45

2.4.4 Proximal Policy Optimization (PPO) . . . 47

2.4.5 Asymmetric Actor-Critic (AAC) . . . 48

2.5 Imitation Learning (IL) . . . 49

2.5.1 Behavioral Cloning (BC) . . . 49

2.5.2 Distribution Shift Problem . . . 50

2.5.3 Dataset Aggregation (DAgger) . . . 51

2.6 Simulation-based Robot Learning 52
2.6.1 WeBots Simulator 53
2.6.2 Nvidia IsaacSim Simulator 54
2.7 Neural Architecture Search (NAS) 55
2.7.1 Differentiable NAS (DNAS) 55
2.7.2 Types of DNAS 57
2.7.3 DNAS Training Procedure 58

3 Related Work **59**
3.1 Machine Learning in Autonomous Drone Racing 59
3.1.1 End-to-End Learning 59
3.1.2 Domain Generalization 61
3.2 TinyML on Ultra-Low Power SoCs 62
3.2.1 Nanodrones as Resource Constrained Application . . . 62

4 Methods **65**
4.1 Task Description and Challenges 65
4.2 Approach Overview 67
4.2.1 Simulator and Environment 67
4.2.2 Policy Development 69
4.3 Priviledged Teacher Agent Training 72
4.3.1 Observation and Action Spaces 72
4.3.2 Reward Function 73
4.3.3 Teacher Neural Network Architecture 80
4.4 Vision-based Student Agent 81
4.4.1 Partial-Information Observation Space 81
4.4.2 Student Neural Network Architecture 82
4.4.3 Student Architecture Computational Constraints . . . 83
4.4.4 Seed Architecture Analysis 85
4.4.5 Seed Network Training through IL 88
4.4.6 Domain Generalization 88
4.4.7 Student Neural Architecture Optimization 92
4.4.8 Asymmetric Actor-Critic RL Fine-tuning 93

5 Experiments and Results **95**
5.1 Experimental Setup 95
5.2 Policy Performance Assessment Method 97
5.3 Priviledged Teacher Agent Results 98
5.3.1 Teacher training through RL 98

5.4 Vision-based Student Agent 102
5.4.1 Seed Architecture Exploration 102
5.4.2 Seed Network Training thorugh IL 107
5.4.3 Student Network Architecture Optimization 112
5.4.4 Asymmetric Actor-Critic RL Fine-tuning 115
5.5 Comparison of Pipeline Steps 118

6 Conclusions and Future Works 119

Bibliography 121

List of Tables

4.1 Computation of the operations per clock (N) given the neural network and GAP8 configurations reported in [33], using equation 4.10. 83
4.2 Analysis of tiny-CNN architectures' performance on GAP8 SoC 86
4.3 Module Hyperparameters and Choices 87
5.1 Python package versions. 96
5.2 Reward Scales for Different Components 98
5.3 Closed-loop performance of the teacher policy. 99
5.4 Seed architecture exploration results: average results grouped by history size 104
5.5 Seed architecture exploration results: average results grouped by projector size 105
5.6 Seed architecture exploration results: average results grouped by visual feature encoder 106
5.7 Open loop performance of the seed network policy on the test set. 108
5.8 Closed-loop performance of the seed network policy. 108
5.9 Seed architecture MACs vs NAS architecture MACs in detail by module. 113
5.10 Open loop performance of the NAS policy compared with the Seed policy open loop performance. 113
5.11 Closed-loop performance of the NAS policy. 113
5.12 Closed-loop performance summary of RL fine-tuning policy. . 116
5.13 Closed-loop performance summary of the policies trained. . . . 118

List of Figures

1.1 Number of related publications per year, as evidenced by a google-scholar search for “autonomous drone racing” [16]. . . . 14

1.2 The Crazyflie 2.1 equipped with the AI Deck. Source: `https://www.bitcraze.io/` 15

2.1 Drone coordinate system: the X axis is the ’forward’ direction, the Y axis the lateral direction and the Z axis the vertical direction. The rotational angles of the drone coordinate system influence its translational motion in the following way: the roll angle ϕ influence the motion on the Y axis, while the pitch angle θ influences the motion on the X axis. Finally the yaw angle ψ influence the orientation. [3] 22

2.2 Crazyflie 2.1 along with a categorization of UAVs by weight. . 24

2.3 GAP8 System-on-a-Chip Architecture [33] 25

2.4 A long exposure photograph of autonomous drone racing competition taken from [21]. In blue the autonomous drone, while in blue Alex Vanover, the 2019 Drone Racing League world champion 26

2.5 A classic pipeline architecture for an autonomous system programmed using model-based approaches 27

2.6 Learned control pipeline. 29

2.7 Learned control pipeline. 30

2.8 Learned perception and planning pipeline. 30

2.9 Learned perception and planning pipeline. 31

2.10 Modular end-to-end learned pipeline 32

2.11 Fully end-to-end learned pipeline 33

2.12 Deep learning taxonomy. [5] 34

2.13 Illustration of one feed forward layer. [5] 35

2.14 The overall architecture of the CNN includes an input layer, multiple alternating convolution and max-pooling layers, one fully-connected layer and one classification layer. [5] 36

2.15 Visual representation of the TCN architecture. 37
2.16 A visual explanation to domain generalization. 38
2.17 Visual explanation of the covariate shift and error accumulation problem. [13] 50
2.18 IsaacLab multiple randomized environments [28] 52
2.19 An example of DNAS method (DARTS [25]): (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities. 56
2.20 Path-based DNAS illustration, as implemented in PLiNIO Python library [19]. 57
4.1 Example of track generated with the method described. 68
4.2 Proposed method: leverage the imitation learning supervised nature to introduce a neural architecture search step, in order to respect the computational constraints 69
4.3 The progress reward in 2D, assuming the gate is in direction (0,1) 76
4.4 Safety reward illustration, in which the gate's position is in (0,0) and the gate plane is positioned on the x-axis. 77
4.5 Perception reward illustration. 78
4.6 Teacher network illustration, with the correct input, hidden and output layer sizes. 80
4.7 The student network architecture used. 82
4.8 An illustration of the student network architecture in a deployment scenario. 84
4.9 Some of the visual environments the student agent sees during training as 360 views. 89
4.10 Visual explanation of the pencil filter idea presented in [34]: closing the visual sim-to-real gap by narrowing the discrepancy between different brightness images and blurred images. 90
4.11 Different images. The first row is the RBG image, the second row is the respective grayscale image, the third row is the respective pencil-filtered image. 91

4.12 The same image (the first column is the original), with different random augmentations. The first row is the RBG image, the second row is the respective grayscale image, the third row is the respective pencil-filtered image. 91
5.1 Teacher policy's episodic total reward progression during the first 100k rollouts of the training. 100
5.2 Teacher policy's episod length progression during the first 100k rollouts of the training. 100
5.3 Teacher policy's crash reward progression during the first 100k rollouts of the training. 101
5.4 Teacher policy's smoothness reward progression during the entire training (500'000 rollouts). 101
5.5 The seed architecture hyperparameter analysis pareto-front, colored by history size 104
5.6 The seed architecture hyperparameter analysis pareto-front, colored by projector output size 105
5.7 The seed architecture hyperparameter analysis pareto-front, colored by visual feature encoder 106
5.8 Avg. episodic reward progress for DAgger policies. 109
5.9 Avg. percentage of episodic reward progress for DAgger policies. 109
5.10 Avg. successful episodes for DAgger policies. 110
5.11 Action command comparison between teacher commands and student commands in an example run, where only student's commands were applied. Actions are rescaled in the range (-1,1) for better comparison. 111
5.12 NAS Loss progression during training. 114
5.13 NAS Cost progression during training. 114
5.14 RL fine-tuning episodic total reward progress during the training. 117
5.15 RL fine-tuning episode length progress during the training. . . 117

Chapter 1

Introduction

Today, drones are employed across various fields such as industrial inspection, search and rescue operations or surveillance. Mastery in control and navigation through complex environments is necessary for these kind of tasks and requires years of training for expert pilots to achieve. Drone racing competitions, in which pilots navigate their drones through a series of gates as quickly as possible, offer a safe and challenging environment to improve and compare pilots' drone maneuverability skills. In these competitions pilots operate their drones using First Person View (FPV) information, which provides real-time footage from a camera mounted on the front of the drone, allowing them to drive the drone via remote radio controllers. Drones able to autonomously perform these maneuvers, however, greatly enhance the potential in real-world applications.

Researchers use **autonomous drone racing** (ADR) competitions to measure their autonomous drone systems progress. Similarly to the traditional drone racing competitions, the objective is to navigate autonomously through a series of gates as quickly as possible using the on-board algorithms. The autonomous drones' ability to rapidly navigate complex and unknown environments without years of experience and training recommend them as an attractive alternative to the human pilots, which is why there is increasing interest in this field as reported in [16] and shown in fig. 1.1. To encourage the research, projects like the European Research Council's AgileFlight [12] and many competitions like the IROS Autonomous Drone Racing Series [29], and AlphaPilot AI Drone Racing Innovation Challenge where a 1 million dollar grand prize was awarded to the winning team [4] have been launched in the last few years.

Because of the high speeds at which the autonomous drones are flying

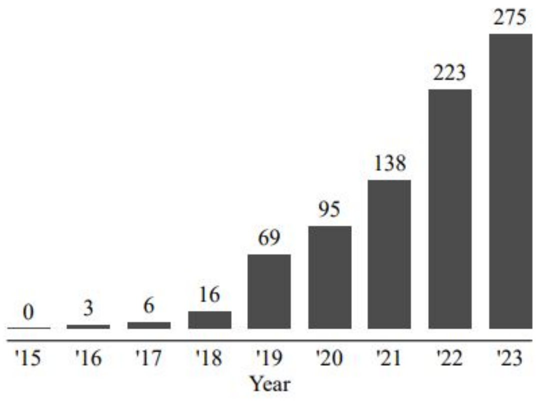


Figure 1.1: Number of related publications per year, as evidenced by a google-scholar search for “autonomous drone racing” [16].

in this context, the algorithms proposed by researches must be robust to sensor noise, efficient and provide optimal decision and control behaviors in real time [16]. Given these requirements, recent research found **fully end-to-end vision-based deep learning** systems to be a good candidate for their good generalization capabilities, robustness to sensor noise and fast adjustment to environmental changes [14, 50]. These kind of systems use one single neural network (often called also policy) which takes sensor inputs and computes directly motor actions. The sensor inputs often come mainly from a camera, which is why they are referred to as ’vision-based’. They are particularly interesting to researchers for their ability to handle both high-dimensional inputs (such as images) and low-dimensional inputs (such as robot states) and their ease of implementation and deployment. A key advantage of these methods is their simplicity in using one single technique in order to produce motor commands directly from sensor inputs, which results lower latency and no compounding errors when compared to traditional approaches. While traditional approaches often consist in multiple different techniques which rarely make efficient use of specific hardware, the fully end-to-end deep learning policies consists of only one algorithm which can take advantage of hardware accelerators, such as GPUs, which speed up the inference. These characteristics led the fully end-to-end deep learning policies to become state-of-the-art when it comes to gate-navigation ADR competitions.

Even though state-of-the-art neural networks algorithms require hardware accelerators in order to run efficiently, this kind of hardware is too energy demanding to be featured on palm-sized nano-drones, which have a battery

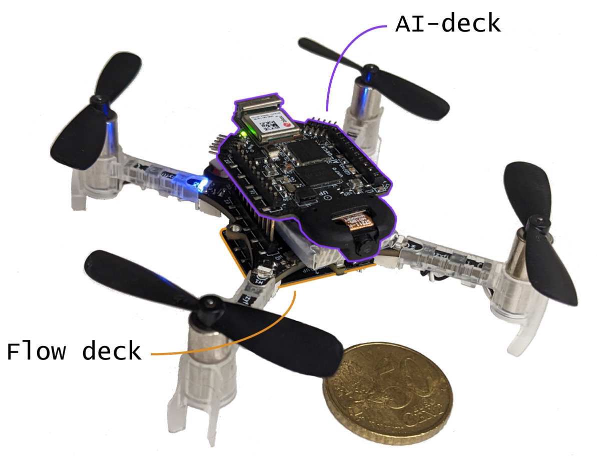


Figure 1.2: The Crazyflie 2.1 equipped with the AI Deck. Source: `https://www.bitcraze.io/`

that allows a time of flight of only a few minutes [2] and a microcontroller unit (MCU) limited to under 100 mW of power. The nano-drones are interesting from a real world applications point of view as they are able to navigate in very narrow spaces and they are very safe to humans. The **Bitcraze Crazyflie 2.1**, an ultra-low-power nano-drone which weighs 27 grams and has a time of flight of only 7 minutes, tries to overcome this limitation by allowing the addition of the **GAP8 System-on-a-Chip**, an energy-efficient IoT application processor that enables the onboard execution of neural networks. Even though this addition improves on the limitations described previously, the constrained memory and the low frequency continue to be an issue, so the neural network algorithms deployed on such devices have to be subject to further architecture improvements to ensure the real-time operability and to fully exploit the hardware at hand.

While the deployment of the end-to-end deep learning policies on resource constrained nano-drones needs particular attention to their computational cost due to the lack of on-board GPU-like hardware accelerators, the training is done off-board and can fully exploit the GPU parallelization capability. This is particularly beneficial as deep learning systems are known for their **high sample complexity**, meaning that a substantial amount of data is required for effective training and model development. In robotics, collecting large amounts of real-world data is often discouraged, as it is both time consuming and can lead to damaging the real-world robot. Recently **simulators** have been used as an efficient way to generate large volumes of data in a controlled, cost-effective manner. Even if state-of-the-art simulators provide high quality rendering, there is still a mismatch between the input distribution that the neural network sees in the simulator at training time and the input distribution in the real world. This mismatch is often referred to as "**reality-gap**" and it leads to poor performance of the policy in the real world scenario w.r.t. the performance in simulation.

While separately all the topics previously discussed are fairly popular research topics, there is, to the best of our knowledge, no line of work that addresses the task of building end-to-end deep learning policies for vision-based autonomous racing on ultra-low-power nano-drones. This thesis aim is to investigate the intersection of these research fields.

The **contributions** of this thesis are:

- An in depth analysis of the state-of-the-art neural network architectures used for resource constrained vision-based tasks and/or autonomous drone racing;
- Analyze state-of-the-art techniques for narrowing the reality gap, for a direct deployment in unknown environments;
- Propose a novel method of developing fully end-to-end deep learning vision-based policies for resource limited deployment targets, which in this case is the Bitcraze Crazyflie 2.1;

Given the multiple constraints that the problem poses, the final policy cannot be trained in a single step, thus the proposed method consists of four steps. The *first step* is to train a priviledged information policy, which uses ground truth information given by the simulator in order to produce optimal actions for the drone. This first policy is not deployable on the Crazyflie 2.1 as it uses information not obtainable with the drone's on-board sensors. However, given its ability to produce optimal actions thanks to the priviledged information that it gets as input, it is used in the *second step* to collect a dataset in which in which the partial information (here, visual information) data is associated with the teacher's optimal outputs. This dataset is then used to train a vision-based network. Because priviledged information policy collects the data it is called a teacher policy, while the trained policy is called the student policy. By leveraging the teacher-student framework, a vision-based policy which is able to navigate the race track is trained. The vision-based student policy, however, has a computational cost which is not low enough to be deployed in real-time on the Crazyflie 2.1, thus, in the *third step*, neural architecture search (NAS) techniques are employed in order to reduce the student policy's computational cost and ensure its deployability. These techniques automatically explore multiple sub-networks, starting from the student network, and chose the one that better optimizes the computational cost vs task performance trade-off w.r.t. the computational constraints that the target deployment system imposes. Finally, in the *fourth step*, a reinforcement learning fine-tuning on the pretrained post-NAS policy is done. Reinforcement learning optimizes a neural network through the direct exploration of the simulated environment. This allows the student policy to get more robust and maximize the performances in deployment. In order to reduce the discrepancy between the simulator data and the real world data and to ensure deployability in unseen environments, multiple domain generalization techniques were used during all the training stages.

The teacher policy is taken as an upper bound in this work and it is able to finish successfully the tracks 99% of the time, but it requires priviledged information which is not obtainable in the real world. Our final policy, on the other hand, uses only information obtainable from the on-board sensors, such as the camera and the accelerometer, and is able to complete successfully an unknown ADR track up to 70% of the time when in a already seen visual environment. Thanks to the NAS techniques applied, the final policy results deployable at 30Hz entirely onboard the GAP8 SoC, allowing it to process image frames in real-time at the native camera frame rate, requiring only 15M MAC, a 12x reduction w.r.t the pre-NAS model (183M MAC). The student policy performances are obtainable thanks to the RL fine-tuning, which maximized the student network performances, going from a 3% success rate in known visual environment of the NAS-optimized network to the 70% cited previously. Special attention has been put also in minimizing the reality gap, enhancing its generalization to new visual environments capabilities. Thanks to these techniques the final student policy is able to fly in an unknown track with a never-seen-before visual environment with a success rate of 51%, whereas without these techniques the agent would not be able to fly in an unknown visual environment.

In the following chapters will get in more depth about the methodology and the results. The organization of the thesis is the following: chapter 2 an overview of the topics and the main algorithms used is presented. Chapter 3 presents an analysis of the state-of-the-art works for each topic of interest of this thesis, briefly analyzing strengths and weaknesses, and explaining the difference with the project at hand. Next, chapter 4 describes what is the method used in order to produce the results, while chapter 5 describes the experiments and the results obtained. Finally in the last chapter (6) an overview of the work is given, as well as future directions and works that can be done starting from this.

Chapter 2

Background

The following sections are devoted to introduction of the theoretical concepts that are needed for this work. In section 2.1, we describe the quadcopter system, analyzing its dynamics model; then we present the actual quadcopter platform that we will be working with. In section 2.2, an overview of the main approaches in autonomous drone racing is provided, with particular focus on the end-to-end approaches, as it is the main focus of this work. In the sections after, an introduction to the main techniques used for end-to-end racing in the context of autonomous drone racing are presented. In particular in section 2.3 we give a brief introduction to Supervised Learning (SL), Deep Learning (DL) and the various neural network architectures used in this work. Section 2.4 provides a theoretical background of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL), while also explaining the main concepts and algorithms used. Finally in section 2.5 we introduce Imitation Learning (IL), describing its advantages and limitations, along with the algorithms used in this work. The last two sections of this chapter focus on more practical aspects of the work. Section 2.6 explains what are the advantages and disadvantages of robot learning in simulation and explores some of the most used simulators for autonomous drone racing. Lastly section 2.7 focuses on neural network optimization techniques, as this work focuses on resource constrained quadcopter platforms, which can only host lightweight neural networks.

2.1 Quadcopters

2.1.1 Quadrotor Dynamics

Quadcopters, or quadrotors, are unmanned aerial vehicles (UAVs) that utilize four rotors to generate lift and control motion. Unlike traditional helicopters, which typically use a single large rotor and tail rotor for control, quadcopters rely on differential thrust between four independent motors. This design provides agility, stability, and simplicity in control. The dynamics of the motors are inherently nonlinear and coupled, leading to the need for sophisticated modeling and control techniques.

Flight Dynamics

The motion of a quadcopter is governed by Newton-Euler equations, that describe how the forces and moments acting on the vehicle affect its translational and rotational motion. The thrust generated by each rotor contributes to the overall lift that counters gravity and allows the quadcopter to hover or ascend. By varying the speeds of individual motors, the quadcopter can control its roll, pitch, and yaw, enabling it to tilt, rotate, and change direction [22]. This enables six degrees of freedom (DOF): three are translational (up/down, forward/backward, left/right) and three rotational (roll, pitch, yaw).

The configuration of the motors in either an "X" or "+" arrangement allows for the control of attitude (orientation) and position through the manipulation of the rotor speeds. For instance:

- Roll, the angle which describes the rotation around the front-to-back axis (longitudinal axis), is controlled by increasing the speed of the two motors on one side and decreasing the speed on the opposite side.
- Pitch, the angle which describes the rotation around the side-to-side axis (lateral axis), is similarly controlled by varying the speeds of the front and rear motors.
- Yaw, the angle which describes the rotation around the vertical axis, is achieved by adjusting the differential speeds of clockwise and counter-clockwise rotating motors, exploiting the reaction torque generated by the rotors.

This dynamic interaction between thrust and torque leads to a highly maneuverable system, but also one that requires real-time feedback and control to maintain stability during flight.

Quadcopter as an underactuated system

A quadcopter is classified as an *underactuated system*, meaning that it has fewer independent control inputs than the degrees of freedom (DOF) to be controlled. Specifically, a quadcopter operates with four independent control inputs (the thrust generated by each of its four rotors) but has six degrees of freedom: three translational DOF (along the x, y, and z axes) and three rotational DOF (roll, pitch, and yaw).

The main challenge of underactuation is that not all degrees of freedom can be controlled directly. In the case of a quadcopter:

- The four control inputs are the thrusts generated by each rotor, which can control the overall thrust and the rotational motions (roll, pitch, and yaw).
- The translational motion along the x and y axes, however, cannot be controlled directly. Instead, they are controlled indirectly through the tilting of the vehicle, which changes the direction of the total thrust vector.

Since controlling each rotor individually for every movement would make the control problem highly complex, it is easier to specify higher level controls, such as desired setpoints for thrust, roll, pitch, and yaw, rather than specifying the thrust for each motor independently [22]. More in detail, the four parameters controlled are:

- **Thrust:** The total upward force to counteract gravity and control vertical movement.
- **Roll and Pitch:** The angles that determine the tilting of the quadcopter, which indirectly control the motion in the x and y directions.
- **Yaw:** The rotation around the vertical axis, controlling the orientation of the quadcopter.

This abstraction simplifies the control problem by separating the dynamics into more manageable terms [22]. This separation is more clear in Fig. 2.1 which illustrates the coordinate system of the Crazyflie 2.1.

Advantages of Using Angle Setpoints

By using desired setpoints for thrust, roll, pitch, and yaw, the control problem is significantly simplified [22]:

- The flight controller only needs to manage four high-level parameters, rather than four independent motor thrusts.
- The coupling between rotational and translational dynamics is handled indirectly, making it easier to maintain stable flight.
- Complex flight maneuvers can be achieved through coordinated changes in roll, pitch, and yaw, rather than requiring fine adjustments to individual motors.

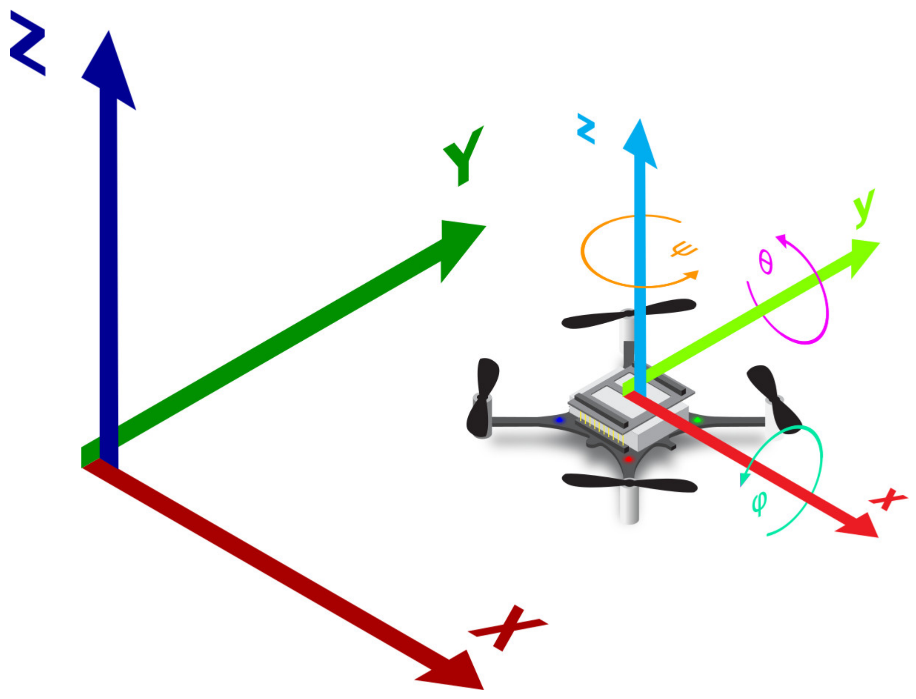


Figure 2.1: Drone coordinate system: the X axis is the 'forward' direction, the Y axis the lateral direction and the Z axis the vertical direction. The rotational angles of the drone coordinate system influence its translational motion in the following way: the roll angle ϕ influence the motion on the Y axis, while the pitch angle θ influences the motion on the X axis. Finally the yaw angle ψ influence the orientation. [3]

Motor Mixing Algorithm

To be able to use the desired thrust, roll, pitch, and yaw as setpoints, these values must still be translated into specific thrust commands for each of the four motors. This process is known as the *motor mixing algorithm* [6]. The goal of the motor mixing algorithm is to compute individual motor thrusts (u_1, u_2, u_3, u_4) based on the total thrust and the torques required for roll, pitch, and yaw.

For a quadcopter in an "X" configuration, the relationship between the control inputs (thrust, roll, pitch, and yaw) and the individual motor thrusts is typically expressed as:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & l & 0 & -l \\ -l & 0 & l & 0 \\ c & -c & c & -c \end{bmatrix} \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}$$

Where:

- u_1, u_2, u_3, u_4 are the individual motor thrusts.
- T is the total thrust.
- $\tau_\phi, \tau_\theta, \tau_\psi$ are the torques required for roll, pitch, and yaw, respectively.
- l is the distance from the center of the quadcopter to each motor (lever arm).
- c is a constant related to the direction and magnitude of yaw torque generated by each rotor (which depends on the motor spin direction).

The motor mixing algorithm works by assigning specific motor thrust values that balance the total desired thrust and generate the necessary torques to achieve the desired roll, pitch, and yaw.

Once the thrust, roll, pitch, and yaw setpoints are calculated by the flight controller, these values are fed into the motor mixing algorithm to determine the exact motor speeds required to achieve the desired motion. This approach allows for smooth and coordinated control without needing to directly calculate individual motor thrusts for each flight maneuver [6].

2.1.2 Bitcraze Crazyflie 2.1

Released in 2019, the **Crazyflie 2.1** [2] is a light (27g) nano-quadcopter of 10 cm diameter with up to 7 minutes of flight using stock batteries. Through the addition of expansion decks, its hardware can be enhanced in terms of sensing, positioning, and vision. Its basic hardware components are:

- the **Micro Controller Unit STM32F405**, which handles the low-level and high-level controls [44].
- the **nRF51822**, another MCU designated for radio and power management [41].

Two widely used expansion decks are:

- the **Flow deck**, for visual odometry navigation. It allows the drone to detect the motion in any direction thanks to:
  - the **VL53L1x Time of Flight (ToF)**, a laser-based sensor which measures the distance from the ground (the altitude) [1].
  - the **PMW3901 optical flow sensor**, which measures the displacement in the x, y direction as long as the altitude is at least 80mm [18].
- the **AI deck**, which comes with the Himax HM01B0, an Ultra-Low Power (ULP) 320x320 grayscale mono-camera with 30 FPS acquisition rate [47], and the GAP8 System-On-Chip (SoC) [46], which enables AI-based applications to run onboard thanks to the Parallel Ultra-Low Power (PULP) paradigm.

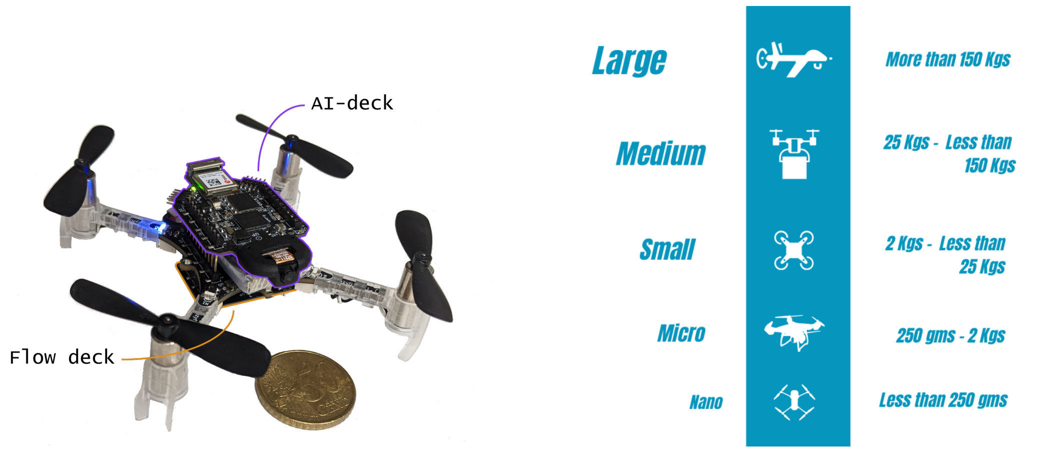


Figure 2.2: Crazyflie 2.1 along with a categorization of UAVs by weight.

2.1.3 PULP Architecture and GAP8 SoC

Being the "brain" of Crazyflie's AI deck, the **GAP8 System-on-a-Chip (SoC)** is an IoT application processor that enables the onboard execution of neural networks thanks to the **Parallel Ultra-Low Power (PULP)** paradigm.

Produced by GreenWaves Technologies, this ultra-low power processor features a total of **nine identical RISC-V cores**. One is called Fabric Controller (FC) and is devoted to being the main core in the Micro Controller Unit (MCU), controlling all the GAP8 operations. It enables and dispatches the workload to the Cluster (CL). The remaining eight cores compose the CL, the parallel general-purpose accelerator, which can be programmed to compute efficiently highly parallel workloads. The CL receives from the FC digital signal processing (DSP) workload in order to speed up the execution by exploiting the cluster parallelism and low-latency L1 shared memory..

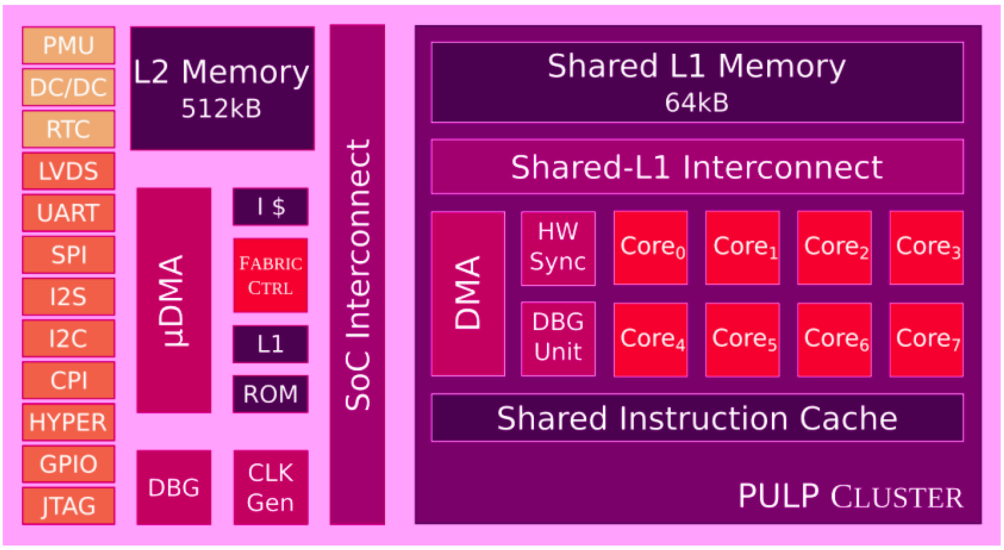


Figure 2.3: GAP8 System-on-a-Chip Architecture [33]

2.2 Autonomous Drone Racing

Autonomous Drone Racing (ADR) systems add nothing new to the classic pipeline of mobile robotic autonomous systems. They need to *perceive* the environment, to *plan* their actions and finally to calculate the *control actions* that better follow the trajectory planned.

Initially these were three different, sequential, model-based algorithm blocks, but since the rise of neural networks many researchers tried to approximate these blocks (either separately, to ensure modularity, or altoghether, reducing the compouding error) with neural-networks architectures.

Following the taxonomy that [16] proposes, in the next sections we give an overview of the most well known methods, starting by describing more in detail the classic approach in section 2.2.1. Then, in the following section 2.2.2 we will be looking at different ways to approximate these blocks with neural networks. More specifically, we will talk about learned perception, learned perception and planning, learned planning and controls. Lastly in section 2.2.3 we will dive deeper in the end-to-end approaches.

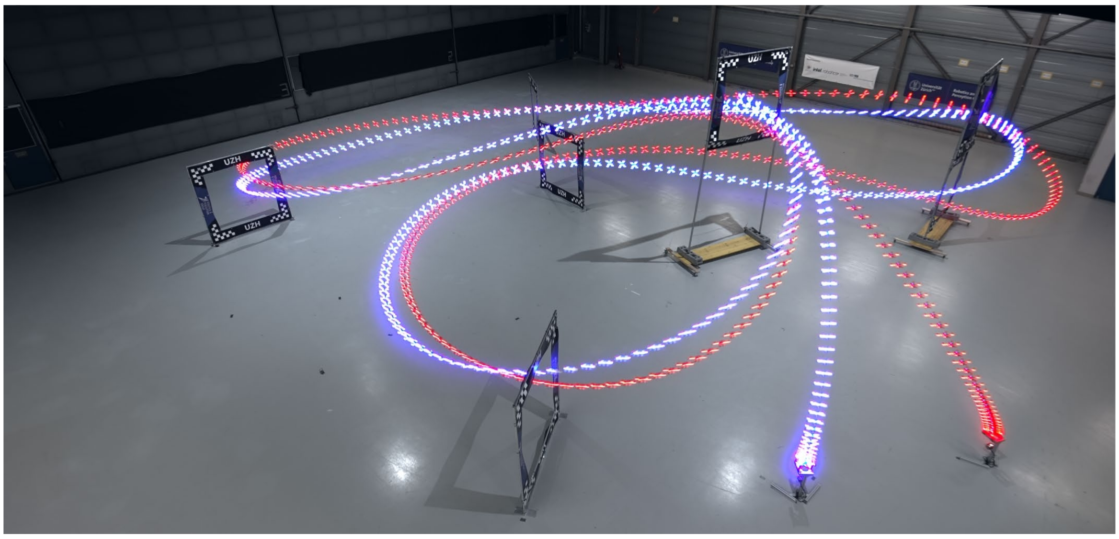


Figure 2.4: A long exposure photograph of autonomous drone racing competition taken from [21]. In blue the autonomous drone, while in blue Alex Vanover, the 2019 Drone Racing League world champion

2.2.1 Classic Approach

Classic autonomous systems are commonly split in three model-based sequential blocks: *perception, planning* and *control* as shown in Fig 2.5. Each of these blocks plays a fundamental role in the autonomous systems' ability to perform the task at hand. The following subsections aim at giving a more in depth explanation of each block and an overview of the main techniques used.

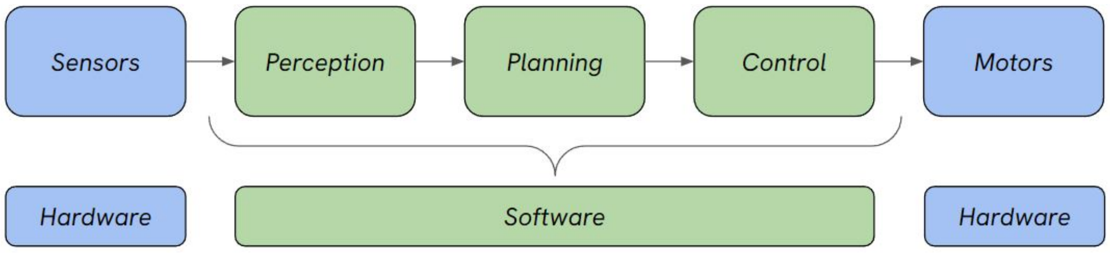


Figure 2.5: A classic pipeline architecture for an autonomous system programmed using model-based approaches

Perception

The perception block is responsible of estimating the vehicle state and perceiving the environment using onboard sensors, and of processing it in a useful and interpretable way.

In ADR, visual-inertial odometry (VIO) is commonly used for state estimation due to its low cost and lightweight advantages. VIO combines camera and inertial measurement unit (IMU) data to estimate the drone's position, orientation, and velocity. While IMUs provide fast motion updates, they suffer from errors like drift and misalignment over time. Cameras, while offering detailed environmental data, operate at a lower rate and are sensitive to poor lighting, low texture, and motion blur. Despite these limitations, the combination of IMU and camera data makes VIO the standard method for state estimation in ADR.

Planning

The planning block interprets the processed data given by the perception block and outputs a trajectory to be followed by the agent (the trajectory is based on the model used for the agent, thus it can be 3d spatial coordinates, but it can also include angular position, velocity, angular velocity, acceleration and so on).

In ADR, after obtaining a state estimate, the next step in drone navigation is planning a feasible, time-optimal trajectory that adheres to the drone's physical limits and environmental constraints. This process involves predicting the drone's future states to minimize lap time while avoiding crashes. Planning is often divided into two tasks: path planning and trajectory planning. Path planning finds a geometric path from the start to the goal, avoiding obstacles, while trajectory planning refines this path by allocating time or ensuring a collision-free flight. Some methods rely solely on trajectory planning, assuming no collisions, while others find collision-free trajectories directly or focus only on the geometric path for drone control without time allocation.

Control

The control block is responsible for executing the planned trajectory by adjusting the system's actuators, ensuring that the drone follows the desired path with precision and stability.

In ADR, control techniques are used to translate the high-level trajectory into low-level commands that dictate the drone's motor speeds and orientations. The controller typically uses the state-estimation feedback from the perception block to continuously adjust its actions and compensate for disturbances or deviations from the intended trajectory. Classical control methods, such as Proportional-Integral-Derivative (PID) controllers, are often employed due to their simplicity and robustness in stabilizing the drone. More advanced model-based controllers, like Model Predictive Control (MPC), can also be used to optimize the control commands by predicting future states and accounting for system constraints. Both approaches ensure the drone's smooth flight, keeping it within its operational limits while reacting dynamically to changes in the environment or the system itself.

2.2.2 Learning-based Approaches

In this section, we explore various learning-based approaches for drone racing, which replace the planner, controller, and/or perception stack with neural networks. These methods have gained considerable popularity in recent years due to their ability to handle both high-dimensional inputs (e.g., images) and low-dimensional inputs (e.g., states), their strong representational power, and the ease of implementation on hardware.

A key advantage of learning-based approaches is their lower computational demand compared to traditional methods, potentially enabling low-latency re-planning and control. Furthermore, they exhibit greater robustness to system latencies and sensor noise, as these factors can be identified on physical drones and incorporated into the training environments.

In the following paragraphs, we discuss some of the most popular approaches for learning-based autonomous systems.

Learned Perception

For learned perception modules, the goal of the network is to use images from an RGB, depth, or event camera to detect landmarks within the environment and output useful representations such as waypoints, or the location of gates on the track. The information gathered from this processing step is usually fused with other sensors' data through Extended Kalman Filter (EKF) or other traditional algorithms. These algorithms allow to reduce the uncertainty of the data and to produce more reliable estimates of the drone's state.

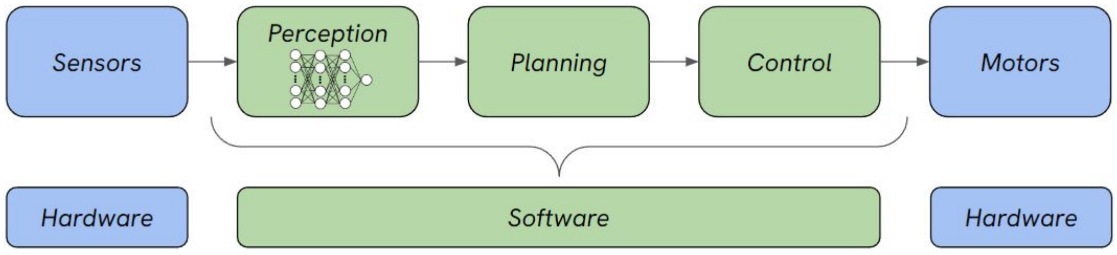


Figure 2.6: Learned control pipeline.

Learned Controls

Choosing data-driven control methods, such as reinforcement learning or imitation learning, addresses many of the limitations found in traditional model-based controllers by learning effective controllers directly from experience. Difficult and abstract finetuning (such as in PID controllers) or being limited by the model of the system (such as in MPC controllers) are issues which don't affect learning-based controllers. By training neural networks via reinforcement learning or imitation learning, complex hyperparameter tuning is not needed and, since they are model-free methods, they don't need an explicit model implementation.

However, unlike traditional methods, it can be challenging to guarantee robustness with learning-based controllers. While they may show superior performance in simulation, their application in the real world is often limited by the difficulty in analyzing and ensuring the controller's stability properties.

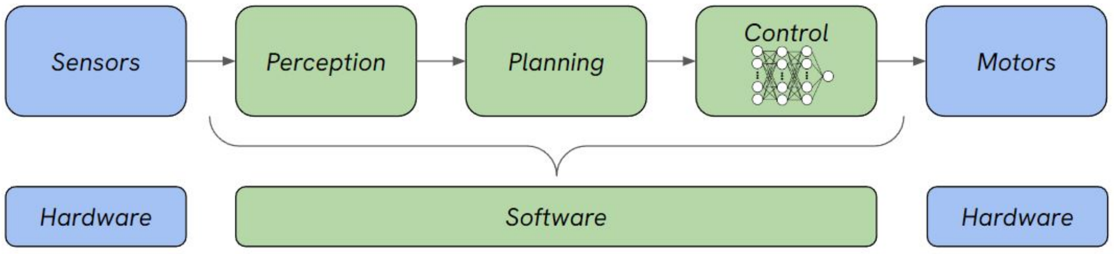


Figure 2.7: Learned control pipeline.

Learned Perception and Planning

A tightly-coupled, learning-based, perception and planning block has numerous advantages. As a first advantage, a complex, computationally heavy mapping algorithm is no longer needed, as an explicit map is not strictly necessary anymore. Furthermore, by leveraging large amounts of data, the planning becomes robust against noise in perception or dynamics.

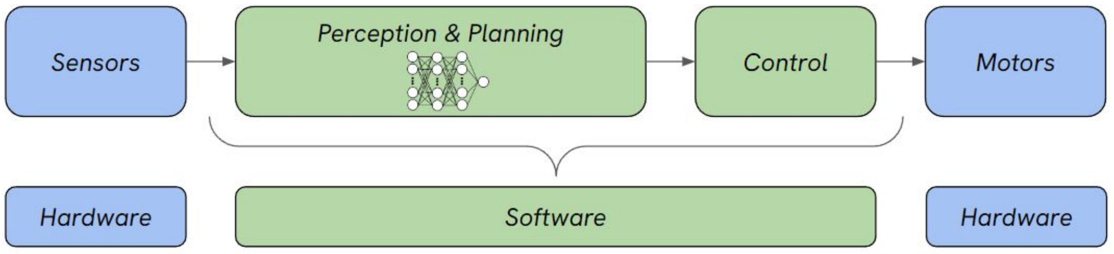


Figure 2.8: Learned perception and planning pipeline.

Learned Planning and Controls

This paradigm of learned planning and control is to produce the control command directly from state inputs without requiring a high-level trajectory planner. This paradigm makes use of traditional perception methods of sensor data processing and sensor fusion to extract data representations, which are then given as input to the planning-control learning-based module.

A first approach to this paradigm is to substitute the full planning-control module with a learning-based algorithm, such as reinforcement learning or imitation learning. Major advantages of the reinforcement-learning-based method are its capability to handle relevant track changes and the scalability to tackle large-scale random track layouts while retaining computational efficiency. The learned policy solves the planning and control problem simultaneously, forgoing the need for explicit trajectory planning.

A second approach to this paradigm is to exploit the benefits of model-based and learning-based approaches using differentiable optimizers which leverage differentiability through controllers.

Even though this paradigm does solve the problem of having an intermediate representation of the path, it still suffers from some of the limitations of traditional planning methods, such as needing a globally-consistent state estimation.

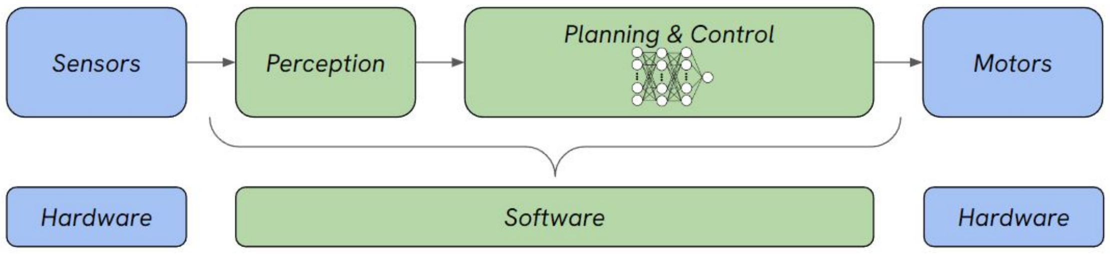


Figure 2.9: Learned perception and planning pipeline.

2.2.3 End-to-End Learned Approaches

End-to-end learning has recently gained popularity for tasks that need quick perception and decision-making, with autonomous drone racing becoming a key testing ground for these methods. Human pilots, who control drones based solely on sensor feedback, can rapidly interpret their surroundings and react to changes, easily navigating in complex environment. Recreating this level of control and adaptability in autonomous drones is challenging, but end-to-end learning aims at archiving such objective. It enables systems to turn raw sensory data directly into control commands without relying on traditional, manually designed algorithms: a crucial advantage in drone racing, where every fraction of a second matters.

In this section, we explore two key approaches that aim to emulate human navigation through learning-based methods: the modular end-to-end approach and the fully end-to-end approach [16].

Modular end-to-end learned approach

This first end-to-end paradigm builds upon the core principle of maintaining distinct but interdependent modules for perception, planning, and control. It swaps each model-based module of the classic pipeline with its own learning-based module. The separation of each block approach leads to flexibility and modularity, allowing for individual modifications to the neural network within each block that do not structurally affect the whole system.

This methodology allows for independent training of each block to output 'classic pipeline'-like outputs, promoting interpretability. The modularity also allows for the the incorporation of supplementary information to the input to the following block, such as (partial) results of computationally cheap classic algorithms, with ease.

Nevertheless the division in separate modules leads to compounding errors, which impact negatively the performance when flying at high speeds.

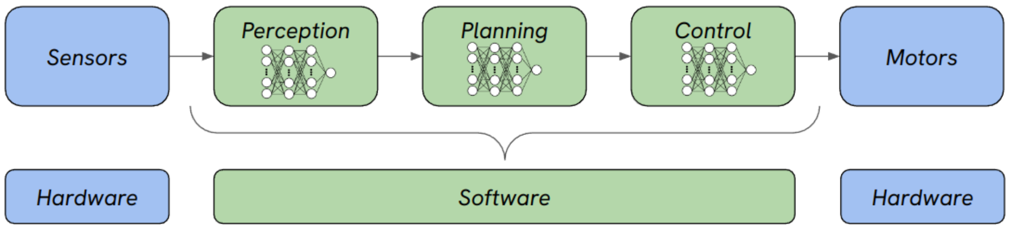


Figure 2.10: Modular end-to-end learned pipeline

Fully end-to-end learned approach

In the fully end-to-end approach, a single neural network is used to directly map raw sensory input, such as images from an onboard camera, to control actions for the drone. This approach bypasses the need for distinct perception, planning, and control modules, leveraging a deep neural network to autonomously derive navigation strategies based solely on sensory data. Training a fully end-to-end model involves exposing it to a variety of racing scenarios and allowing it to learn the optimal control strategies based on observed inputs and desired outputs.

This method solves the previous' compounding error problem, while also relying on a simpler design. Because of the latter, it is easier to allocate more development time towards complex training methods, potentially archiving superior adaptability and generalization. Depending on the deployment tools used, a simpler design also mean reduced latency in the decision making: a crucial advantage in fast-paced environments like drone racing.

On the other side, fully end-to-end approaches lack the interpretability that the modular approaches provide, making it difficult to debug and fine tune. Furthermore it may require a vast amount of diversified data in order to capture the complexity of different racing scenarios. The absence of distinct modules means the model must learn every aspect of the racing task simultaneously, demanding extensive datasets and potentially increasing training costs and time.

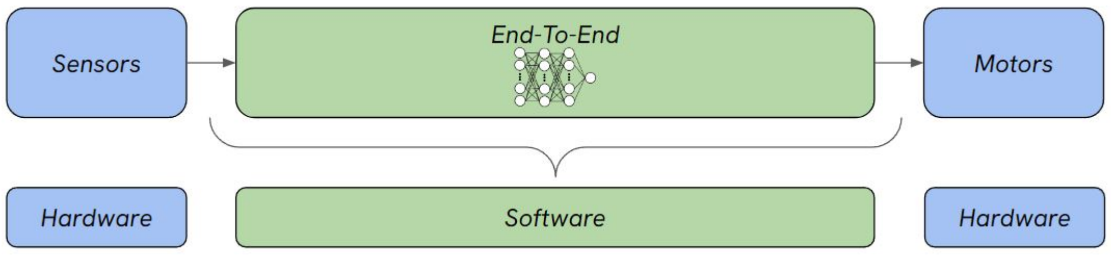


Figure 2.11: Fully end-to-end learned pipeline

2.3 Supervised Learning

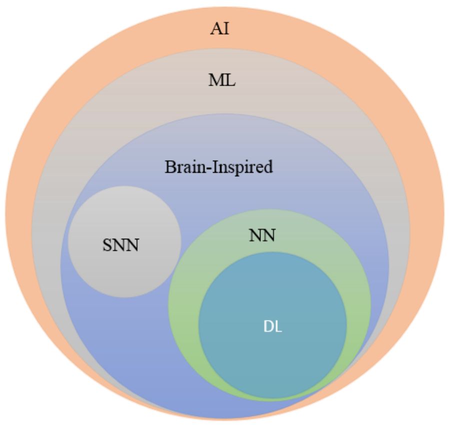
Supervised Learning is a fundamental ML technique extensively used in autonomous drones for tasks like object detection, classification, and semantic segmentation. In this context, a model is trained on a labeled dataset where the input data, such as images or sensor readings, are paired with corresponding labels, like object categories or navigation commands. The goal is to learn a function $f : X \to Y$ that maps inputs X (e.g., drone sensor data) to outputs Y (e.g., classification labels or control actions), based on a training set $\{(x_i, y_i)\}_{i=1}^{N}$, where $x_i \in X$ and $y_i \in Y$.

The learning objective is to find the model parameters θ that minimize a loss function $L(f_\theta(x_i), y_i)$, which quantifies the discrepancy between the predicted outputs and the actual labels. This can be formulated as:

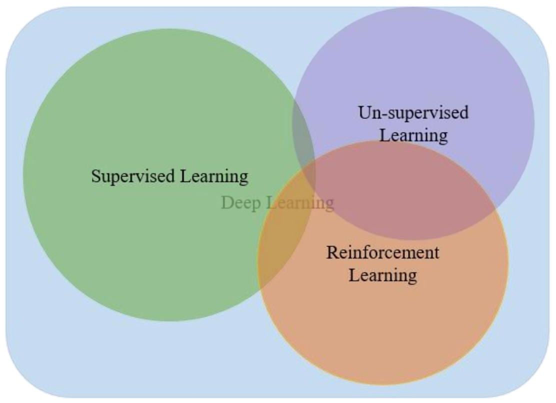
$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(f_\theta(x_i), y_i).$$

2.3.1 Deep Learning

Deep Learning is a subset of machine learning techniques that leverages neural networks with multiple layers to learn hierarchical representations of data. These deep neural networks can automatically extract features from raw data and are powerful tools for a wide range of applications. The mathematical formulation of different types of neural networks highlights their strengths and limitations compared to other architectures.



(a) A taxonomy of learning algorithms.



(b) Subdivisions of deep learning

Figure 2.12: Deep learning taxonomy. [5]

2.3.2 Neural Network Layers

Feed-forward Neural Networks (FNN)

Feed-forward Neural Networks (FNNs, also known as Artificial Neural Networks ANN or Fully-Connected Neural Networks (FC)) are the simplest form of artificial neural networks. They consist of layers of neurons where the data flows in one direction, from input to output, without any cycles or loops.

Consider an FNN with L layers. Each layer l consists of n_l neurons, and the input to the network is a vector $\mathbf{x} \in R^d$. The output of layer is computed as:

$$\mathbf{a}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}), \tag{2.1}$$

where:

- $\mathbf{a}^{(l)} \in R^{n_l}$ is the activation vector of layer l,
- $\mathbf{W}^{(l)} \in R^{n_l \times n_{l-1}}$ is the weight matrix connecting layer $l-1$ to layer l,
- $\mathbf{b}^{(l)} \in R^{n_l}$ is the bias vector for layer l,
- $\sigma(\cdot)$ is the activation function (e.g., ReLU, sigmoid, or tanh).

The final output $\hat{\mathbf{y}}$ of the network is obtained from the last layer. The network is trained by minimizing a loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, where $\mathbf{y}$ is the true label.

Compared to Convolutional Neural Networks (CNNs) and Temporal Convolutional Networks (TCNs), FNNs lack the specialized mechanisms to handle spatial and temporal dependencies effectively.

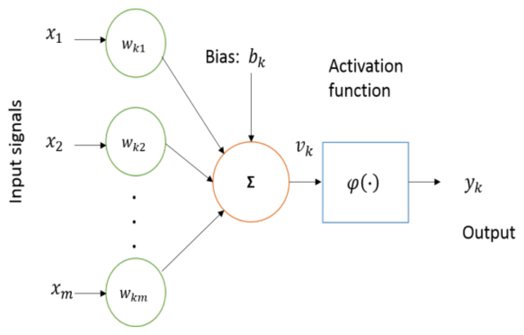


Figure 2.13: Illustration of one feed forward layer. [5]

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are designed to process data with a grid-like topology, such as images. They use convolutional layers to automatically learn spatial hierarchies of features.

A CNN applies a convolutional filter $\mathbf{K} \in R^{h \times w}$ to the input $\mathbf{I} \in R^{H \times W}$ to produce a feature map. For a given position (i, j) in the output feature map, the convolution operation is defined as:

$$(\mathbf{I} * \mathbf{K})(i, j) = \sum_{m=0}^{h-1} \sum_{n=0}^{w-1} \mathbf{I}(i + m, j + n) \cdot \mathbf{K}(m, n). \tag{2.2}$$

The output is then passed through an activation function $\sigma(\cdot)$, such as ReLU, and typically followed by pooling layers to reduce dimensionality while capturing dominant features and they may be also followed by batch normalization layers, that learn the new features' distribution and normalize them in order to ease the learning of the following layers.

Compared to FNNs, CNNs better capture spatial dependencies due to their convolutional layers and pooling operations. However, they are less suited for temporal sequences, where RNNs or TCNs might be preferred.

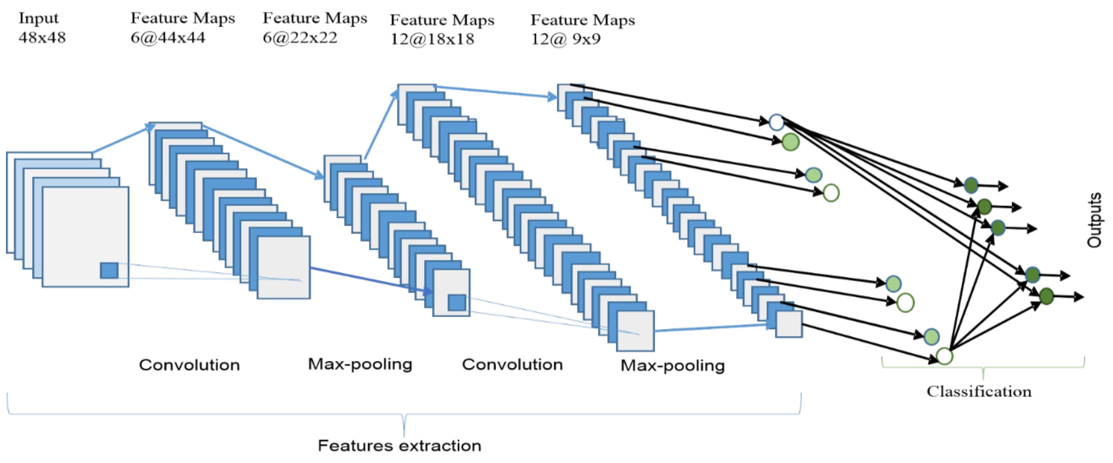


Figure 2.14: The overall architecture of the CNN includes an input layer, multiple alternating convolution and max-pooling layers, one fully-connected layer and one classification layer. [5]

Temporal Convolutional Networks (TCN)

Temporal Convolutional Networks (TCNs) are designed to handle sequential data by applying convolutional operations along the temporal dimension. They leverage dilated convolutions to capture long-range dependencies efficiently.

In a TCN, a dilated convolution operation is defined as:

$$(\mathbf{X} * \mathbf{K}_d)(t) = \sum_{k=0}^{K-1} \mathbf{X}(t - k \cdot d) \cdot \mathbf{K}(k), \tag{2.3}$$

where $\mathbf{X}$ is the input sequence, $\mathbf{K}_d$ is the dilated kernel with dilation factor d, and K is the kernel size. The dilation allows the network to have a wider receptive field without increasing the number of parameters.

TCNs offer a powerful alternative to RNNs by leveraging convolutions for sequential data, allowing for parallel processing and better handling of long-range dependencies.

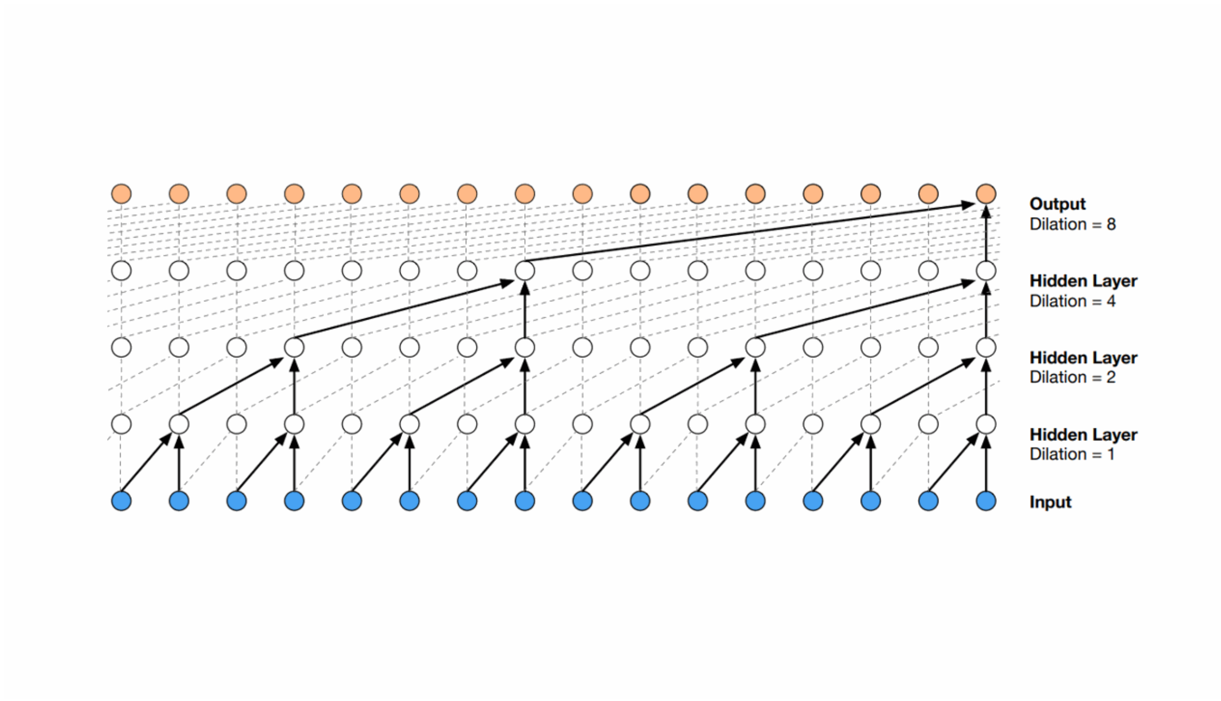


Figure 2.15: Visual representation of the TCN architecture.

2.3.3 Domain Generalization

In domain generalization within supervised deep learning, the goal is to train a model on data from multiple source domains to generalize well to an unseen target domain. Given source domains $\mathcal{D}_s = \{(\mathbf{x}_i^s, y_i^s)\}_{i=1}^{N_s}$ for $s = 1, \dots, S$, each with distribution $P_s(\mathbf{x}, y)$, we aim to learn a model that minimizes the expected risk over an unknown target distribution $P_t(\mathbf{x}, y)$. This process is shown in fig. 2.16.

Domain-invariance can be directly encouraged by ensuring that the distributions of the extracted features are similar across source domains:

$$P(g_\theta(\mathbf{x}^1)) \approx P(g_\theta(\mathbf{x}^2)) \approx \cdots \approx P(g_\theta(\mathbf{x}^S))$$

To improve generalization, **data augmentation** techniques can be applied to simulate variations that might occur in the target domain. This encourages the model to learn features that are robust across a broader range of inputs, indirectly aiding domain-invariance. By increasing the diversity of source data, these augmentations expose the model to a wider range of scenarios, indirectly preparing it to handle potential shifts in the unseen target domain.

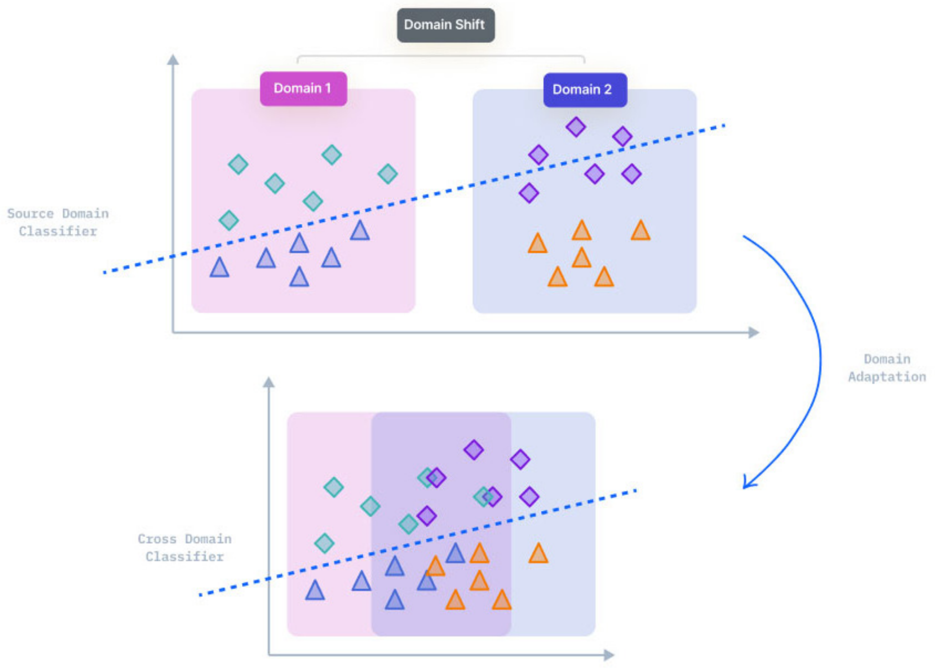


Figure 2.16: A visual explanation to domain generalization.

2.4 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a subfield of machine learning focused on training agents to make sequences of decisions by interacting with an environment, with the goal of maximizing a cumulative reward signal. Unlike supervised learning, where a model learns from a labeled dataset, RL is characterized by an agent learning through trial and error, using feedback from its own actions and experiences. The following subsections aim at giving a base understanding of RL following [45] and then going deeper on algorithms or techniques relative to RL that are useful for this thesis.

2.4.1 RL Theoretical Foundation

Markov Decision Process (MDP)

A fundamental concept in reinforcement learning is the *Markov Decision Process* (MDP), which provides a mathematical framework for modeling decision making in environments where outcomes are only partially under the control of a decision-maker. An MDP is defined by a tuple (S, A, P, R, γ), where:

- S is a finite set of states, representing all possible situations the agent can encounter.
- A is a finite set of actions, representing all possible decisions the agent can make.
- $P : S \times A \times S \to [0, 1]$ is the state transition probability function, where $P(s'|s, a)$ represents the probability of transitioning from state s to state s' after taking action a.
- $R : S \times A \to R$ is the reward function, where $R(s, a)$ gives the immediate reward received after taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor, which balances the trade-off between immediate and future rewards, by determining the present value of future rewards.

The *Markov property* implies that the future state s' depends only on the current state s and action a, and not on the sequence of events that preceded it. This property simplifies the modeling of decision-making processes and is crucial for developing efficient algorithms.

Policy

A policy $\pi : S \to A$ is a strategy used by the agent to determine actions based on states. It can be deterministic, $\pi(s) = a$, or stochastic, $\pi(a|s)$, where the probability of selecting action a in state s is given by $\pi(a|s)$. The policy is central to the agent's behavior as it maps the states to actions, dictating the course of actions for a given environment.

The goal of the agent in an RL setting is to learn a policy π, which is a mapping from states to actions that maximizes the expected cumulative reward. The expected cumulative reward, also known as the *return*, is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.4}$$

where G_t is the return at time step t, and $\gamma \in [0, 1]$ is the discount factor.

State-Value Function

The state-value function (or simpler, value function) $V(s)$ represents the expected cumulative reward that an agent can achieve from a state s under a given policy π. Formally, it is defined as:

$$V^{\pi}(s) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s\right].$$

The value function provides an estimate of the desirability of states, guiding the agent towards higher rewards.

Action-Value Function (Q-Function)

The action-value function (or Q-function) $Q(s, a)$, represents the expected cumulative reward of taking action a in state s and thereafter following a policy π. It is defined as:

$$Q^{\pi}(s, a) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s, a_0 = a\right].$$

The Q-function is fundamental in RL as it forms the basis of Q-learning, an important algorithm in the field.

Relationship between Value Function and Q-Function

The value function $V(s)$ and the Q-function $Q(s,a)$ are closely related concepts in reinforcement learning, both aimed at estimating expected rewards but from slightly different perspectives. The value function $V(s)$ gives the expected cumulative reward of starting from state s and following a specific policy π, aggregating the returns across all actions weighted by the policy's probabilities. In contrast, the Q-function $Q(s,a)$ provides a more granular view by estimating the expected reward starting from state s, taking a specific action a, and thereafter following policy π.

The two functions are linked by the equation $V(s) = \sum_{a\in A} \pi(a|s)Q(s,a)$, which shows that the value of a state under a policy can be derived from the Q-values by averaging over all possible actions according to the policy. This relationship is fundamental, as it allows an agent to determine optimal actions by comparing Q-values across actions within each state, a principle central to many reinforcement learning algorithms.

Bellman Equations

The Bellman equations provide recursive definitions of value functions, capturing the relationship between a state (or state-action pair), immediate rewards, and the value of subsequent states. These equations form the basis for many RL algorithms and offer a mechanism to iteratively compute the optimal value of each state. For any policy π, the Bellman equation for the state-value function $V^\pi(s)$ is defined as:

$$V^\pi(s) = \sum_{a\in A} \pi(a|s) \sum_{s'\in S} P(s'|s,a) \left[R(s,a) + \gamma V^\pi(s')\right],$$

Similarly, the Bellman equation for the action-value function, or Q-function, $Q^\pi(s,a)$ is:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'\in S} P(s'|s,a) \sum_{a'\in A} \pi(a'|s') Q^\pi(s',a').$$

The idea of the Bellman equation is that instead of calculating each value as the sum of the expected return, which is a long process, we calculate the value as the sum of immediate reward and the discounted value of the state that follows. These recursive relationships enable dynamic programming techniques such as policy iteration and value iteration.

Optimality and the Bellman Optimality Equations

The Bellman optimality equations define the maximum expected cumulative reward that can be obtained from any state by following the optimal policy π^*. For the optimal value function $V^*(s)$, the Bellman optimality equation is:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \left[R(s,a) + \gamma V^*(s') \right].$$

Similarly, for the optimal Q-function $Q^*(s,a)$, it is given by:

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

These equations represent a way to iteratively update the value of each state or state-action pair by maximizing over all possible actions. Solving the Bellman optimality equations yields the optimal policy, as an optimal policy always takes the action with the highest Q-value:

$$\pi^*(a|s) = \arg\max_a Q^*(s,a)$$

Policy Iteration and Value Iteration

Policy Iteration is an algorithm for finding the optimal policy by alternating between policy evaluation and policy improvement. The process can be summarized as:

1. **Policy Evaluation:** Given a policy π, compute the value function $V^\pi(s)$ for each state s by solving the Bellman equation for that policy, either exactly or approximately.

2. **Policy Improvement:** Use the computed value function to improve the policy by choosing the action that maximizes the expected return, updating π to a better policy.

These steps are repeated until the policy no longer changes, at which point the policy and value function are optimal.

Value Iteration, on the other hand, combines policy evaluation and improvement into a single step. Starting with an initial value function, it iteratively applies the Bellman optimality equation, updating the value of each state directly. This process continues until the value function converges to the optimal value function V^*.

2.4.2 Classic Model-Free Approaches

Classical model-free approaches in reinforcement learning estimate optimal policies directly from experience, without requiring a model of the environment's dynamics, such as transition probabilities. Unlike methods presented in the previous subsection, which simulate future states (thus called model-based, as they use a model of the problem in order to simulate future states), model-free techniques adjust value estimates and policies based solely on observed rewards. Key methods, including Monte Carlo Learning, Temporal Difference (TD) Learning, and Q-Learning, use different strategies to refine these estimates through direct interaction with the environment.

Monte Carlo Learning

Monte Carlo Learning is a class of algorithms that estimates value functions by averaging returns over complete runs, called episodes. In this approach, value estimates are only updated after observing the full sequence of rewards from an episode, making it well-suited for environments where episodes naturally end. Given a policy π, the Monte Carlo estimate of the state-value function $V(s)$ for a state s is obtained by averaging the returns following each visit to s under that policy. Formally, for a state s, the value estimate is given by:

$$V(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_i,$$

where $N(s)$ is the number of times state s has been visited, and G_i is the return (cumulative reward) following the i-th visit to s. This formulation provides an unbiased estimate of the expected return since it relies on actual observed outcomes over entire episodes.

Temporal Difference Learning

Temporal Difference (TD) Learning is a class of algorithms that updates value estimates based on observed rewards and the difference between subsequent estimates. Unlike Monte Carlo methods, which require full episodes to update values, TD learning can update values on a step-by-step basis, making it suitable for online learning.

The TD(0) update rule for the state-value function is:

$$V(s) \leftarrow V(s) + \alpha \left[R(s,a) + \gamma V(s') - V(s)\right],$$

where α is the learning rate and $R(s,a) + \gamma V(s') - V(s)$ is the temporal difference error, measuring the difference between predicted and actual returns.

Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that seeks to learn the optimal action-value function, $Q^*(s,a)$. Unlike Monte Carlo methods, Q-Learning can update value estimates at each step, making it suitable for online learning. The update rule for Q-Learning is based on the Bellman optimality equation and adjusts the Q-value for a given state-action pair (s,a) as follows:

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

where α is the learning rate, $R(s,a)$ is the immediate reward, and γ is the discount factor that balances immediate and future rewards. The term $\max_{a'} Q(s',a')$ represents the estimated maximum future reward achievable from the next state s', guiding Q-Learning towards the optimal policy regardless of the agent's actual behavior policy.

Because Q-Learning directly approximates the optimal Q-values, it can converge to the optimal policy as long as all state-action pairs are visited infinitely often and the learning rate decays appropriately. This ability to find an optimal policy without following it during learning distinguishes Q-Learning as a flexible algorithm well-suited for dynamic and stochastic environments.

2.4.3 Deep Reinforcement Learning

As we consider environments with high-dimensional or continuous state and action spaces, such as those encountered in robotics or video games, the limitations of classical RL algorithms become apparent. Traditional methods like tabular Q-learning and value iteration struggle to handle the vast state-action spaces due to their reliance on exhaustive search and storage. *Deep Reinforcement Learning* (Deep RL) addresses these limitations by leveraging the power of deep neural networks to approximate value functions, policies, or both. This allows RL algorithms to scale to complex environments by generalizing across large or continuous spaces, effectively capturing the essential features of states and actions. By replacing tables with neural networks, Deep RL methods can learn directly from high-dimensional sensory inputs, such as images or raw sensor data, leading to breakthroughs in fields where traditional RL methods were previously infeasible.

The next sections will delve into Deep RL techniques, starting with value-based methods, such as *Deep Q-Networks* (DQN), and then exploring policy-based methods and advanced algorithms like *Proximal Policy Optimization* (PPO), which combine the strengths of both value and policy-based approaches.

Value-Based Deep RL Methods

Value-based methods in Deep RL focus on learning an approximation of the optimal action-value function $Q^*(s, a)$. The most notable algorithm in this category is *Deep Q-Networks* (DQN), which was the first to successfully apply deep learning to RL problems.

DQN approximates the Q-value function using a neural network parameterized by θ, denoted as $Q(s, a; \theta)$. The objective of DQN is to minimize the following loss function, which is derived from the Bellman equation:

$$L(\theta) = E_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right], \tag{2.5}$$

where D is a replay buffer storing past experiences, and θ^- are the parameters of a target network that are periodically updated to stabilize training.

Because value-based methods approximate the action-value function $Q^*(s, a)$, their output has to be enumerable. In practice the Q-Network will return the expected Q-function for each action, given the current state. This makes value-based methods hardly usable for continuous action spaces.

Policy-Based Deep RL Methods

Policy-based methods directly parameterize the policy $\pi_\theta(a|s)$ using a neural network and optimize the parameters θ to maximize the expected return. These methods have several advantages over value-based methods, including better handling of high-dimensional or continuous action spaces and providing naturally stochastic policies.

The *REINFORCE* algorithm is one of the simplest policy gradient methods. The objective function $J(\theta)$ to be maximized is the expected cumulative reward:

$$J(\theta) = E_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right]. \tag{2.6}$$

The gradient of this objective function, according to the policy gradient theorem, is:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]. \tag{2.7}$$

Actor-Critic Methods

Actor-critic methods combine the advantages of value-based and policy-based methods. The *actor* is responsible for selecting actions based on a policy $\pi_\theta(a|s)$, while the *critic* estimates the value function $V^\pi(s)$ or the action-value function $Q^\pi(s,a)$ to guide the actor's updates. This combination allows for more stable and efficient learning.

The *Advantage Actor-Critic* (A2C) algorithm improves the basic actor-critic framework by using the *advantage function* $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ to reduce the variance of policy updates. The objective for the actor in A2C is:

$$L^{A2C}(\theta) = E_{(s,a)\sim\pi_\theta}\left[\log \pi_\theta(a|s)\hat{A}(s,a)\right], \tag{2.8}$$

where $\hat{A}(s,a)$ is an estimator of the advantage function.

2.4.4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [40] is an advanced policy gradient method that improves the stability and efficiency of training in reinforcement learning. Traditional policy gradient methods like Vanilla Policy Gradient (VPG) can suffer from instability due to large policy updates, leading to performance degradation or even divergence. To address this, PPO restricts how much the policy can change during each update, providing a more stable learning process.

PPO Objective and Clipped Surrogate Function

The key idea behind PPO is to update the policy in a way that ensures the new policy does not deviate too much from the old one. This is achieved using a clipped surrogate objective function:

$$L^{CLIP}(\theta) = E_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right], \tag{2.9}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, $\hat{A}_t$ is the advantage estimate, and ϵ is a hyperparameter that controls the allowable change in policy.

The clipping mechanism in the objective function limits the size of the policy update by ensuring that $r_t(\theta)$ stays within $[1-\epsilon, 1+\epsilon]$. This prevents overly large updates, leading to more stable training. By taking the minimum of the clipped and unclipped objective, PPO avoids pushing the policy too far in one update, even when the advantage is high, which stabilizes learning.

Advantages of PPO

PPO offers key advantages over traditional methods:

- **Stability**: The clipped objective prevents large, destabilizing policy changes, making training smoother.
- **Simplicity**: Unlike more complex methods like TRPO, PPO is straightforward to implement and computationally efficient while achieving similar performance.
- **Sample Efficiency**: PPO strikes a good balance between exploration and controlled policy updates, improving sample efficiency compared to naive policy gradient methods.

2.4.5 Asymmetric Actor-Critic (AAC)

Asymmetric actor-critic (AAC) [35] methods are a type of RL architecture where the actor and critic networks receive different inputs. The actor is responsible for choosing actions based on observations, while the critic evaluates the actions by estimating the value function or the advantage function. The key idea is that the critic can access privileged information not available to the actor, such as true state values, which improves learning efficiency.

In environments involving visual tasks, such as visual navigation or object manipulation, the actor typically operates using high-dimensional sensory inputs like images. In contrast, the critic can receive privileged, structured information, such as object positions or velocity, leading to more accurate evaluation and faster learning.

Asymmetric Structure

In a standard actor-critic architecture, both the actor and the critic typically share the same observation o_t, representing the environment at time t. The actor outputs a policy $\pi_\theta(a_t|o_t)$ that selects actions a_t, and the critic evaluates the quality of these actions using a value function $V(o_t)$ or a Q-function $Q(o_t, a_t)$.

However, in an asymmetric setup, the critic receives additional privileged state information s_t, which is not available to the actor. This privileged information allows the critic to compute a more accurate value estimate $Q(s_t, a_t)$, which would not be possible using only the observation o_t.

$$\pi_\theta(a_t|o_t) \quad \text{(Actor policy based on observations)}$$

$$Q_w(s_t, a_t) \quad \text{(Critic's value function based on privileged state information)}$$

The key benefit of this asymmetric structure is that the critic can learn a better approximation of the value function $Q(s_t, a_t)$, as it is based on more precise state information s_t rather than raw visual inputs. This helps in reducing the variance in the value estimates, leading to more stable policy updates for the actor, which operates only on o_t. As a result, the actor can improve its policy faster, despite having access to limited information. The critic's ability to exploit privileged data also provides better learning efficiency, and by reducing variance, the overall training process becomes more stable, allowing for smoother policy learning in challenging visual tasks.

2.5 Imitation Learning (IL)

Imitation Learning (IL) is a framework in machine learning where an agent learns to perform tasks by observing and imitating the behavior of an expert. This approach is particularly appealing in robotics because it bypasses the need for manually designing complex reward functions, which can be difficult and time-consuming. Instead, an agent can learn directly from demonstrations provided by a human expert or another agent. IL is especially useful in scenarios where defining a reward function is challenging, but demonstrations are readily available.

Imitation Learning methods can be broadly categorized into two main approaches: *Behavioral Cloning* and *Inverse Reinforcement Learning* [32]. In this section, we focus on the former, which directly learns a policy from the expert's actions, and a popular algorithm within that domain, *Dataset Aggregation (DAgger).*

2.5.1 Behavioral Cloning (BC)

Behavioral Cloning (BC) is one of the simplest forms of Imitation Learning. It involves training a policy to mimic the actions of an expert directly from state-action pairs collected from expert demonstrations. Essentially, BC treats imitation as a supervised learning problem where the input is the state of the environment and the output is the action taken by the expert.

Formulation In Behavioral Cloning, the goal is to learn a policy $\pi_\theta : S \to A$ that maps states $s \in S$ to actions $a \in A$. Given a dataset of expert demonstrations $\mathcal{D} = \{(s_i, a_i)\}_{i=1}^N$, where each (s_i, a_i) pair represents a state and the corresponding action taken by the expert, the learning objective is to minimize the discrepancy between the actions taken by the learned policy π_θ and those taken by the expert. This can be formulated as a supervised learning problem:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\pi_\theta(s_i), a_i), \tag{2.10}$$

where $\mathcal{L}(\pi_\theta(s_i), a_i)$ is a loss function measuring the difference between the predicted action by the policy π_θ and the expert action a_i. Common choices for the loss function include mean squared error (for continuous actions) or cross-entropy loss (for discrete actions).

2.5.2 Distribution Shift Problem

Behavior Cloning is simple to implement and effective when expert demonstrations are abundant and cover the state space comprehensively. However, it faces key limitations that reduce its robustness in real-world applications.

A primary issue is **covariate shift**: the policy is trained on states from expert demonstrations, but small prediction errors during deployment can lead it to encounter unfamiliar states. This distribution mismatch often results in degraded performance. Additionally, **error accumulation** becomes problematic, as even minor mistakes can compound over time, further pushing the policy away from the expert's intended trajectory.

The limited exploration of the expert can also lead to **limited generalization** of Behavior Cloning. As the expert provide demonstrations for the policy training, it will generally only follow optimal trajectories. Since the distribution of data provided as training data will only cover optimal or near-optimal trajectories, the policy will not be able to recover from the states far from the optimal trajectory, states which will very likely be encountered because of the error accumulation.

To overcome these issues, methods like Dataset Aggregation (DAgger) iteratively refine the policy by gathering data from states encountered by the policy itself, helping it handle diverse and unexpected situations more effectively.

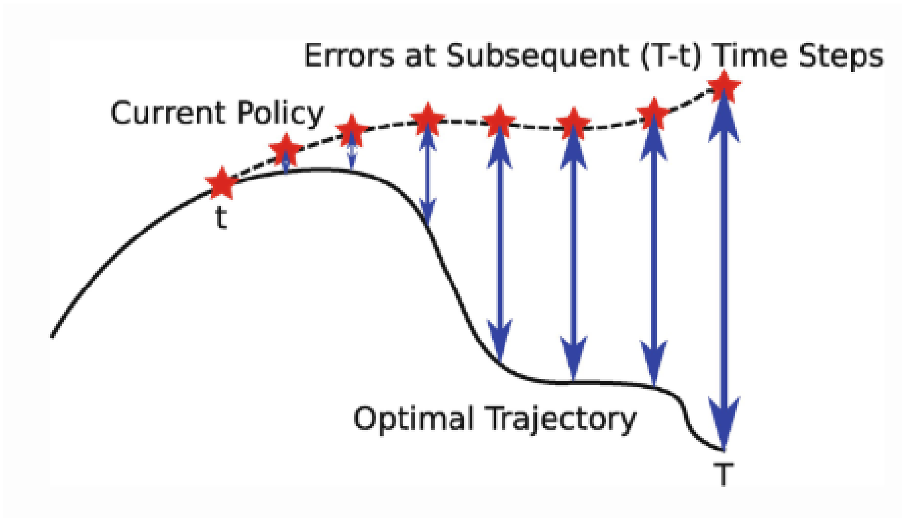


Figure 2.17: Visual explanation of the covariate shift and error accumulation problem. [13]

2.5.3 Dataset Aggregation (DAgger)

Dataset Aggregation (DAgger) [38] is an iterative algorithm designed to address the issues of covariate shift and the accumulation of errors. DAgger improves the robustness of the learned policy by incorporating states visited by the policy itself into the training process.

Algorithm Overview The DAgger algorithm works as follows:

1. **Initialize**: Start with an initial policy π_1 trained on an initial dataset of expert demonstrations $\mathcal{D}_1 = \{(s_i, a_i)\}_{i=1}^{N}$.
2. **Iterative Refinement**: For each iteration $t = 1, 2, \ldots, T$:
   (a) Use the current policy π_t to interact with the environment, generating a set of new trajectories.
   (b) For each state encountered during the trajectories, query the expert policy π^* to obtain the correct action, creating a new dataset $\mathcal{D}_t$ of state-action pairs.
   (c) Aggregate the datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_t$.
   (d) Retrain the policy π_{t+1} on the aggregated dataset $\mathcal{D}$ using supervised learning to minimize the discrepancy between π_{t+1} and the expert policy π^*.
3. **Output**: Return the final policy π_{T+1}.

Advantages of DAgger One of the main advantages of DAgger is its ability to reduce covariate shift. By collecting additional data from the states encountered by the learned policy, DAgger minimizes the mismatch between the training and testing distributions, enabling the policy to perform more consistently when deployed. This approach also significantly enhances the policy's robustness. Through the inclusion of expert corrections during training, DAgger helps the policy recover from errors and improves its capacity to generalize effectively to new, unseen states. Furthermore, DAgger's iterative framework ensures continuous improvement, allowing the agent to refine its performance over successive iterations and become more adaptive and robus in diverse situations.

2.6 Simulation-based Robot Learning

In the domain of autonomous drone racing, developing and testing control algorithms in real-world environments can be time-consuming, costly, and prone to hardware risks. Moreover, real-world data collection presents challenges related to sample complexity, where a large number of interactions are required for machine learning algorithms, particularly reinforcement learning (RL), to converge to an optimal policy. To mitigate these challenges, simulation-based robot learning has emerged as a powerful tool. By training autonomous drones in virtual environments, simulators enable faster data gathering, reduced sample complexity, and parallelized training processes.

Simulators allow developers to expose learning agents to diverse scenarios that would be difficult or risky to recreate in the real world. This capability accelerates the learning process and allows extensive exploration of failure modes, which are especially critical in high-speed environments like drone racing. Despite the advantages of simulators, one of the major challenges in this approach is the *sim-to-real gap*, where models trained in simulation fail to perform optimally when transferred to real-world environments. This gap arises from discrepancies in the dynamics, sensor noise, and environment modeling between simulations and reality.

Various simulators have been employed in robotic learning, each with unique strengths and weaknesses. In the next subsections, two different simulators will be presented: WeBots [49] and Nvidia IsaacSim.

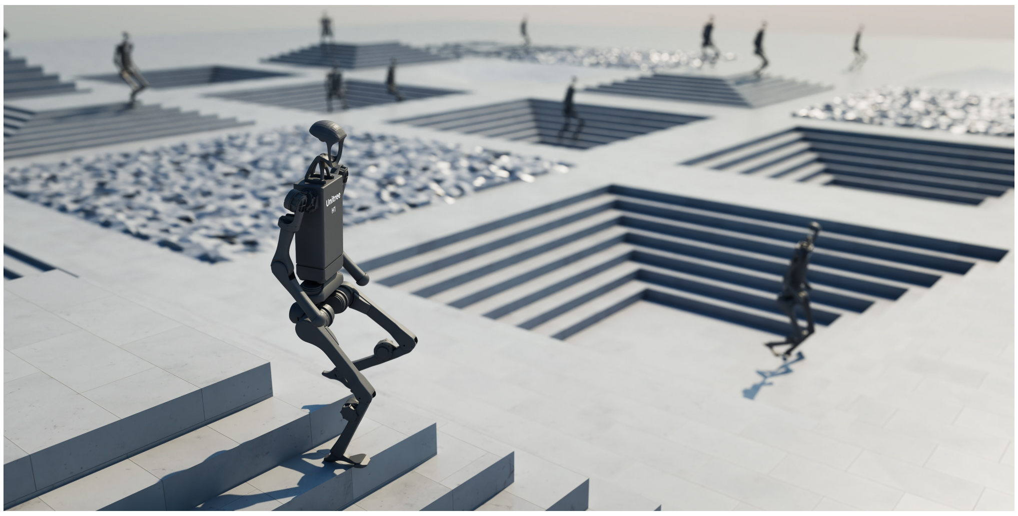


Figure 2.18: IsaacLab multiple randomized environments [28]

2.6.1 WeBots Simulator

Webots [49] is a versatile, open-source robotic simulator widely used in educational, research, and development contexts. Built with an emphasis on usability and cross-platform compatibility, Webots provides a user-friendly environment for designing, simulating, and testing robotic models, including autonomous drones. Webots offers high-quality 3D modeling and physics-based simulation for a variety of robots, making it an accessible tool for robotics projects where quick iteration and ease of setup are essential.

One of Webots' core strengths is its extensive library of prebuilt models and environments, which allow developers to quickly design and test new control algorithms without building complex simulations from scratch. This library includes various sensor and actuator models, making Webots a suitable choice for prototyping. Although Webots does not offer photorealistic rendering or advanced physics, it provides reliable physics engines, such as ODE (Open Dynamics Engine), capable of simulating realistic dynamics for standard robotic applications, including obstacle avoidance and navigation tasks. For drone applications, Webots can approximate aerial dynamics and sensor feedback, which supports the development of control strategies in a simplified environment before transitioning to more sophisticated simulators or real-world testing.

While Webots lacks GPU acceleration and the advanced parallelism, its simplicity and broad compatibility make it a practical choice for initial testing and educational projects. Furthermore, Webots is open source, allowing developers to customize the simulator to meet specific requirements, an advantage in research contexts where adaptability is key. However, due to limited capabilities in rendering and aerodynamic accuracy, Webots is less suited for high-fidelity applications, such as autonomous drone racing, where the *sim-to-real gap* becomes more pronounced. In these cases, more specialized simulators like Nvidia IsaacSim are better suited for achieving high-performance outcomes in complex, dynamic environments.

In summary, Webots provides a balance of accessibility and functionality that makes it a valuable tool in the early stages of robotic learning projects. It allows developers to iterate on design and control algorithms in a straightforward environment, complementing the capabilities of higher-fidelity simulators in the overall development workflow.

2.6.2 Nvidia IsaacSim Simulator

Nvidia IsaacSim is a state-of-the-art robotic simulator built on the Nvidia Omniverse platform, designed to support high-fidelity simulations and seamless integration with AI tools. It is especially suited for tasks like autonomous drone racing, where both the accuracy of the simulation and the efficiency of training are crucial.

One of the key advantages of IsaacSim over traditional simulators lies in its ability to reduce the *sample complexity* in reinforcement learning. By enabling highly parallelized simulations, IsaacSim allows drones to be trained in multiple environments simultaneously, drastically increasing the data collection rate. This parallelism significantly reduces the time required to gather sufficient experience for RL models to converge. Additionally, IsaacSim leverages Nvidia's GPU hardware to accelerate not only the physics simulation but also the training process, making it highly efficient compared to CPU-bound simulators.

IsaacSim also addresses several aspects of the *sim-to-real gap* by providing high-fidelity physics and sensor models. It employs Nvidia PhysX and FleX physics engines, which ensure that the simulated dynamics of drones closely mimic real-world conditions. This includes realistic aerodynamic models and support for complex environmental factors, such as wind, lighting, and collisions. Moreover, IsaacSim supports photorealistic rendering, enabling better training of vision-based algorithms, such as those used for object detection and navigation, which are essential in drone racing.

Compared to traditional simulators like Webots or Gazebo, which often prioritize ease of use over simulation fidelity, IsaacSim excels in creating highly realistic virtual environments. It integrates advanced GPU-powered rendering, which is crucial for simulating perception systems that rely on high-resolution imagery. This detailed rendering not only enhances the visual realism but also improves the training of deep learning models in environments that closely resemble real-world race tracks.

In summary, while the sim-to-real gap remains a challenge, Nvidia IsaacSim significantly narrows this gap through its high-fidelity simulations, advanced physics, and photorealistic rendering, providing a more effective platform for developing autonomous drone racing systems compared to traditional simulators.

2.7 Neural Architecture Search (NAS)

Neural Architecture Search (NAS) is a technique in machine learning that automates the process of designing neural network architectures, allowing for more efficient and optimized models. Traditionally, neural network architectures are manually designed, which is time-consuming and relies heavily on expert knowledge. NAS alleviates this by using algorithms to automatically explore a search space of potential architectures, evaluating their performance on a given task, and iteratively refining the topology of the network to find the most effective structure.

NAS applied for edge devices, in particular, focuses on creating efficient neural network architectures that meet the computational and memory constraints typical of mobile, IoT, and embedded systems. Edge devices have limited processing power, memory, and battery life, so NAS is applied here to optimize for lightweight architectures that deliver high accuracy while maintaining low latency and power consumption.

In this context, NAS relies on a search strategy that favours the architectures within the search space with a lower complexity, and often incorporates constraints directly into the search process to balance performance and efficiency. These constraints guide the search strategy to minimize the number of parameters, reduce computational demands, and ensure models fit within the hardware capabilities of the edge device.

2.7.1 Differentiable NAS (DNAS)

Differentiable Neural Architecture Search (DNAS), also known as one shot NAS methods, is a family of NAS approaches that formulate the architecture search process as a differentiable optimization problem, allowing gradient-based methods to guide the search.

In DNAS, instead of evaluating each candidate architecture individually, the algorithm constructs a supernet: a large, over-parameterized network that includes multiple possible architectures within it. Through this supernet, DNAS assigns learnable weights to various architecture choices (e.g., types of layers or connections) and optimizes them simultaneously with model parameters. To search for accurate and efficient architectures, DNAS tools enhance the normal training loss function with an additional differentiable regularization term that encodes the cost of the network. Typical cost metrics are the number of parameters and the number of Multiply-Accumulate operations (MACs) per inference. Mathematically, DNAS tools search for:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \lambda \mathcal{R}(\theta) \tag{2.11}$$

where $\mathcal{L}$ is the task loss function, $\mathcal{W}$ is the set of standard trainable weights (e.g., convolutional filters), θ is the set of additional NAS-specific trainable parameters that encode the different paths in the supernet, $\mathcal{R}$ is the regularization loss that measures the cost of the network and λ is a hand-tuned regularization strength, used to balance the two loss terms.

By applying backpropagation across these choices, DNAS identifies high-performing architectures in a fraction of the time and computational cost of traditional NAS methods. This approach enables more efficient, scalable architecture discovery, making it especially useful for applications requiring rapid iteration or deployment on devices with limited computational resources.

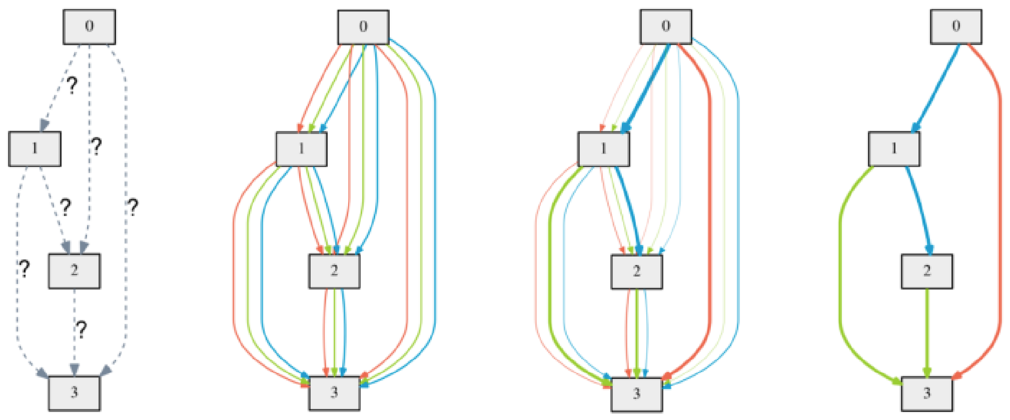


Figure 2.19: An example of DNAS method (DARTS [25]): (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

2.7.2 Types of DNAS

Path-based DNAS

Path-based DNAS methods define a DNN (the supernet) whose graph includes multiple alternative paths corresponding to the possible alternative operations in the search space. The optimization reduces to selecting one of these paths. An illustration of this method can be seen in Fig. 2.20. The main issue with this approach is that the supernet size grows quickly with the search space, limiting scalability.

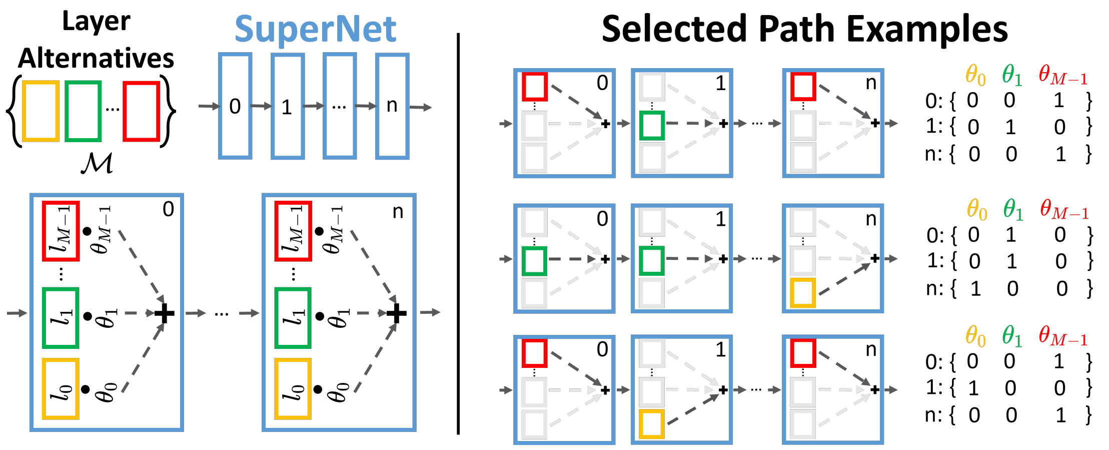


Figure 2.20: Path-based DNAS illustration, as implemented in PLiNIO Python library [19].

Mask-based DNAS

Mask-based DNAS introduces an additional level of efficiency in the architecture search process by using binary masks to selectively activate or deactivate parts of a supernet (here called seed network) during training. Instead of assigning continuous weights to each possible path or operation in the seed network, Mask-based DNAS applies binary decisions to either "mask in" or "mask out" certain architectural choices, streamlining the gradient-based optimization. This masking mechanism significantly reduces memory and computational overhead because only the active parts of the architecture need to be evaluated and trained at any time. By focusing on a discrete subset of options, Mask-based DNAS accelerates the search while retaining the flexibility of differentiable approaches, enabling the development of lightweight, deployable models optimized for specific hardware constraints.

2.7.3 DNAS Training Procedure

In Differentiable Neural Architecture Search (DNAS), the process is divided into three key phases: **warmup, search, and fine-tuning**.

The **first phase, warmup**, involves standard training of the full supernet (in path-based DNAS) or the seed network (in mask-based DNAS). Here, only the regular weights W are trained, while the architecture parameters θ are kept fixed at their initial values. This allows all possible paths (for path-based DNAS) or all channels (for mask-based DNAS) to be fully trained. For path-based DNAS, this results in all supernet paths being sampled uniformly, whereas for mask-based DNAS, all channels in the seed model remain active. Throughout warmup, gradients are computed only with respect to the task-specific loss function, $\mathcal{L}(W; \theta)$. Importantly, warmup results are independent of any architectural constraints, making them reusable across multiple searches targeting different objectives or hardware requirements.

The **second phase is the architecture search**, where both weights W and architecture parameters θ are optimized together according to an objective function (such as that described in eq. 2.11 or eq. **??** with the DUCCIO regularizer). This phase involves a training loop that runs for at least E_{sr} epochs and continues until an early-stopping mechanism detects convergence by monitoring improvements in the task loss and NAS loss sum on the validation split. Once the search converges, the architecture corresponding to the final values of θ is exported.

In the **final phase, fine-tuning**, only the extracted model's weights W are further trained, again using the task-specific loss $\mathcal{L}$, similar to the warmup phase.

Chapter 3

Related Work

3.1 Machine Learning in Autonomous Drone Racing

:earning-based approaches have shown significant promise in autonomous drone racing by leveraging data-driven methods that allow drones to navigate and control themselves in complex environments. These methods replace or augment traditional methods with neural networks, offering flexibility in high-dimensional data processing, robustness to sensor noise, and fast replanning capabilities. In this chapter, a review key works in this field is presented.

3.1.1 End-to-End Learning

End-to-end learning systems aim to process raw sensory inputs and directly produce control commands, simulating the process by which human pilots operate drones using first-person-view cameras.

Modular End-to-End Learning

In modular end-to-end learning approaches the classic perception, planning and control blocks are all substituted with neural networks.

In [31] and [24], the authors train a perception-planning network and a control network using IL. The perception-planning network uses RGB images as input and predicts high-level waypoints (a path). These path predictions, together with the state estimation of the drone are then used by the controller networks as inputs to produce the controller outputs. In order to speed up the IL training, both these works used multiple teachers. In [24], for instance, Li et al propose observational imitation learning: a IL variant that supports online training and automatic selection of optimal behavior by observing multiple imperfect teachers.

By using a modular approach, these methods allow for independent training of each block to output 'classic pipeline'-like outputs, promoting interpretability. Yet, the division into independent blocks leads to compounding errors and latencies, which negatively affect performance when flying at high speeds [16].

Fully End-to-End Learning

The second family of learning approaches is the fully end-to-end. In these methods a unique neural network is used to approximate the classic perception, planning and control blocks, without any supervision on intermediate results of the network.

In [30], Muller et al implement an end-to-end learning framework where a CNN was trained through imitation learning to map images from a camera to control commands, bypassing the need for explicit trajectory planning or state estimation. In [37], Rojas-Perez et al also analyze the improvement gained by using a history of images, showing better performance of the network when using a history of images.

Similarly [14] and [50] show the use of a history of images and state estimates. However they do so by separating the visual feature encoder from the temporal feature encoder. This division allows for more advanced techniques to be used on the visual-feature encoder, such as latent-space learning. In [14], for example, they use BYOL [15] in order to learn more robust latent embeddings for the images through contrastive learning. The latent image embeddings are then used as a sequence for the temporal feature extractor.

While the previous works all used IL as their core policy search algorithm, in [51] Xing et al. propose the use of IL only as an intermediate step, which trains a working, yet not optimal, policy for RL. By using an already trained (even if not optimal) policy they ensure the convergence of the RL policy search, whereas without the IL bootstrap wouldn't be possible. The use of RL allows the vision-based agent to explore actions different by the one learned by the teacher, allowing it to go beyond the performance upper limit imposed by simply cloning the teacher's behavior.

Even though these methods show state-of-the-art performances, they don't show interest in the computational cost of the policies, allowing for costly policies which are not deployable on the nano-devices of interest of this work.

3.1.2 Domain Generalization

Domain generalization is essential in autonomous drone racing to bridge the gap between simulation and real-world environments. This challenge has motivated a variety of methods to improve model robustness across domains.

Domain randomization, for instance, introduces controlled variability in simulated environments to enhance model transferability to real-world conditions. In their work [26], Loquercio et al. apply domain randomization, modifying lighting and gate textures in simulation to create a model that achieves zero-shot sim-to-real transfer. This enables drones to handle real-world variations, such as lighting changes and diverse gate appearances, without additional tuning on real data.

Feature-level techniques also play a significant role in domain generalization. Pham et al. [34] propose a pencil filter to improve robustness by emphasizing geometric structures in images, thus helping models generalize to new environments where visual features differ from those seen in training.

Other works also improve domain generalization and robustness by working directly on the latent space produced by the neural networks. In [14], Fu et al. use contrastive learning and data augmentation in order to build robust feature representations from images, demonstrating resilience against visual disturbances and unstructured environmental variations that were not experienced during training.

3.2 TinyML on Ultra-Low Power SoCs

TinyML at the edge focuses on deploying models directly on devices with limited computational power, memory, and energy (such as mobile phones, sensors, IoT devices or, in this case, nano-drones) rather than relying on centralized cloud processing. Robots, particularly those used in industrial automation, agriculture, or healthcare, benefit from the low latency of these models, enabling real-time processing of sensory data like images, audio, and physical sensor readings. This is crucial in applications such as autonomous navigation, object recognition, and human-robot interaction, where immediate response is necessary for safety and efficiency.

3.2.1 Nanodrones as Resource Constrained Application

DroNet is a compact 8-layer residual CNN tailored for real-time obstacle avoidance and navigation in urban environments introduced by Loquercio et al. in [27]. The architecture includes skip connections typical of residual networks, enhancing stability and enabling generalization across different scenarios despite the network's simplicity. DroNet's design relies on lightweight convolutions to reduce memory and compute demands, which allows it to run efficiently on resource-constrained drones while maintaining high-level navigation capability. This study shows how combining residual architectures with small, streamlined networks can enable responsive navigation on minimal hardware.

In [33] Palossi et al. introduce PULP-Frontnet, a neural network optimized for real-time human-drone interaction on nano-UAVs. This network was specifically designed to operate on the parallel ultra-low-power (PULP) architecture, a constraint-driven approach that includes reduced convolutional layers and 8-bit quantization. PULP-Frontnet achieves 135 frames per second (fps) while requiring only 86 mW on the GAP-shield mounted on top of the Crazyflie 2.1 drone. Notice that this power consumption is negligible over the power of the full systems, in which the propellers' power dominates the overall consumption of the drone. The architecture focuses on maximizing efficiency, balancing reduced memory and power with accuracy. This work underscores how hardware-specific design choices, such as using lightweight layers and integer quantization, can allow effective DNN deployment in extreme resource-constrained environments.

Building upon the DroNet, Lamberti et al. present in [23] a series of Tiny-PULP-Dronet, an advanced version of Dronet that achieves a 50× reduction in parameters and a 27× reduction in MACs. The architecture modifications focus on trading off redundant channels and neurons without impacting essential features, allowing the model to reach 160 fps. This work demonstrates the potential of pruning and layer optimization in supporting multi-tasking on ultra-low-power systems.

On the same idea of optimizing successfully deployed networks, Cereda et al. apply NAS to PULP-Frontnet and MobileNetV2 architectures for visual pose estimation in order to achieve better performance to computational cost tradeoffs on the Crazyflie 2.1 nano-drone [10]. The NAS process systematically explores combinations of convolutional layers, filter sizes, and layer depths to generate architectures that balance accuracy with low computational load. Using NAS, the resulting CNNs achieves up to 50 FPS with a 32% improvement in control accuracy over prior models. This study underscores how NAS can streamline the architecture tuning process for nano-drones, producing models with optimized convolutional configurations and reduced latency, essential for resource-limited onboard AI applications.

These works, although successful in deploying deep learning policies on computationally constrained devices (such as the Bitcraze Crazyflie 2.1), investigate tasks such as were obstacle avoidance or human-drone interaction. Differently, this work analyses the potential of deep learning policies for autonomous drone racing, in which more reactive perception and a more precise control is needed as the drone flies at high speeds.

Chapter 4

Methods

This chapter is devoted to a comprehensive description of the methodologies employed to address the task of developing an end-to-end deep learning policy for autonomous drone racing competitions, aimed at the deployment on the Bitcraze Crazyflie 2.1, an ultra-low-power nano-drone. In the first section 4.1, an explanation of the task and its challanges is given. Next, the section 4.2, a high-level overview of the whole development process. In the next sections (4.3 and 4.4), there is a more in depth explanation of the approach steps introduced in section 4.2.

4.1 Task Description and Challenges

This thesis focuses on **ADR gate-based navigation competitions**. This category of competitions require the autonomous systems to have precise control, accuracy, and spatial awareness as drones must accurately detect and fly through specific gates positioned along the race course at high speeds.

In these competitions the track is often unknown a priori, requiring the autonomous drone systems to be robust to unknown tracks. Given the requirements of such a competition, recent research found **fully end-to-end deep learning policies** to be a good candidate for their good generalization capabilities, robustness to sensor noise and fast adjustment to environmental changes [14, 50]. These characteristics led the fully end-to-end deep learning policies to become state-of-the-art when it comes to gate-navigation ADR competitions, which is why this thesis focuses on this kind of systems.

However, these deep learning systems often have **high computational cost**. Commercial drones, often come with hardware accelerators, such as GPUs, in order to allow on-board real-time performance. Ultra-low-power nano-drones, such as the Bitcraze Crazyflie 2.1, cannot carry these kinds of accelerators as they are both too heavy w.r.t. their size and too expensive in terms of energy. The Crazyflie 2.1, however, often comes with the AI deck, which features the GAP8 (sec. 2.1.3), an IoT application processor that enables the onboard execution of neural networks. Even though the deployment is out of the scope of this thesis, the deployability of the neural network is a main objective of this work. This is why a substantial portion of the work is dedicated on ensuring that the policy network developed satisfies the computational constraints which allow for the on-board real-time deployability on the Crazyflie 2.1.

A central challenge that the training of the fully end-to-end deep learning policy faces is their **high sample complexity**. Deep learning systems are known for being very data-hungry, requiring a large amount of data in order to produce good results. In robotics, collecting real-world data is often discouraged, as it is both time consuming and can lead to damaging the real-world robot. Recently **simulators** have been used as a efficient way to generate large volumes of data in a controlled, cost-effective manner. Even if state-of-the-art simulators provide high quality rendering, there is still a mismatch between the input distribution that the network sees in the simulator at training time and the input distribution in the real world. This mismatch is often called "**reality-gap**" and it leads to poor performance of the policy in the real world scenario w.r.t. the performance in simulation, which is why particular attention has given also to training techniques that enhance the **domain generalization** capabilities of the network, in order to reduce the reality-gap and have a zero-shot deployment on the Crazyflie 2.1.

To sum up, this thesis focuses autonomous nano-drone navigation in ADR competitions. The main goal is to develop a fully end-to-end vision-based deep learning policy that can handle precise control, adapt to unknown tracks, and operate reliably in different environments. A key part of the work is ensuring that the policy is efficient enough to run in real time on the Crazyflie 2.1, a resource-limited nano-drone, which relies on lightweight processors like the GAP8. The thesis also tackles the reality gap between simulated training and real-world use by using domain generalization techniques for better transferability. Finally, it addresses the high data requirements of deep learning by relying on simulators to generate training data, reducing the need for time-consuming and potentially risky real-world data collection.

4.2 Approach Overview

4.2.1 Simulator and Environment

The simulator choice is one of the most important decisions that have to be made in a robot-learning project, such as this thesis, as it influence what techniques can and cannot be used. Some simulators, such as WeBots [49], better described in section 2.6.1, focus on ease of use and flexibility, instead of performance or high-fidelity, which can be of good use for initial testing or educational projects. On the other hand, more complex simulators, such as Nvidia IsaacSim, better described in 2.6.2, yield better speed and higher quality rendering by taking advantage of GPUs, but at the cost of being less user-friendly.

While WeBots was used in the early stages of the development of this thesis, the final project was based on **Nvidia IsaacSim** because of its native integration with the software infrastructure used to fully exploit NVIDIA GPUs, allowing for fast and high-fidelity rendering. Along with that, the IsaacLab [28] library built upon IsaacSim gives a Gym-like [48] interface, which allows for the usage of more complex and data-hungry techniques such as reinforcement learning.

Environment

In ADR competitions focused on gate-based navigation the objective is to traverse a sequence of N gates, with an arrangement often unknown a priori, as fast as possible. These competitions are usually held in large, empty warehouses, in which are present only the competing drones and the gates that make up the track. Because of the simplicity of this setting, the environment is composed only by the drone agent, the floor, the N gates and the skybox (e.g. the image background). Furthermore, since these are time-based competitions, there is no need to add competitor agents within the environment.

Random Track Curriculum

As said previously, in ADR the tracks are rarely known in advance: because of that, it is important to build an agent that can handle a distribution of tracks, instead of one single track. For this reason, a *track generator* has to be employed. Taking inspiration from [43], the track is defined as:

$$\mathbf{T} = [\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_N]$$

where $\mathbf{G}_{i+1} = f(\mathbf{G}_i, \Delta\mathbf{p}, \Delta\mathbf{R}) \in R^3 \times SO(2)$ is the definition of the gate $\mathbf{G}_{i+1}$ as a function parametrized by the previous gate $\mathbf{G}_i$, as well as the relative position $\Delta\mathbf{p} \in R^3$ and the relative orientation $\Delta\mathbf{R} \in SO(2)$. By adjusting the range of relative poses and the total number of gates, race tracks of arbitrary complexity and length can be generated.

In our specific case, the $\Delta\mathbf{p}$ is defined through polar coordinates on the 2D plane, while the height (defined on the z-axis) was defined independently. On the other hand $\Delta\mathbf{R}$ in our case only defines a yaw angle, here called ψ as the gates are intended to be upright, while the yaw angle is only important for the curvature of the track. Since the curvature of the track is handled by $\Delta\mathbf{R}$, the angular part in $\Delta\mathbf{p}$ called θ is used just as noise.

In order to allow the creation of unique ADR tracks, both $\Delta\mathbf{p}$ and $\Delta\mathbf{R}$ are sampled from uniform distributions. Furthermore, to give the tracks structure, instead of using the pure random samples from the uniform distribution, a simple moving average filter and subsequent rescaling is used over the samples of the same track. In fig. 4.1 an example of a track generated with the method described is shown.

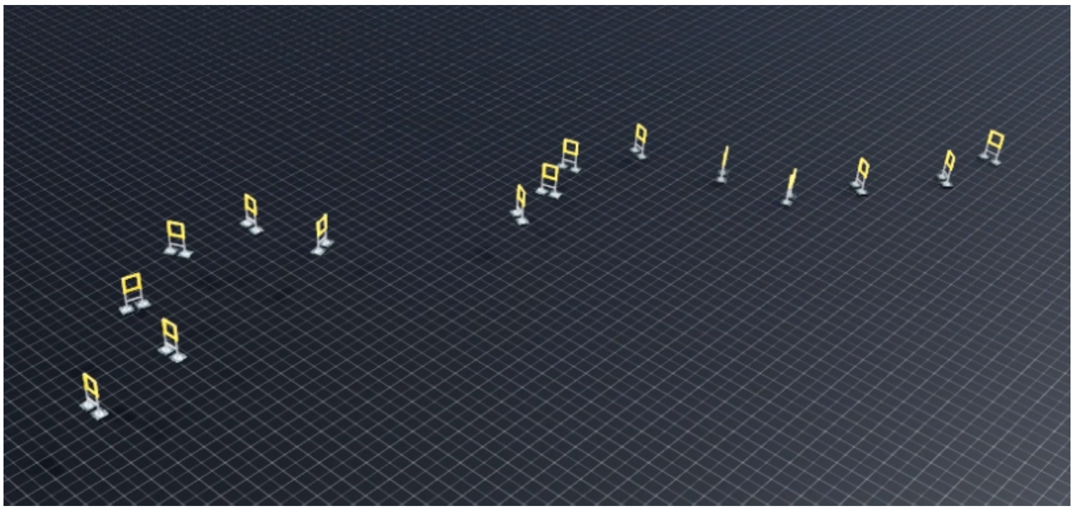


Figure 4.1: Example of track generated with the method described.

4.2.2 Policy Development

In order to satisfy all the constraints that the problem at hand imposes, several steps have to be done in the policy development. The main characteristics that the final policy has to have are:

- *On-board sensor based*: it has to use images from the on-board camera as its main input as well as state estimation from the on-board IMU and accelerometer;
- *Accurate navigation*: it has to be able to finish the ADR competition tracks;
- *Real-time performance*: it has to comply to the computational constraints given the resource limited deployment target.

In order to develop a policy that has all the characteristics cited the following steps were done:

Step 1: Teacher Policy via RL

The teacher policy leverages privileged information (such as the exact gate position and orientation, drone velocity, acceleration, etc.) in order to allow it to take the optimal actions, which consists of the optimal thrust and angle setpoints for the quadrotor (sec. 2.1.1). This smaller and simplified input space allows the teacher policy to be trained with reinforcement learning.

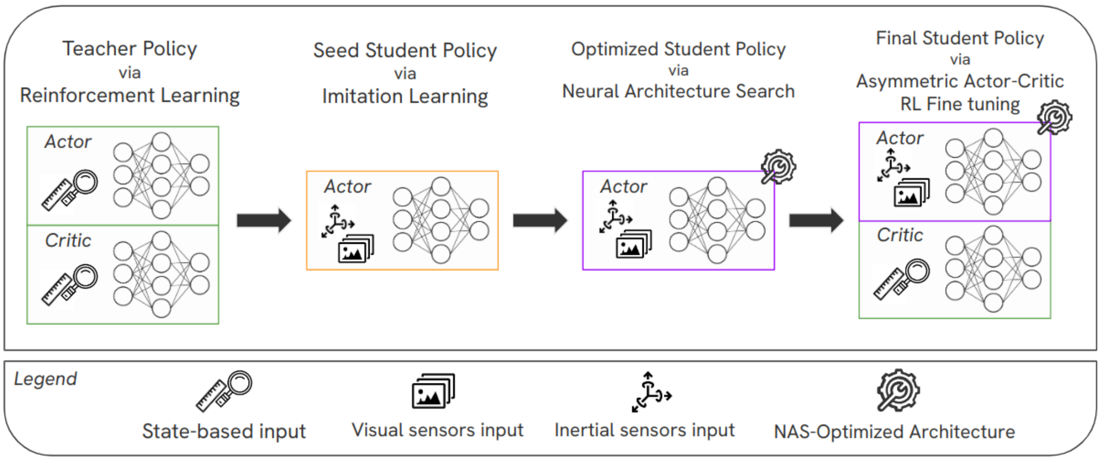


Figure 4.2: Proposed method: leverage the imitation learning supervised nature to introduce a neural architecture search step, in order to respect the computational constraints

The teacher is trained, not only to complete the gate navigation task, but also to have behaviors that would allow the student network training: such as always having the next gate in the field of view of the camera and having similar actions in subsequent timesteps.

The **teacher policy** is able to perform in **real-time** (as its network is very small) and **navigate accurately the race track** (as it reaches optimal results), but it is **not on board sensor based** (as it leverages information that is not coming from the on-board sensors and that is not obtainable in a real world scenario).

Step 2: Student Network via IL

The student cannot be trained directly with reinforcement learning, as the training would be too unstable and it would not converge due to the large, highly dimensional input space. Because of this the student policy has to be trained with the learning-by-cheating framework. The learning-by-cheating (presented in [14]) uses a teacher policy to collect the partial information data (in this case images and state estimates) and the optimal actions computed by leveraging the priviledged information. The dataset collected is used for the partial information student policy training, in which it learns to mimic the actions that the priviledged-information teacher has used, but by observing the partial information as input (image and state estimation). By leveraging such a dataset, the student policy is trained in a supervised manner, often called imitation learning (sec. 2.5). In order to overcome the distribution shift problem (sec. 2.5.2) that a simple collect & train approach would introduce, the DAgger algorithm is used (sec. 2.5.3). This algorithm iteratively expands the dataset by allowing the student to explore and collecting the teacher's actions in the new situations the student is encountering.

The student is trained on a regression task, as the teacher's outputs are continuous. The student network architecture receives as input a history of images and state estimates, along with the current ones, as it has been shown to increase performances [14, 50]. This allows the network to expand its observation space, enhancing it with historical information from which it can extract intrinsic information, such as velocity and acceleration.

This second step outputs a **student network** that uses **on-board sensor data**, however it is **not accurate** (due to IL's limitations, sec. 2.5) and it is **not real-time** deployable. The goal of this first step is not to have the optimal student network in terms of navigation ability, but to create a dataset which is robust enough to train a policy that shows reasonable performance.

Step 3: Student Network Optimization via NAS

The dataset created as a byproduct by the DAgger algorithm allows for the optimization of the student network with PIT [36], a NAS algorithm which reduces the size of the network in a structured way, by removing the channels/neurons which are less impactful for the outputs, having minimal task performance drops. This procedure is used to reduce the student network's computational cost, in order to ensure its real-time performance. In this context, the student policy trained at the second step is called a seed network, as it can be seen in fig. 4.2.

The NAS procedure outputs a **NAS-optimized student network**, which uses**on-board sensor data** and has **real-time performance**, but it is still **not accurate**. However, since the PIT algorithm objective is to reduce the task performance drops as much as possible, the performances are similar to the seed network's. This means that the optimized policy is able to fly, even if not optimally.

Step 4: Student Network RL fine-tuning via AAC

The NAS-optimized student network given by the previous step is here used as an actor in the asymmetric actor-critic (sec. 2.4.5) framework. In the AAC framework a the actor has access only to partial-information, while the critic has access to the priviledged information, as it would not be used for deployment and it only helps with training stability. Since the optimized policy is pre-trained and able to navigate, in this context is called a *bootstrapped* agent. In this step a bootstrapped critic it's also used, by leveraging the pre-trained critic of the teacher policy.

The RL fine-tuning objective is to maximize the performances of the agent, by allowing the bootstrapped agent to explore different actions in the environment. It is noteworthy that this step wouldn't be applicable unless the actor is bootstrapped, orelse the training would not converge due to the highly dimensional observation space. In practice, if the agent is still not able to fly at this step, the exploration would hardly ever come across a high positive reward signal, thus the RL process would lead to a catastrophic forgetting. By using a bootstrapped student policy, it is already close enough to the optimal student policy, so it converges, achieving the best performance the student policy can achieve.

The resulting RL fine-tuned agent uses **on-board sensor data**, is **accurate** and has **real-time performance**, which are all the characteristics that our final policy has to have.

4.3 Priviledged Teacher Agent Training

The teacher policy is trained via reinforcement learning due to its low dimensional input space. In the next sections an overview of the setup of the training of the priviledged teacher agent is presented, going through the observation and action spaces, the reward function used for the training and the teacher policy neural network architecture.

4.3.1 Observation and Action Spaces

Inspired by [43], the **observation space** consists in two main components: s_t^{quad} which gives information about the drone state and s_t^{task} which gives information about the race track.

The drone state component s_t^{quad} consists of the drone's linear velocity in the drone's coordinate system, angular velocity in the drone's coordinate system, angles in the world's coordinate system, the projected gravity in the drone's coordinate system and the actions taken at the previous timestep.

$$s_t^{quad} = [v_t^D, \omega_t^D, \theta_t^W, g_t^D, a_{t-1}] \in R^{16}$$

The task state component s_t^{task} consists of the position of the next N gates in spherical coordinate's system $p_i = [p_r, p_\theta, p_\phi]_i \in R^3, i \in \{1, \ldots, N\}$. As discussed in [43] the spherical coordinates representation separates the distance of the gate and its direction in the reference frame origin and provides a more informative description for gate-navigation task w.r.t. the Carthesian coordinates. For the immediately next gate to be passed, a drone-centered reference frame is used, while all other gate observations are recursively expressed in the frame of the previous gate. Thus the task state component is composed as follows:

$$s_t^{task} = [p_1, p_2, \ldots, p_N] \in R^{3N}$$

The complete teacher observation is the concatenation of s_t^{quad} and s_t^{task}.

The **action space** is defined by the desired total thrust (force, N), the desired roll rate, the desired pitch rate and the desired yaw rate (angular velocities, $\frac{rad}{s}$). As described in chapter 2.1, this representation of the desired setpoints is simpler and more understandable than the thrust setpoint for each motor. Nevertheless the mapping to motor thrusts is a linear one, through the motor mixing algorithm (sec. 2.1.1). Thus the action space is:

$$a_t = [T, \tau_\phi, \tau_\theta, \tau_\psi] \in R^4$$

4.3.2 Reward Function

Drone racing's objective in general is finishing a track in the least time possible. Even though it would make sense to use directly this as a reward feedback, the reward signal would be incredibly sparse: at the beginning of the training the agent acts basically as a random agent, thus all track runs (in this context called episodes) finish up with a crash. The probability of such an agent of using the correct combination of (random) actions in order to finish a track fastly approaches zero, making the credit assignment for each individual action impossible.

A popular approach to circumvent this problem is to use a proxy reward that closely approximates the true performance objective while providing feedback to the agent at every time step [26].

Along with objective proxy reward, which make the agent successful in the task, the reward function has to also promote/discourage certain behavioral aspects of the agent, such as safety and stability. In our specific case, it has to incentivize also the perceptive behavior in order to allow a correct data collection for the vision-based agent.

The complete reward function (inspired by multiple works in ADR via deep RL, such as [26, 51, 14]) results in a summation of different rewards (R), each scaled relatively to their importance by their own factor (λ):

$$\begin{aligned} R(s) = {} & \lambda_{passed\_gate} R_{passed\_gate} \\ & - \lambda_{crash} R_{crash} \\ & + \lambda_{progress} R_{progress} \\ & - \lambda_{safety} R_{safety} \\ & + \lambda_{perception} R_{perception} \\ & + \lambda_{blur} R_{blur} \\ & - \lambda_{ang\_velocity} R_{ang\_velocity} \\ & - \lambda_{smoothness} R_{smoothness} \end{aligned} \tag{4.1}$$

In the next sections a more in depth presentation of each one of components of the reward function is given.

Whenever possible the R components were scaled into a range of $(0, 1)$ or $(-1, 1)$ in order to ease the choice of the λ parameters.

Based on each term's meaning and implementation, the rewards are split between rewards and penalties. The way they are distinguished is by the sign present in the eq. 4.1: *a positive sign* reward is encourages the behavior that produces it, while a **negative sign** reward is discourages it.

Passed gate reward

This reward encloses the main objective of our agent: passing the gates that describe the track. This reward is given as a feedback every time the drone passes the gate which is next in order to finish the track. Mathematically it's described as:

$$R_{passed\_gate} = \mathbb{1}_{passed\_gate} \tag{4.2}$$

Where the indicator function $\mathbb{1}_{passed\_gate}$ is equal to one only if the agent has passed the objective gate. In order to better understand this indicator function, the following concepts have to be introduced:

- pos_T^G: which is the position vector of the drone at time T in the gate's (G) coordinate system;
- $pos_{T,n}^G$: which is pos_T^G, projected on the gate's plane vector (in other words, the absolute value is the distance from the gate's plane, while the sign represents which side of the gate's plane the agent is on);
- $pos_{T,t}^G$: which is pos_T^G, projected on the gate's plane.

Given the previous definition, we can provide a more detailed explanation of the previous indicator function. More specifically we can define:

$$\mathbb{1}_{passed\_plane} = ((pos_{T,n}^G \cdot pos_{T-1,n}^G) < 0)$$

Which indicates if the drone has traversed the gate plane. This alone is not enough to indicate the pass of a gate though, thus we need another function:

$$\mathbb{1}_{inside\_gate} = (||pos_{T,t}^G||_{inf} < 0.5)$$

Which indicates if the drone is 'inside' the gate. The infinity norm was used as the gates used are squared, while 0.5 is half of the length of its side.

By using the previous two functions we can define the indicator function for our reward as:

$$\mathbb{1}_{passed\_gate} = \mathbb{1}_{passed\_plane} \cdot \mathbb{1}_{inside\_gate} \tag{4.3}$$

Even though this should be enough to ensure the agent finds a good policy, the sparsity of this reward may become an issue, as the agent may not be exploring enough to ever find this reward. Because of this issue denser rewards were introduced, such as the 'Progress reward' which will be described later.

Crash penalty

Crashing in drone racing is generally not encouraged: while some very small contacts with the gate can be tolerated in order to optimize the trajectory, in the majority of the cases even small contacts can introduce such a big deviation from the trajectory that the drone stability is compromised and it cannot recover. Because of this, a penalty for crashing was introduced using the in-simulation force sensors:

$$R_{crashed} = \mathbb{1}_{crashed} = (\sum ||F_{ext}||_2 > 0) \tag{4.4}$$

Of course the external forces F_{ext} do not take into consideration propeller forces, but only forces applied by external objects. In this case air drag is not considered.

Progress reward

The progress reward serves multiple objectives: the first one is to give a denser reward towards the main aim which is to pass the gates, while the second one is to limit the maximum velocity magnitude the drone reaches, in order to account for simulator's representation limitations. Because of this the progress reward is split in two parts:

$$R_{progress} = clip(R_{towards\_gate} + R_{max\_vel}; \quad min = -1; \quad max = 1) \tag{4.5}$$

In order to better understand these two components we have to introduce the following:

- v_T^D: is the velocity vector of the drone at time T in the drone's coordinate system;
- $gate\_pos_T^D$: is the gate position vector at time T in the drone's coordinate system;
- $\bar{v}$: is the target velocity, a hyperparameter which defines the maximum speed at which we want the drone to fly

The $R_{towards\_gate}$ is described as:

$$R_{towards\_gate} = clip(\frac{1}{\bar{v}}\langle v_T^D \ , \ \frac{gate\_pos_T^D}{||gate\_pos_T^D||_2}\rangle; \quad min = -1; \quad max = 1)$$

where the velocity vector in the drone's coordinate system is dot-multiplied with the unit vector representing the gate's position, also expressed in the drone's coordinate system. Dividing by the target velocity ensures that when the drone reaches the projected velocity equal to the target velocity the reward is equal to one, while the clip ensures that projected velocities bigger than the target velocity are not further incentivised.

The $R_{towards\_gate}$ alone does not ensure that the agent will not go beyond the target velocity, thus R_{max_vel} component was introduced as:

$$R_{max\_vel} = \begin{cases} 2e^{-\frac{(||v_T^D||_2 - \bar{v})^2}{0.2}} - 1, & \text{if } ||v_T^D||_2 \geq \bar{v} \\ 0, & \text{otherwise} \end{cases}$$

Which is a half-gaussian, centered in $\bar{v}$ (here 2.5) having a small standard deviation (here 0.2) and rescaled in the range $(-1, 1)$. By using this function we let the agent exceed the target velocity without much penalty, but for a very short range, whereas if we used a simpler way to define it, such as giving reward equal to -1 everytime it exceeds it, the agent would not want to approach the target velocity too closely, as it would risk to get a negative reward.

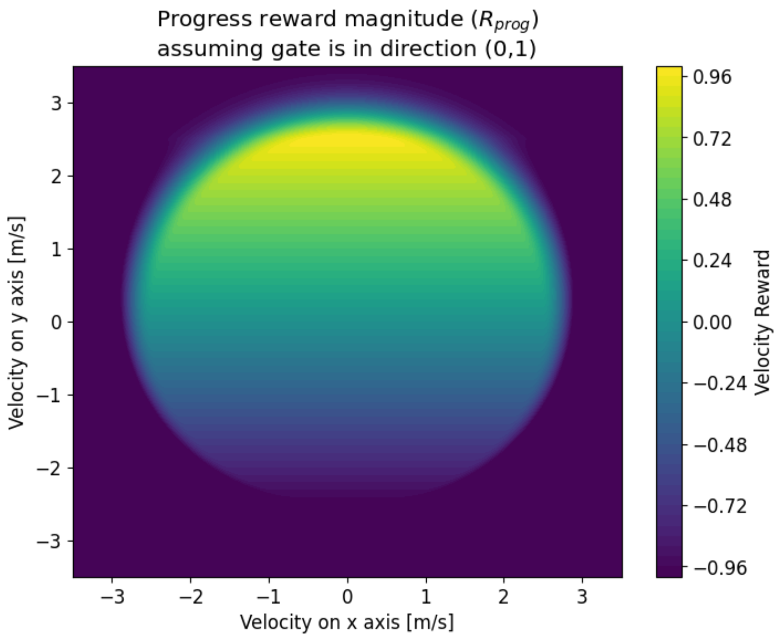


Figure 4.3: The progress reward in 2D, assuming the gate is in direction (0,1)

Safety penalty

In order to introduce a denser penalty w.r.t. the crash penalty, the safety penalty is introduced, taking inspiration from [43]. It penalizes the agent for being in positions which are dangerous, while incentivising it to progress through the center of the gate. The safety reward R_{safety} is defined as:

$$R_{safety} = f^2 \cdot \left(1 - \exp\left(-\frac{0.5 \cdot d_n^2}{v}\right)\right) \tag{4.6}$$

where $f = \max\left[1 - \left(\frac{d_p}{d_{\max}}\right), 0.0\right]$ and $v = \max\left[(1 - f) \cdot \left(\frac{w_g}{6}\right), 0.05\right]$.

Here, d_p and d_n denote the distance of the quadrotor to the gate normal and the distance to the gate plane, respectively. The distance to the gate normal is normalized by the side length of the rectangular gate w_g, while $d_{\max}$ specifies a threshold on the distance to the gate center in order to activate the safety reward. A visual representation can be seen in fig. 4.4.

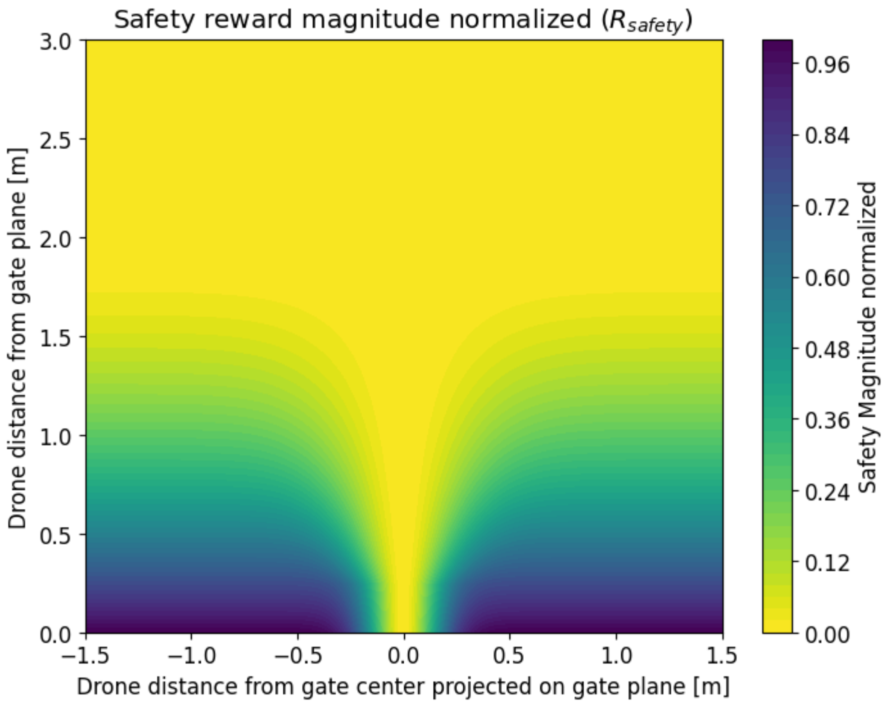


Figure 4.4: Safety reward illustration, in which the gate's position is in (0,0) and the gate plane is positioned on the x-axis.

Perception reward

Since the final objective of the teacher is to collect data for the student training, particular care has to be put into its perception capabilities. Taking inspiration from [14], a perception reward is employed in order to keep the next gate always in vision. Mathematically it is described as:

$$R_{perception} = e^{-8 \cdot \theta^3}$$

where θ is the angle created between the drone's reference frame X-axis and the vector which links the drone and the center of the gate in the world's reference frame. Since the camera's FOV is 90 degrees, θ should be less than 45 degrees. In fig. 4.5 a visual representation of the perception reward in function of θ is given. It is notable how the reward starts increasing significantly only after the 45 degrees.

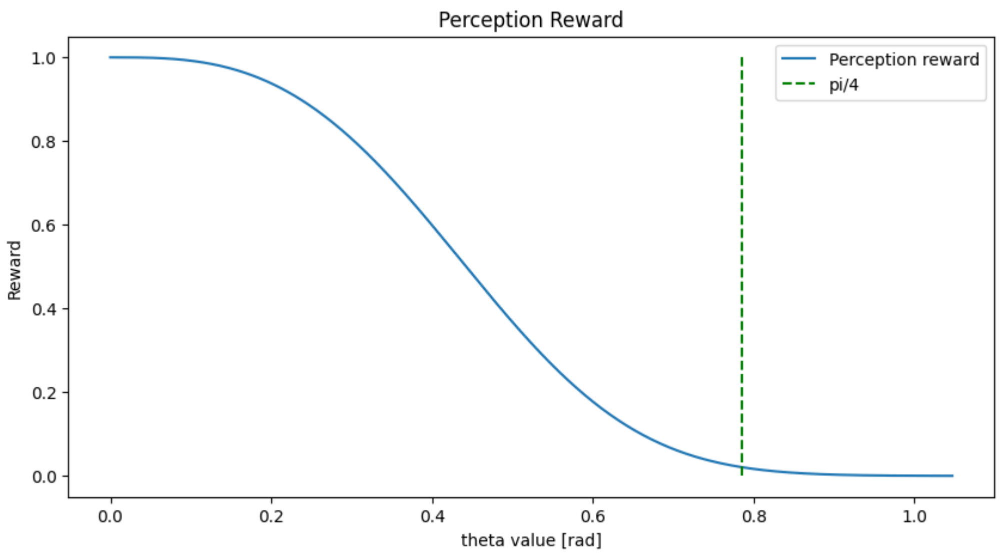


Figure 4.5: Perception reward illustration.

Motion Blur reward

Along with the *perception reward*, another reward concerning the perception capabilities of the drone is introduced in order to reduce the motion blur. This is done by penalizing the agent if big variations in θ (introduced in the perception reward) are made. Mathematically:

$$R_{blur} = e^{-10 \cdot |\theta(t) - \theta(t-1)|}$$

Angular velocity penalty

In order to keep the drone stable, a reward that tries to minimize its angular velocity ω is introduced. It is defined as:

$$R_{ang\_velocity} = ||\omega||_2$$

This penalty is mainly focused on drone stability, however it intrinsically helps also to reduce motion blur. The main difference between this penalty and the motion blur penalty is that, while the angular velocity penalty penalizes *all* angular velocities indiscriminately, the motion blur penalty penalizes only the angular velocities that change the position of the position of the gate in the drone's body reference frame. It does so by penalizing the changes in the angle created between the drone's reference frame X-axis (which represents the camera direction) and the vector which links the drone and the center of the gate in the world's reference frame. This means that the motion blur reward doesn't penalize the agent as long as the gate remains in the same place in the image frame (not creating motion blur).

Smoothness penalty

Since the objective of the teacher agent is to gather data for the student agent, data quality is very important. Because of this, a reward that ensures smooth action trajectories is employed. By minimizing the difference between actions takes in sequent timesteps, a trajectory without noise is promoted. The 'smoothness reward' is defined as:

$$R_{smoothness} = ||a(t) - a(t-1)||$$

4.3.3 Teacher Neural Network Architecture

The observation space presented in sec. 4.3.1 is simple enough to be processed with a small feed forward neural network (sec. 2.3.2). The feed forward neural network is composed of multiple hidden neural network, in particular five hidden layers have been used, with dimensions $[256, 512, 512, 512, 256]$, which is also represented in fig. 4.6. Between each hidden layer a *tanh* activations is used. A *tanh* activation is used on the outputs aswell, which rescales in the range $(-1, 1)$ and is then rescaled before applied in the simulation. The output $a_t = [T, \tau_\phi, \tau_\theta, \tau_\psi]$ is rescaled in the following way:

$$\begin{aligned} T_{rescaled} &= (F/W) \cdot w \cdot \frac{1+T}{2} \\ \tau_{\phi,rescaled} &= s \cdot \tau_\phi \\ \tau_{\theta,rescaled} &= s \cdot \tau_\theta \\ \tau_{\psi,rescaled} &= s \cdot \tau_\psi \end{aligned} \tag{4.7}$$

where $F/W = 1.9$ is the thrust-to-weight ratio of the Crazyflie 2.1 [11], $w = 0{,}264\ N$ is the weight of the Crazyflie 2.1 and $s = 0.001$ is a moment scale factor used to rescale the angular velocity setpoint outputs in the same range as the thrust output (found empirically). This last scaling factor is important as it helps the smoothness penalty to give equal importance to the smoothness of each action.

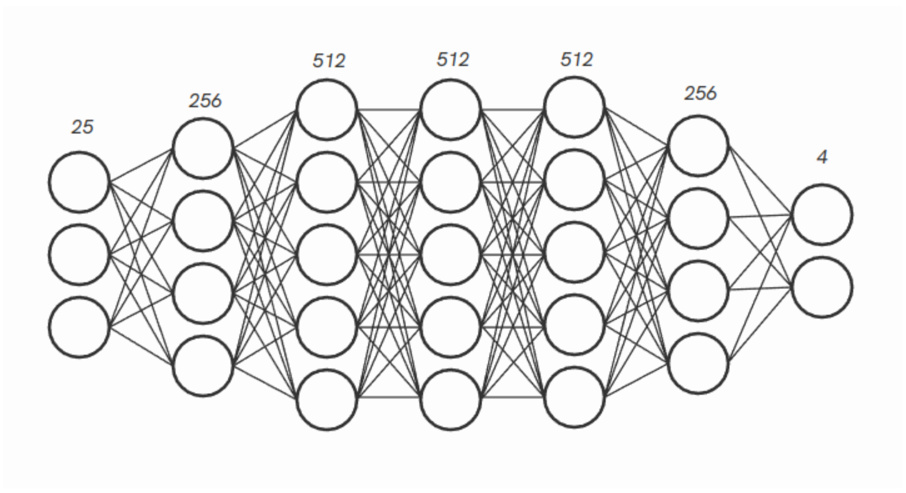


Figure 4.6: Teacher network illustration, with the correct input, hidden and output layer sizes.

4.4 Vision-based Student Agent

The student policy is the main focus of the thesis: it has to be vision-based, has to have good navigation skills and it has to be within the computational constraints that the GAP8 SoC imposes. In this section a more in depth analysis of the student policy will be done: in sec. 4.4.1, the observation space of the student agent will be introduced, from sec. 4.4.2 to 4.4.4, an in depth explanation about the student architecture is done, going over architectural choices, hyperparameters, computational constraints, as well as literature review of tinyCNNs. Following, the sec. 4.4.5, explains the training procedure for the seed network, the sec. 4.4.6, explains the techniques used in order to achieve the domain generalization needed for the student to be able to fly in unknown environments, sec. 4.4.7, explains the NAS optimizations used in order to comply with the computational constraints and finally, in sec. 4.4.8, an overview of the asymmetric actor critic RL fine-tuning is given.

4.4.1 Partial-Information Observation Space

Differently from the teacher policy, the student policy cannot observe the full state of the agent, which was previously called priviledged-information. The student policy will only have partial information provided by the on-board sensors as its observations.

As explained previously (sec. 2.1.2) the AI-Deck provides the CrazyFlie 2.1 the Himax HM01B0, an ULP 320x320 grayscale mono-camera. The image is provided as a matrix of 320x320 with values in the range [0,1]. Mathematically it can be expressed as:

$$\mathbf{i} \in [0,1]^{320\times320}$$

Along with the image provided by the Himax camera, the integrated IMU sensor and the Flow-Deck sensors can provide state-estimation vector. This vector is composed by the 3d linear velocity, 3d linear acceleration, 3d angular velocity and 3d angular acceleration:

$$\mathbf{s}_{est} = [\mathbf{v}, \mathbf{a}, \boldsymbol{\omega}, \boldsymbol{\alpha}] \in R^{12}$$

Thus the observation space of the student policy will be the Cartesian product of the image space and the state-estimation space:

$$\mathbf{o} = [\mathbf{i}, \mathbf{s}_{est}] \in [0,1]^{320\times320} \times R^{12} \tag{4.8}$$

4.4.2 Student Neural Network Architecture

Taking inspiration from [14], the student neural network architecture is composed of three key components: a visual feature extractor, a temporal feature extractor and a regressor.

When using a single camera, the environment becomes a partially observable environment. To this end, a history of observations is used as input, expanding the definition 4.8 to:

$$\mathbf{o}_{input} = [\mathbf{o}_1, \dots, \mathbf{o}_T] \tag{4.9}$$

The **visual feature extractor** is a CNN (sec. 2.3.2) which is commonly used when dealing with image inputs. This part of the network is used to reduce the dimensionality of the images to a single embedding vector. The CNN acts in the same manner on all images, extracting the same kind of information.

In order to lower the computational costs of the following blocks, a further **projector** is attached. Given the vector representation of the image, the projector produces a lower dimensionality vector representation with a fully-connected layer (sec. 2.3.2). This vector is then concatenated with the state estimation vector.

A TCN (sec. 2.3.2) is then used as a **temporal feature extractor** in order to process the history of visual features. Because of the causal way of processing data the TCN employs, only the last step of the resulting sequence of vectors is used and fed into a **regressor**, which in our case is an feed-forward neural network, which produces the network outputs.

An illustration of the architecture is shown in fig. 4.7 and more details about the architectural choices are given in sec. 4.4.4.

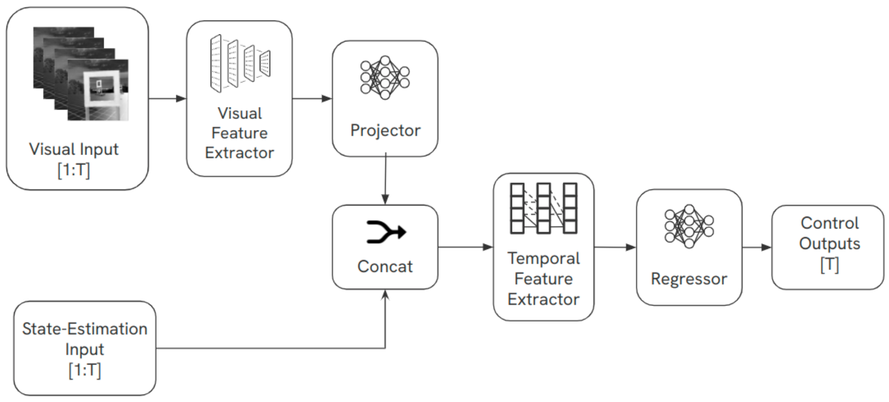


Figure 4.7: The student network architecture used.

4.4.3 Student Architecture Computational Constraints

In order to know how computationally cheap the student policy has to be, knowledge about the used SoC performance capabilities is needed. Theoretically predicting the performance of a SoC in terms of latency is near impossible and empirical analysis have to be done. Fortunately an extensive study of the GAP8 performances can be found in [33], where multiple CNN architectures have been deployed and had their latency and power consumption compared. In this thesis the power consumption is not considered, as it's focus is maximum performance. Instead, one of the hard constraints is to match the camera's acquisition rate in order to fully exploit all the information received. To do so, the maximum number fused multiply-accumulate operation (i.e., one multiplication followed by and addition into an accumulator register, MACs) that our student policy can have can be calculated easily using the data reported in [33]. Let:

- S = the seconds for inference, equivalently it can be expressed as FPS = $\frac{1}{\mathrm{S}}$, the number of inferences per second;
- O = the number of multiply-accumulate operations that the NN needs;
- F = the clock rate at which the GAP8 is working;
- N = the number of operations per cycle that the GAP8 is doing.

and their relationship as:

$$S = O \cdot \frac{1}{N} \cdot \frac{1}{F} \tag{4.10}$$

By using this relationship and the data from [33] (reported in tab. 4.1) it can be found that the avg. MACs per cycle that the GAP8 can achieve at at peak performance (maximum clock frequancy, 175 MHz) is ~3 MACs. Given this information, the number of MACs that the student policy can have in order to match the camera acquisition rate (= 30 FPS) can be computed from eq. 4.10 and it is 17.5 MMACs.

Model [33]	Ops (O)	Clock rate (F)	Inference rate (FPS)	Ops per cycle (N)
160x32	14,1 MMACs	175 MHz	48 FPS	3.867
160x16	4,3 MMACs	175 MHz	110 FPS	2.702
80x32	4,0 MMACs	175 MHz	134 FPS	3.062

Table 4.1: Computation of the operations per clock (N) given the neural network and GAP8 configurations reported in [33], using equation 4.10.

Inference Execution Scheme

In sec. 4.4.2 the student architecture idea has been shown. While representative of the data processing during the training phase, it presents sub-optimalities when talking about the deployment phase. In particular, the continuous reprocessing by the visual feature extractor of the history images lead to wasted computations.

Instead, the deployment strategy would save the vectors representing the limited history needed for the network and use them during inference, while only processing the new image and state estimation. With this scheme, each camera image is processed only once by the visual feature extractor. This is possible as the CNN would always output the same output vector. An illustration of the deployment strategy is shown in fig. 4.8.

When referring to the student's computational cost in MACs from now on, it will be pointing to this configuration in particular.

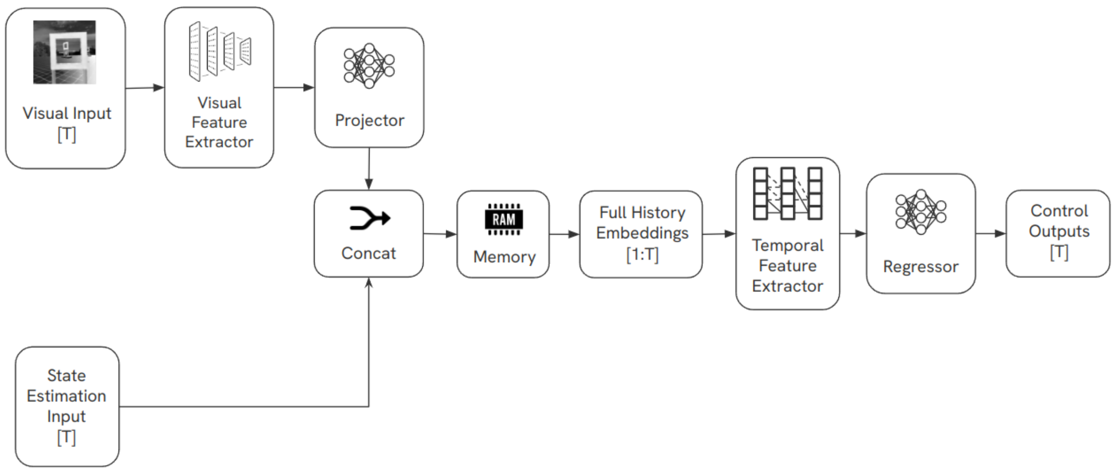


Figure 4.8: An illustration of the student network architecture in a deployment scenario.

4.4.4 Seed Architecture Analysis

Computationally Constrained CNNs Literature Review

Since the student policy is vision-based, the visual feature extractor is the most important part of the architecture. Because of this reason, extensive research went into the choice of this part of the architecture by doing a comparison of the most well known CNN architectures built for edge devices as well as the most well known architectures used for nano-drone tasks.

The objective of the analysis is to decide which tiny CNNs respect the constraints defined in sec. 4.4.3 and which one is the best for the task at hand. By using eq. 4.10 an estimated inference rate on the GAP8 architecture can be found. Another influential aspect was the presence of skip connection as their presence would consume more memory than needed. Even though deployment tools such as DORY [8] allow for efficient use of the memory and load times, the extra memory occupied by the skip connection inference method leads to more memory access, which can reduce inference speed. The analysis is summarized in tab. 4.2.

What the analysis reveals is that in general off-the-shelf pretrained tiny architectures such as variants of MobileNet ([39, 17]) or the nano variants of YOLO architectures [20] are usually way too computationally expensive for this work's purposes, pointing to the CNN architectures built specifically to be deployed on the Bitcraze Crazyflie 2.1 drone, such as DroNet [27] or FrontNet [33] as more promising alternatives.

Furthermore, optimized architectures (either with a hand-made optimization or NAS-optimized) can be reduced in size and computational cost, while maintaining good performances on the tasks proposed. In [23] a hand-made architecture optimization was made on top of the DroNet, building several configurations called Tiny-DroNets. The results show that even with a 8x reduction in model size and eliminating the skip-connections, the resulting model is competitive with the initial DroNet. On the other hand, in [10], a NAS-optimization was done on both the FrontNet and MobileNetV2. In this work, the author shows how the FrontNet architecture (a CNN based on the DroNet architecture) optimized with NAS can still achieve competitive performance w.r.t. the seed network while having half the MACs (7 MMACs vs 14.1 MMACs of the seed model).

Following this analysis, the Frontnet architecture [33] was employed as the visual feature extractor for the architecture described in sec. 4.4.2. Its proven real-time performance on the GAP8 and lack of skip-connections point to it as a promising architectural choice.

Paper	Year	Architecture used	Skip Connection	Operations in MAC	FPS on GAP8	Original deployment target
[39]	2018	MobileNetV2	Yes	300M	1.75	Pixel phone
[17]	2019	MobileNetV3 (Small)	Yes	56M	9.38	Pixel phone
[17]	2019	MobileNetV3 (Large)	Yes	219M	2.40	Pixel phone
[20]	2020	YOLO-v5 NU	Yes	3.85B	0.14	General purpose
[20]	2023	YOLO-v8 N	Yes	4.35B	0.12	General purpose
[20]	2024	YOLO-v10 N	Yes	3.2B	0.16	General purpose
[33]	2021	PULP-FrontNet (80x32)	No	4M	131.25	Bitcraze Crazyflie 2.1 drone
[33]	2021	PULP-FrontNet (160x16)	No	4.3M	122.09	Bitcraze Crazyflie 2.1 drone
[33]	2021	PULP-FrontNet (160x32)	No	14.1M	37.23	Bitcraze Crazyflie 2.1 drone
[27]	2018	DroNet (ResNet-8)	Yes	41.1M	12.77	Parrot Bebop 2.0 drone
[23]	2022	Tiny PULP-Dronets (x0.125, w/o bypass)	No	1.5M	350.00	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x0.25, w/o bypass)	No	4M	131.25	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x0.5, w/o bypass)	No	11.9M	44.12	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x1.0, w/o bypass)	No	39.7M	13.22	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x0.125, w/ bypass)	Yes	1.5M	350.00	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x0.25, w/ bypass)	Yes	4.1M	128.05	Bitcraze Crazyflie 2.1 drone
[23]	2022	Tiny PULP-Dronets (x0.5, w/ bypass)	Yes	12.3M	42.68	Bitcraze Crazyflie 2.1 drone
[7]	2023	Lightweight MobileNetV2	Yes	146.6M	3.58	Bitcraze Crazyflie 2.1 drone
[7]	2023	PULP-FrontNet (160x32)	No	14.1M	37.23	Bitcraze Crazyflie 2.1 drone
[10]	2023	PULP-FrontNet (NAS)	No	7.6M	69.08	Bitcraze Crazyflie 2.1 drone
[10]	2023	MobileNetV2 (Lightweight, NAS)	Yes	7.4M	70.95	Bitcraze Crazyflie 2.1 drone
[10]	2023	MobileNetV2 (Full, NAS)	Yes	12.4M	42.34	Bitcraze Crazyflie 2.1 drone
Paper	Year	Architecture used	Skip Connection	Operations in MAC	FPS	Deployment target

Table 4.2: Analysis of tiny-CNN architectures' performance on GAP8 SoC

Architecture Hyperparameter Choices

Given the network family chosen for the student policy, introduced in sec. 4.4.2, some of their parameters have to be analyzed, as they influence the student policy's performance and computational cost. Particularly, the visual feature extractor choice, the projector output size and the history size are hyperparameters that influence both the performance and the computational cost and cannot be chosen without an in depth analysis.

For what concerns the visual feature extractor a FrontNet architecture was selected. In [33], Palossi et al. proposed three different variations of the FrontNet: 160x32, 160x16, 80x32 (which differ by input size, 160 or 80, and by the number of channels extracted by the first convolutional layer, 32 or 16). Even though each of these have valuable trade-offs, no one can be chosen a priori as the clear best for the task that this thesis faces.

The projector module employed for the dimensionality reduction of the image latent space can have different output space dimensionality. For this analysis, four different values were used: 64, 128, 256 and 512.

The temporal features extractor (in this case, a TCN, sec. 2.3.2) has to make full use of the history provided. Since its receptive field increases with the depth, its depth is depending on the history size used. For simplicity a kernel size of $k = 2$ was used and each layer has a dilation double of the previous' layer, this way the number of layers grows logarithmically w.r.t. the history size. For simplicity the dimensionality of the inner layers of the TCN was kept the same, leaving its optimization to the NAS procedure. For this analysis, different history sizes were used: 1, 4, 8, 16 and 32, meaning respectively 0 (effectively no TCN is employed), 2, 3, 4 and 5 TCN layers.

It is noteworthy that, at this point of the work, the threshold found in sec. 4.4.3 is not to be enforced, yet it is used just as reference. The present analysis searches for the policy that maximizes the performance and shows evident diminishing returns if increased in computational cost.

The analysis was made by training the different architecture configurations on a dataset collected only by the teacher policy via behavior cloning (sec. 2.5.1). The hyperparameter search space is represented in tab. 4.3.

Module	**Hyperparameter**	**Choices**
Visual Feature Extractor	CNN architecture	[FrontNet (160x32, 160x16, 80x32)]
Projector	Output dimensionality	[64, 128, 256, 512]
Temporal Feature Extractor	History size	[0, 4, 8, 16, 32]

Table 4.3: Module Hyperparameters and Choices

4.4.5 Seed Network Training through IL

After training a teacher policy and defining the architectural choices and the hyperparameters of the student policy, a first training via imitation learning is done. Because of the distribution-shift problem (sec. 2.5.2), the student agents trained via behavior cloning in the hyperparameter analysis phase are not able to fly in closed-loop. The DAgger algorithm (sec. 2.5.3) is employed for the student's training in order to alleviate this problem by iteratively, collecting new data in unseen situations for the student agent, letting it explore the environment while the teacher policy indicates the optimal actions.

This training phase has a dual objective: the first is to collect a dataset for the NAS procedure, for which is very important to have exhaustive exploration as the optimized network needs to keep only the most important information, but it is safe to say that a simplified dataset, in which only teacher data is collected will not represent all the nuances of the task. The second objective is to create a dataset which allows for the training of a student policy, which is able to navigate the race track (even if not optimally) in order to allow the RL fine-tuning step to converge. In [51] is shown how RL fine-tuning on a behavior cloning trained student policy is not able to learn the task and the RL exploration leads to catastrophic forgetting, thus this step is crucial for the success of the next steps.

4.4.6 Domain Generalization

In order to ensure the deployment to unseen environments two different domain generalization techniques were applied during the IL training: visual environment randomization and pencil filtering.

Visual Environment Domain Randomization

Data randomization helps reduce overfitting by training neural networks on varied images that keep the same high-level task-related features, enabling them to focus on key elements for the task and ignore superfluous information. In this case, by training on visually diverse environments, the model learns to ignore unimportant details like background changes and concentrate on elements that affect navigation. For this purpose, 30 different backgrounds with varied settings (such as indoor, outdoor, night, day etc.), were used in training. Nvidia IsaacSim's high-fidelity rendering allowed these backgrounds to reflect realistic lighting conditions, making the training data more robust. Some examples of 360 view of these backgrounds are shown in fig. 4.9.

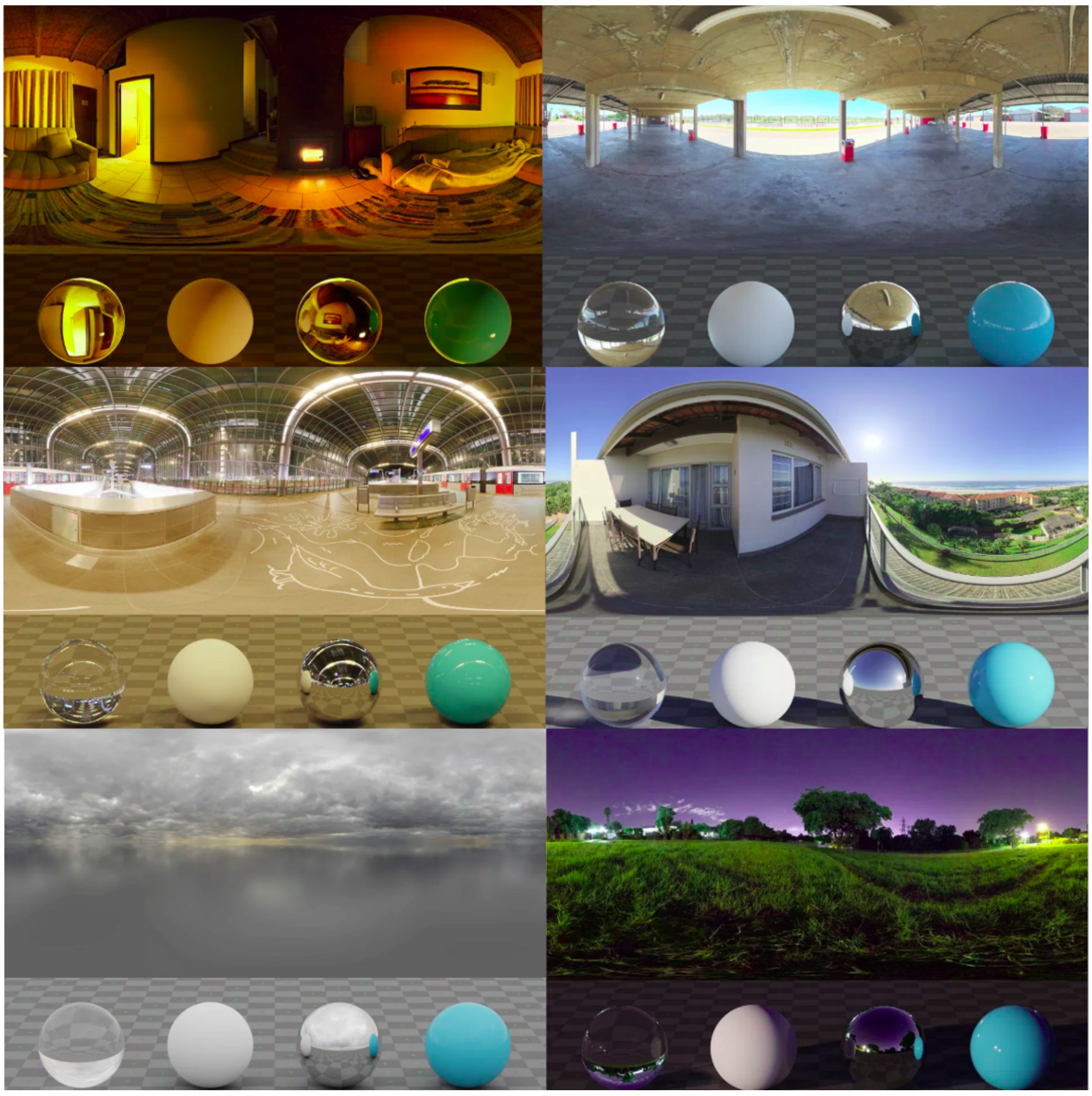


Figure 4.9: Some of the visual environments the student agent sees during training as 360 views.

Pencil Filter

The pencil filter is valuable for autonomous drone racing as it enhances the visibility of key structural features, like gate edges, while reducing the influence of varying lighting conditions and motion blur that often occur in high-speed flight. By converting images to grayscale and emphasizing outlines, the pencil filter helps the perception model focus on shapes rather than colors, making it more robust to environmental changes. This approach improves the model's ability to detect gates accurately in both bright and dim settings and allows for a smoother transfer of training from simulation to real-world conditions. The pencil filter helps bridge the gap between simulated training environments and real-world racing, enabling consistent, robust performance across different visual settings [34]. A visual comparison between simulation and reality can be seen in fig. 4.10, while in fig. 4.11 and 4.12 different scenarios are presented in three different image representations: RGB, Grayscale and Pencil-Filter.

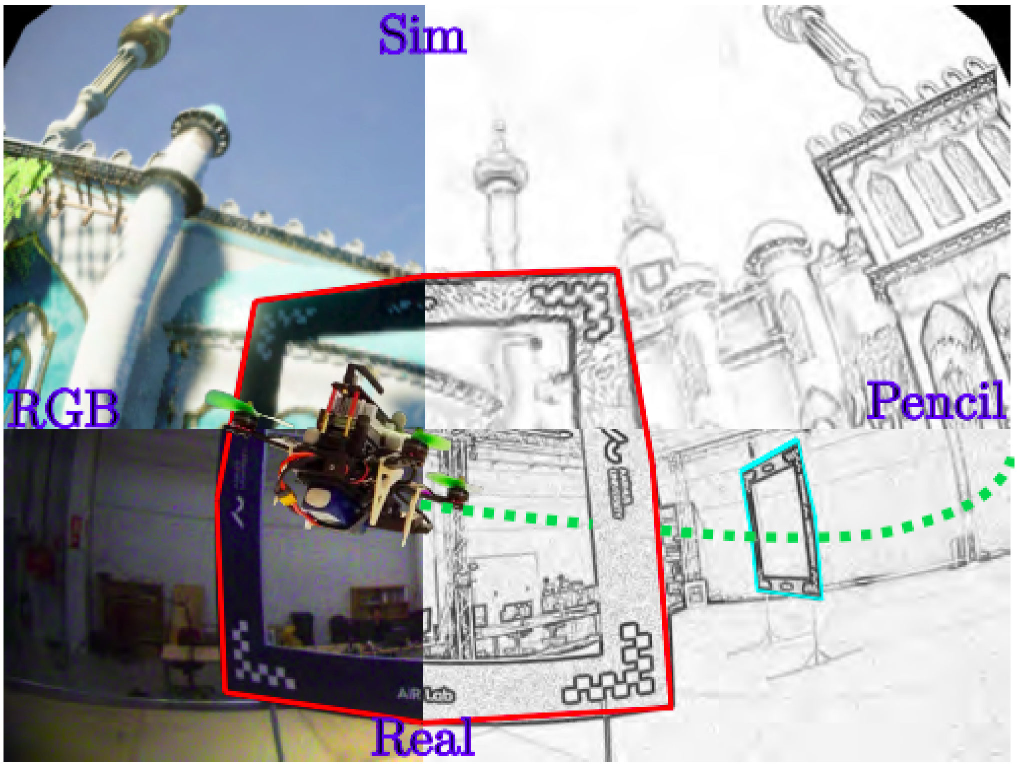


Figure 4.10: Visual explanation of the pencil filter idea presented in [34]: closing the visual sim-to-real gap by narrowing the discrepancy between different brightness images and blurred images.

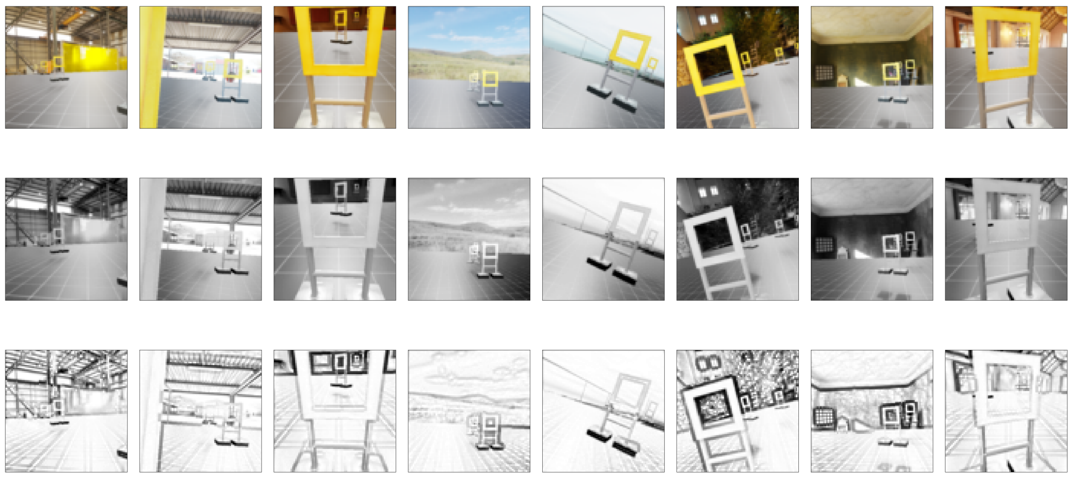


Figure 4.11: Different images. The first row is the RBG image, the second row is the respective grayscale image, the third row is the respective pencil-filtered image.

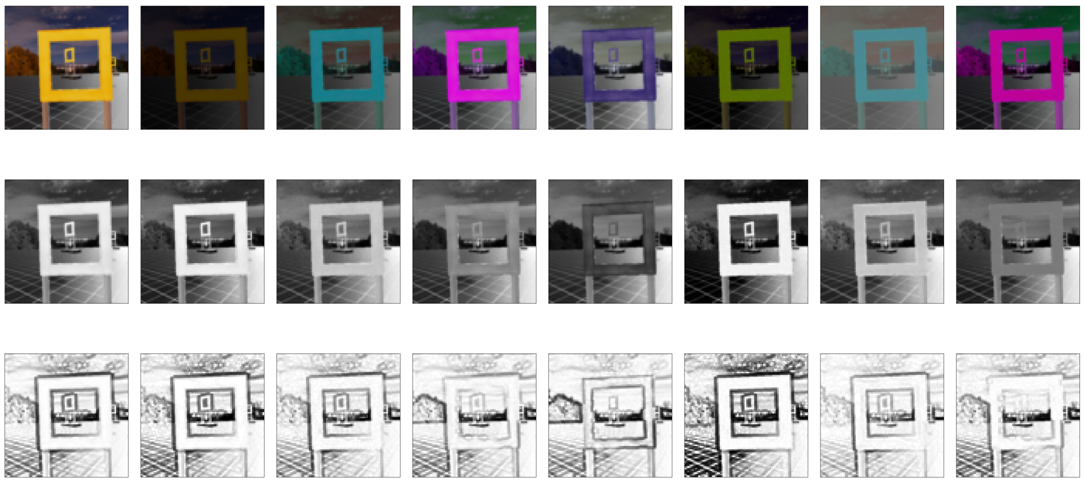


Figure 4.12: The same image (the first column is the original), with different random augmentations. The first row is the RBG image, the second row is the respective grayscale image, the third row is the respective pencil-filtered image.

4.4.7 Student Neural Architecture Optimization

Until this step, the computational threshold found in sec. 4.4.3 have been taken just as reference for the architecture analysis. In order to be deployed on the Bitcraze Crazyflie 2.1 and achieve real-time performance a further neural network architecture optimization is needed. Here a NAS optimization is employed, in particular the PIT algorithm [36] was used in order to prune the network. The way that PIT works is by creating a second set of parameters θ, which are in the range (0,1). PIT creates one parameter in θ_i for each channel/neuron W_i in the in network to be optimized, and the θ_i parameters are used to hold or discard W_i. The θ parameters, in practice, behave like masks which create sub-networks starting from the neural network to be optimized (seed network), in the following way:

$$W_\Theta = W \odot H(\theta)$$

where θ is the vector of trainable mask parameters, $\odot$ is the Hadamard product (also known as element-wise product), and H is a Heaviside step function used to binarize θ. Both θ and W are optimized during training by minimizing the following loss function:

$$\min_{W_\Theta,\theta} \mathcal{L}(W_\Theta;\theta) + \lambda\mathcal{R}(\theta) \tag{4.11}$$

where:

- $\mathcal{L}(W_\Theta;\theta)$ is the task loss, in this case the MSE loss between the network's outputs and the teacher policy output, on the aggregated dataset gathered in the previous step;
- $\lambda\mathcal{R}(\theta)$ is the NAS loss, which represent the cost of the network, weighted by the parameter λ. In this case the cost of the network is represented by the number of MACs it does;

By employing a loss which consists both of the task loss and the NAS loss, PIT optimizes the neural network in order to achieve the trade-off between task performance and computational cost.

Given the explicit computational constraints found in sec. 4.4.3, the DUCCIO regularizer [9] is employed to ensure that the computational target is reached. The DUCCIO regularizer changes the $\mathcal{R}(\theta)$ term of eq. 4.11 of the optimization loss to:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \sum_{j=0}^{J} \lambda_j \max(0, \mathcal{R}_j(\theta) - \mathcal{T}_j) \tag{4.12}$$

which allows for multiple costs to be used and ensures that all of them are within the target ($\mathcal{T}_j$) by subtracting it from the cost and enforcing only positive values (which means that the cost optimization stops when it reaches the target). In this case only one cost is used: the number of fused multiply and accumulate operations (MACs), which has as a target 17.5M MACs as found in sec.|4.4.3. Moreover, the DUCCIO regularizer employs a regularizer weight λ_j annealing, which ensures that the target $\mathcal{T}_j$ is achieved.

The PIT algorithm is applied to the entire network trained with IL (which in this context is called seed network) in the previous step. Following the training procedure presented in sec. 2.7.3 there is a three step procedure that has to be followed in order to achieve optimal results from the NAS optimization: the warm-up phase, the search phase and the fine-tuning phase. The *warm-up phase*, is not needed as the seed network is already pretrained by the IL procedure. The *search phase* optimizes both W and θ with the loss in eq. 4.12. During this phase the optimization algorithm explores the search space of θ which creates all the sub-networks derivable by the seed network. Given the optimal θ^*, the sub-network W_{Θ^*} is extracted from the seed network, effectively discarding all the channels/neurons for which $\theta_i < 0.5$. Finally the *fine-tuning phase* is performed. This last step is essential as it focuses only on the regression task performance by optimizing only the task loss $\mathcal{L}(W_{\Theta^*})$, allowing for the maximum performance to be achieved by the extracted network.

4.4.8 Asymmetric Actor-Critic RL Fine-tuning

The RL fine-tuning is a key step in order to ensure the best performance achieved on the NAS optimized architecture obtained in the previous step. As explained in [51], the DAgger algorithm does alleviate the distribution-shift problem, but it doesn't fully solve it, as the agent may still find itself in situations which have not been covered during the training via the teacher exploration. The RL fine-tuning, on the other hand, cannot converge when training an agent which is fully untrained (not bootstrapped) due to the higly dimensional input space that the navigation from images task has.

In order to better guide the RL fine-tuning training to the convergence, the **asymmetric actor-critic** (sec. 2.4.5) is employed. This framework is based on the classic actor-critic (sec. 2.4.3 framework, which uses the critic value estimates in order to better guide the policy updates by optimizing the loss:

$$L(\theta) = E_{(o,a)\sim\pi_\theta}\left[\log \pi_\theta(a|o)\hat{A}(o,a)\right], \text{ where } \hat{A}(o,a) = \hat{Q}^\pi(o,a) - \max_a \hat{Q}^\pi(o,a) \tag{4.13}$$

where $L(\theta) = E_{(o,a)\sim\pi_\theta}[\log \pi_\theta(a|o)]$ is the log-likelihood, then weighted by $\hat{A}$ which is the estimated advantage function. $\hat{A}$ is the difference between $\hat{Q}^\pi(o,a)$ (the cumulative reward , given by observing o and taking the action a, under the policy π) and $\max_a \hat{Q}^\pi(o,a)$ (the estimated cumulative sum of rewards, given by observing o and takin the best action under the policy π, also known as *value function* $V^\pi(o)$). Weighting the log-likelihood by the advantage function, leads the policy to maximize the probability of seeing again actions that maximize the cumulative sum of rewards. The guidance that $\hat{A}$ gives to the optimization function, however, is limited by the accuracy of its estimations, which are given by the critic (value network). Since the critic suffers from the same convergence issues as the actor, the asymmetric actor-critic framework uses a state-based input critic which computes estimate values using priviledged state inputs $\hat{Q}^\pi(s)$ instead of partial observations, such as images. By employing this kind of value network, better estimates are computed, which leads to a more stable training procedure by optimizing:

$$L(\theta) = E_{(o,a)\sim\pi_\theta}\left[\log \pi_\theta(a|o)\hat{A}(s,a)\right], \text{ where } \hat{A}(s,a) = \hat{Q}^\pi(s,a) - \max_a \hat{Q}^\pi(s,a) \tag{4.14}$$

In this case, the NAS-optimized network is used as a bootstrapped (pre-trained) actor and the teacher's value network. However, a straightforward plug-and-play approach may not yield optimal results as the critic function requires interactions to adapt the pre-trained actor, necessitating a “warm-up” process. Once the critic warm-up is done, the training of the asymmetric agent is the same as the teacher's: using the same reward function and the same environment specifics.

In order not to occur into catastrophic forgetting w.r.t. the domain generalization capabilities acquired in the previous training steps, environment is set to change every 100 steps during training. This way the RL agent has to learn to navigate from images regardless of the visual environment.

Chapter 5

Experiments and Results

This chapter is devoted to reporting and presenting the results obtained. In sec. 5.1, a brief explanation of the experimental setup is given, including package information, simulator version, etc.. In sec. 5.2, an introduction to the assessment methodology is reported, including how the results are gathered w.r.t. closed-loop policy performance. In the next sections (5.3 and 5.4), the main results of the method are shown, both for the teacher and for the various student versions. Finally sec. 5.5, gives a summary of the results and a discussion of possible developments.

5.1 Experimental Setup

During the whole work, the Python programming language was used in the version 3.12.4. The libraries used are listed in tab. 5.1 along with their versions. The simulator used was IsaacSim 4.1.0, while IsaacLab, the python wrapper that enables robot learning modules to work (such as SKRL [42]), is version 1.2.0.

Package	**Version**
gymnasium	0.28.1
numpy	1.26.0
omni-isaac-lab	0.19.4
omni-isaac-lab-assets	0.1.3
omni-isaac-lab-tasks	0.7.10
pandas	2.2.2
plinio	0.0.1
pytorch-tcn	1.1.0
scipy	1.10.1
skrl	1.2.0
torch	2.2.2+cu118
torchaudio	2.2.2+cu118
torchinfo	1.8.0
torchvision	0.17.2+cu118
torchviz	0.0.2

Table 5.1: Python package versions.

5.2 Policy Performance Assessment Method

Since the proposed method yields multiple policies, which differ by their training method and compliance with the constraints imposed by the problem, a way to assess the performance of each version of the policy in an equal manner is needed.

The method chosen is evaluating the closed-loop performance of the policies on 100 different randomly generated tracks. To ensure equal difficulty a starting seed equal to 319029 is set. Because of the IsaacSim graphical rendering, which employs monte-carlo techniques in order to simulate realistic lighting, the seed sequentiality is not guaranteed between a track generation and another. This led to the seeds to be enforced at each new track generation, incrementing them linearly from 319029 to 319129.

For vision-based policies different tests were done:

- **Static visual environment test**: the first test was done in only one visual environment encountered during the training;
- **Random visual environment test**: the second test was done changing the visual environment by picking randomly among the ones seen during training (the same track will always have the same visual environment);
- **Unknown visual environment test**: the last test was done in a never-seen-before visual environment.

Closed-loop Metrics

The task related closed-loop metrics are used to assess the quality of a policy. The metrics used are:

- **Average episode reward**: shows the average cumulative reward on a track run (episode). It gives more continuous information, but it includes behavioral terms which produce noise in the measurement of objective accomplishment.
- **Average track completion**: shows the average number of gates passed over the total on the track. It gives less information about the agent behavior, but maintains the task-relevant information, with a detail which remains high.
- **Episode success rate**: shows the percentage of track runs (episodes) in which the agent successfully finishes the track. This last metric gives no information about the policy behavior and has less detail, yet it is important to show how reliable is the policy.

5.3 Priviledged Teacher Agent Results

5.3.1 Teacher training through RL

The objective of this experiment is to build an optimal teacher policy which uses priviledged information in order to navigate successfully the track. This policy, although unusable in a real scenario due to the lack of perfect information, is used to gather data in terms of optimal actions to use as ground truth in training the student network.

The teacher policy training was done entirely using RL. This was achievable because of a low-dimensionality of the input. By using a number of future gate observations $N = 3$ the input space was $\mathbf{s} = R^{25}$ (sec. 4.3.1).

The algorithm used for the RL training was PPO (sec. 2.4.4), implemented by the SKRL library [42]. Thanks to IsaacSim's ability to leverate GPU computation power, a high parallelization of the environments, more precisely 4096 parallel environments were used. A total of 500'000 rollouts (steps) per environment were used during the entire training procedure. Each policy update consisted of 250 rollouts per environment, thus each policy update consisted of $250 \cdot 4096 \approx 1M$ data points. On each update, the whole dataset was divided in 4 batches and learned for 5 epochs. The learning rate was scheduled with a PPO-specific scheduler, which uses the KL divergence of the policy to adjust the learning rate (KLAdaptiveLR, see [42]). Finally, the scaling λ hyperparameters introduced in sec. 4.3.2 of the reward function used are reported in tab. 5.2.

Reward Component	**Scale**
Passed Gate	20
Contact	10
Progress	0.3
Perception	0.15
Safety	1.0
Motion Blur	0.01
Action Smoothness	0.03
Angular Velocity	0.001

Table 5.2: Reward Scales for Different Components

Training Results

As shown in fig. 5.1 and 5.2, which shows the training progression of teacher policy during the first 100’000 rollouts, it converges to a stable constant way before around the 40’000 rollouts mark, which is only 8% of the entire training budget. By inspecting also fig. 5.3, which displays the progression of the crash penalty during the training, it is obvious that the policy learned how to finish episodes without crashing in the majority of the episodes at the 40’000 rollouts mark. However, the reason for the large number of training rollouts is to be found in behavioral rewards, which have smaller scale, thus lower importance in the overall training procedure. One evident example of such rewards is the smoothness reward, which penalizes the difference between consecutive timesteps’ actions. This last reward’s progression over the entire training (500’000 rollouts) is shown in fig. 5.4. Here is clear how this reward finds a constant stable value only around the 400’000 rollout mark.

Closed-loop Results

In order to have a baseline to compare the vision-based policies with, the policy performance assessment (described in sec. 5.2) was done also for the teacher policy. The results are shown in tab. 5.3. Here only one analysis was done, as changing the visual environment would not make any difference to the teacher policy since it doesn’t use visual information.

It is clear, from these results, how the teacher policy can be considered as the upper-bound for our next experiments, as it shows excellent results, reaching 99% of successful episodes, by using ground-truth priviledged information, which are not obtainable in reality.

Policy	**Environment**	**Average episode reward [-]**	**Average track completion [%]**	**Episode success rate [%]**
Teacher	—	535 ± 50	99%	99%

Table 5.3: Closed-loop performance of the teacher policy.

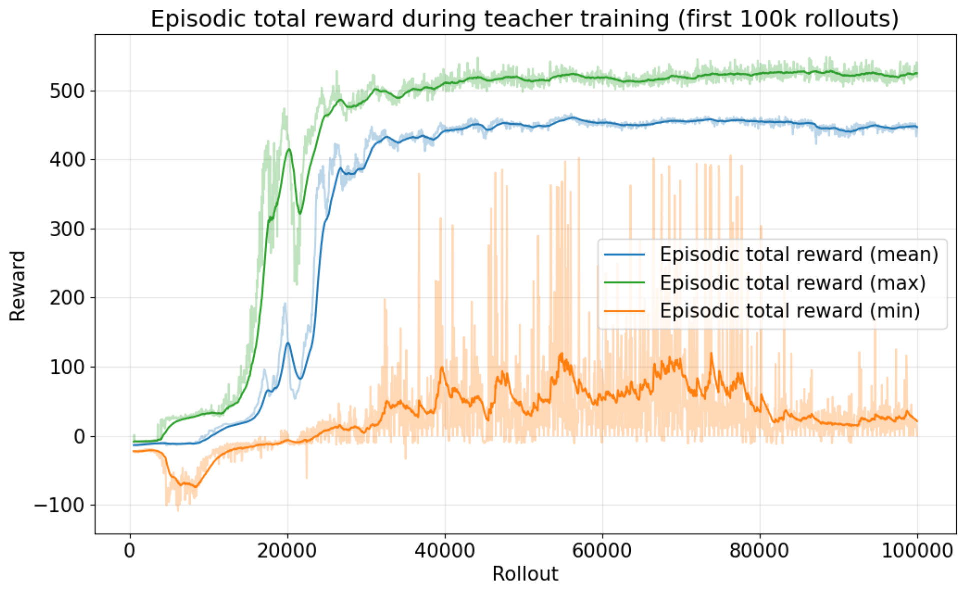


Figure 5.1: Teacher policy's episodic total reward progression during the first 100k rollouts of the training.

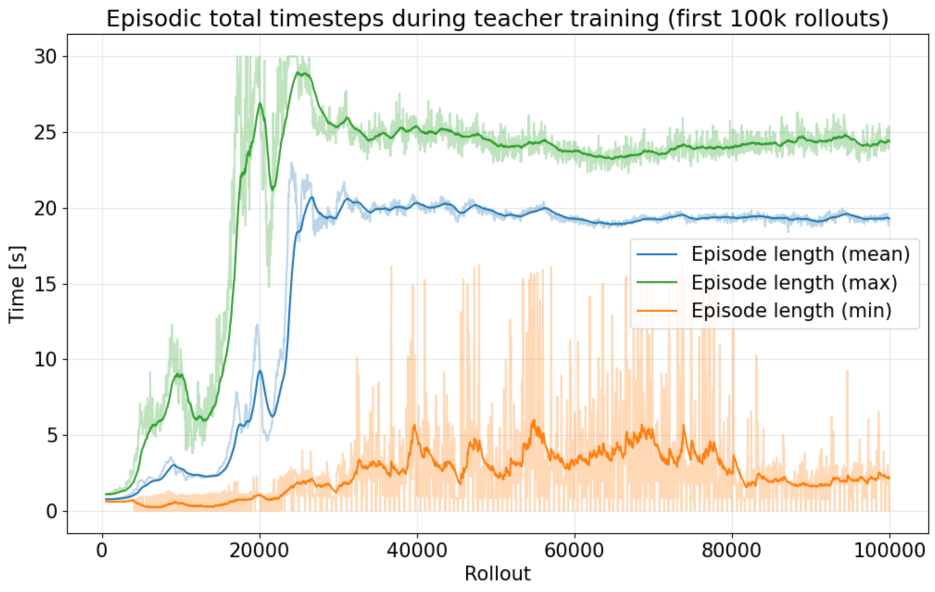


Figure 5.2: Teacher policy's episod length progression during the first 100k rollouts of the training.

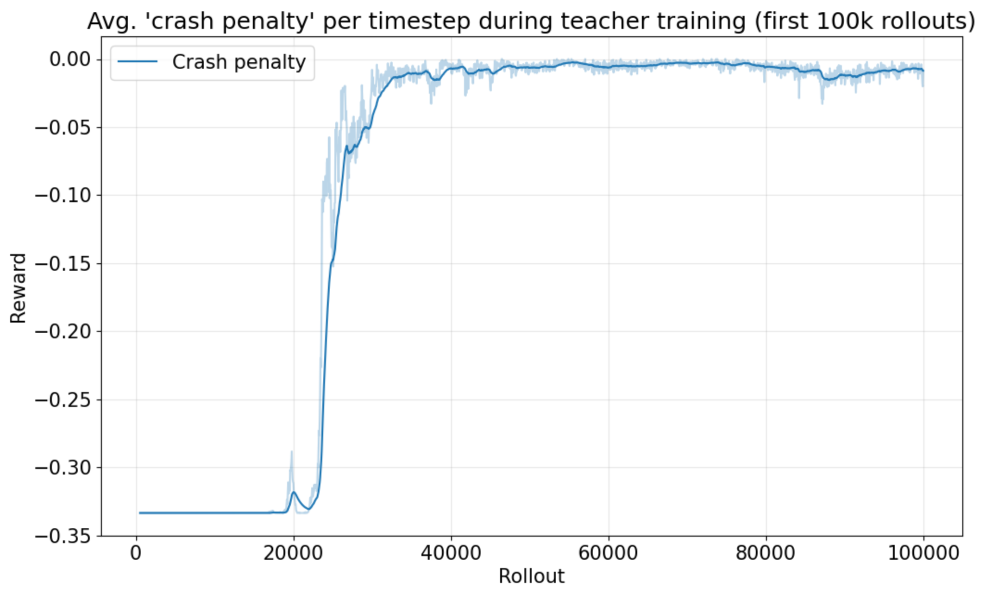


Figure 5.3: Teacher policy's crash reward progression during the first 100k rollouts of the training.

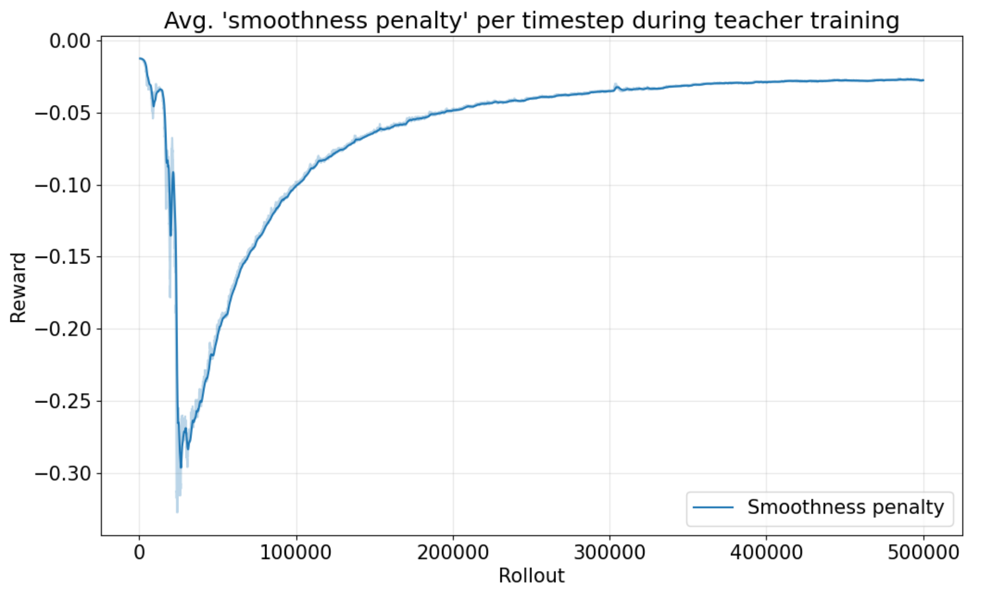


Figure 5.4: Teacher policy's smoothness reward progression during the entire training (500'000 rollouts).

5.4 Vision-based Student Agent

5.4.1 Seed Architecture Exploration

With the next set of experiments, a first analysis of the student architecture is done. Given the family of networks presented in sec. 4.4.2, the objective is to find the best network in terms of *performance vs computational cost trade-off*. The networks are chosen by changing their hyperparameters, chosing from the combinations presented in sec. 4.4.4, for a total of 60 runs. Note that, for this set of experiments, the computational threshold introduced in sec. 4.4.4 is not used as a hard constraint, but only as a reference, thus the network chosen at this point will be the one which shows evident diminishing returns in terms of performance if the computational cost is increased.

In order to train the various networks a dataset was created by using the teacher policy. The training dataset had 100'000 datapoints, of which half was created by using the optimal teacher, while the other half was created by adding small noise to the teacher's actions in simulation. The second half was introduced in order to have a more representative dataset by including also non-optimal trajectories. The dataset gathering was done in 30 different visual environments, in order to apply visual domain randomization on the network's training.

The training of the networks was done giving the history of images (or the simple image if the history was equal to one) and the history of state estimations as input and the teacher's actions as output. The images were preprocessed with the pencil-filter in order to improve the generalization capabilities of the network. Since the actions can range on different effective spans, the outputs learned by the network were normalized. This was done in order to ensure that all the outputs had the same influence over the loss. As the output of the teacher is continuous, thus we used the Mean Squared Error (MSE) loss between the network outputs and the teacher actions in order to model the problem as a regression. In order to evaluate the performance of the network the R2 score between the network's outputs and the teacher actions. The R2 score shows how well a regression model explains the variability in the data, ranging from 0 (no fit) to 1 (perfect fit).

The training was done over a maximum of 30 epochs, with a early-stopping policy that would stop the training if more than a total of 4 epochs (non necessarily consecutive) did not improve the validation results during the training *and* the last epoch also did not improve the validation results. With this training procedure the training would stop only if the performance is not improving, while tolerating some variance in the performance during the first epochs. An exponential learning rate scheduler was also employed, reducing the learning rate of a factor of 0.1x every 3 epochs and starting with a learning rate of 0.01. Finally a batch size of 128 was used, where each data point is a history of images and state estimates.

Results

The results analyse the choice of the history size, the projector size and the feature extractor size. It is done by presenting the results both by showing the complete *performance vs cost* graphs in which the pareto-front is visible (fig. 5.5, 5.6 and 5.7) and the tables in which the result are grouped by and averaged by the different hyperparameters (tab. 5.4, 5.5 and 5.6).

- **History size**: in this case tab. 5.4 clearly shows that more history yields better results, however the number of MACs grows way faster than the R2 score. Particularly, the growth from history size 16 to history size 32 almost doubles the number of MACs needed, while only improving the performance by 0.08 of R2 score on average. This leads to the conclusion that going beyond history size 32 would not have a significant improvement in performance w.r.t. the increase in computational cost.

- **Projector size**: it is clear from both tab. 5.5 and fig. 5.6 how the jump from projector size 64 to projector size 128 is high in terms of performance but very low in terms of computational cost, whereas the jump from projector size 256 to projector size 512 is not that impactful on the performance, but shows an increase in the number of MACs which is more than double. This leads to the conclusion that getting a bigger projector size would not improve performances w.r.t. the increase in computational cost.

- **Visual feature extractor**: in this case, since there is no actual continuity between the various choices, the table does not help in analyzing this choice. On the other hand, by inspecting fig. 5.7 it is clear how the frontnet 80x32 is the one that is more frequently present on the pareto front, whereas the frontnet 160x32 never manages to be present in the pareto front. The frontnet 160x16, on the other hand, has similar performance and cost to the frontnet 80x32, but is less present on the pareto front, which leads to the conclusion that the 80x32 configuration is the most well suited for the task.

With the considerations made above, the choice for the seed architecture for the student policy would be the one that yields the best performance, so in this case the configuration would be: (history size = 32; projector size = 512; visual feature encoder = frontnet (80x32)), which has a performance of 0.95 R2 score and a computational cost of $1.83 \cdot 10^8$ deployment MACs.

History size	Avg. R2 score	Avg. Deployment MACs
1	0.878	$1.29 \cdot 10^7$
4	0.883	$1.81 \cdot 10^7$
8	0.909	$2.68 \cdot 10^7$
16	0.913	$4.25 \cdot 10^7$
32	0.921	$8.19 \cdot 10^7$

Table 5.4: Seed architecture exploration results: average results grouped by history size

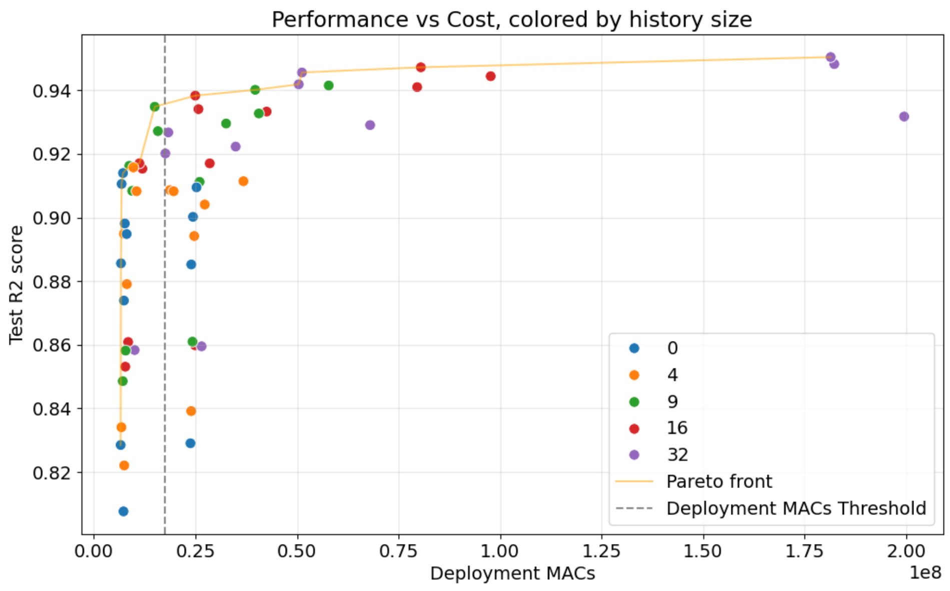


Figure 5.5: The seed architecture hyperparameter analysis pareto-front, colored by history size

Projector size	Avg. R2 score	Avg. Deployment MACs
64	0.844	$1.36 \cdot 10^7$
128	0.904	$1.71 \cdot 10^7$
256	0.923	$3.06 \cdot 10^7$
512	0.928	$8.44 \cdot 10^7$

Table 5.5: Seed architecture exploration results: average results grouped by projector size

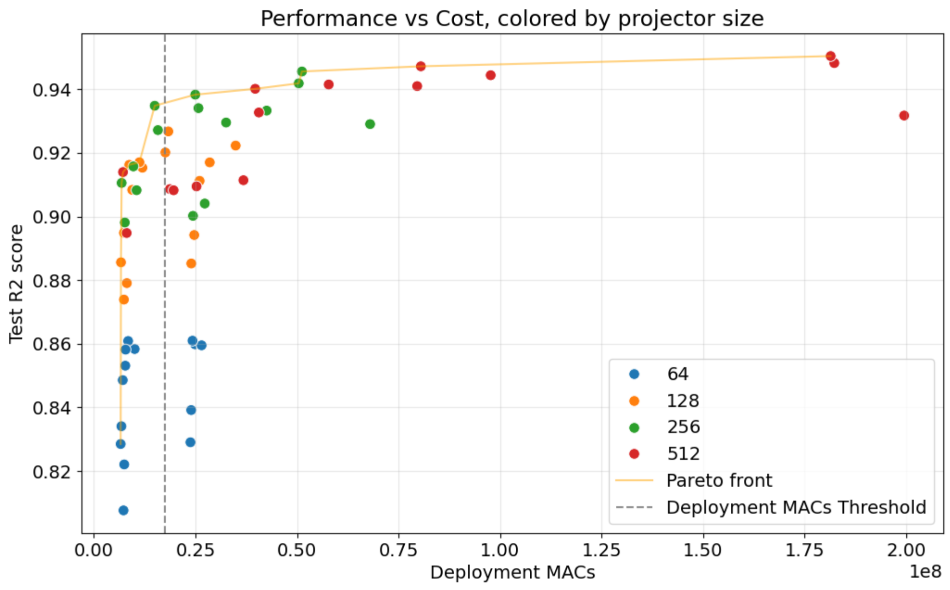


Figure 5.6: The seed architecture hyperparameter analysis pareto-front, colored by projector output size

Feature extractor	Avg. R2 score	Avg. Deployment MACs
FrontNet (160x32)	0.897	$4.79 \cdot 10^7$
FrontNet (160x16)	0.900	$3.11 \cdot 10^7$
FrontNet (80x32)	0.904	$3.03 \cdot 10^7$

Table 5.6: Seed architecture exploration results: average results grouped by visual feature encoder

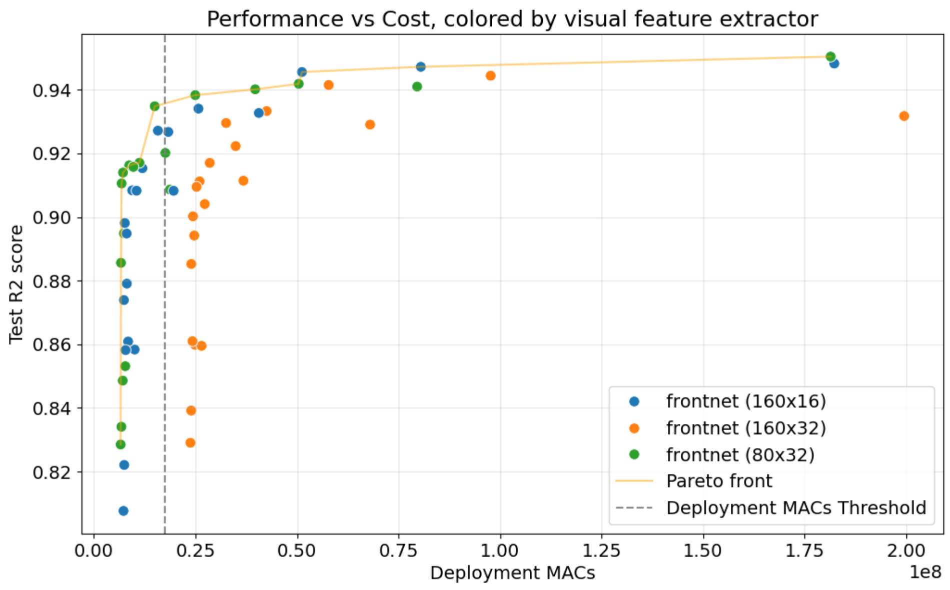


Figure 5.7: The seed architecture hyperparameter analysis pareto-front, colored by visual feature encoder

5.4.2 Seed Network Training thorugh IL

This set of experiments aim at training a student policy through the DAgger algorithm (sec. 2.5.3). As explained in sec. 4.2.2, this student policy does not have to be optimal: its main purpose is to generate a comprehensive dataset for the NAS step. The dataset needs to be comprehensive enough to generate a policy which will not lead to catastrophic forgetting in the RL fine-tuning step due to exploration. This aspect can be evaluated by analyzing the student policy closed-loop performance, as a policy trained on a dataset which is not robust enough, would not be able to fly.

The computational budget given to this experiment was of 15 DAgger iterations. Each iteration is composed of data gathering and retraining. Each data gathering collected 50'000 data points, with a policy which randomly chose between the last iteration's student policy and the teacher policy. During the DAgger process the probability of choosing the teacher decreases exponentially as 0.5^{iter}. The training part of each iteration is done on the aggregated dataset, in the same manner described in the previous experiment. Each training starts from the policy trained at the previous step and the starting learning rate is is linearly decreased from 0.01 to 0.001 in the first 8 epochs and then kept constant for the rest of the DAgger algorithm. The previous experiment training is to be considered as the first DAgger iteration.

Results

In fig. 5.8, 5.9 and 5.10 the progress of the seed network training on the closed-loop evaluation is shown. As expected, the closed-loop evaluation of the first iteration shows that the agent produces 3.19 ± 10.15 as a reward. When compared with the teacher policy closed-loop evaluation from tab. 5.3, it becomes obvious that the student agent cannot follow to the optimal trajectory shown by the teacher policy during the data collection yet, as it deviates too much from it and doesn't know how to recover. Only after the 5th iteration the agent shows signs of improvement. When looking at the last iteration, however both the open-loop and the closed-loop results show promising results. The open loop performance of the last iteration can be analysed in tab. 5.7, where the regression R2 score on all four the commands on the test set are shown. Given the average R2 score of 0.948 (where a value of 1.0 means that the network perfectly replicates the teacher's actions), it is clear that the open-loop performance are very good. The last iteration closed-loop evaluation, which is presented in tab. 5.8, shows that the student policy it reaches a 9% successful run rate in a known visual environment and on average it reaches 40% of the track before colliding either with a gate or with the floor. In fig. 5.11 a comparison between the teacher commands and the seed network commands is shown in a successful episode where the drone is fully controlled by the student policy. It can be seen that the error between the teacher's actions and the student's actions is small and, whenever the error increases, the student is able to recover. Given the closed-loop results it can be claimed that the student policy has learned to navigate the race tracks, meaning that the dataset is robust enough to train policies which will not lead to catastrophic forgetting during the RL fine-tuning step.

Moreover, these results show how the IL methods, differently from RL methods, are sample efficient, allowing the student policy to learn to navigate the race tracks, despite the highly dimensional input space. On the other hand the policy is still not optimal, due to the distribution-shift (sec. 2.5.2), as it doesn't allow for an extensive exploration from the agent, which is allowed by the RL methods. With this considerations we can justify the need for a RL fine-tuning step.

Command setpoint	**R2 score**
Thrust command setpoint	0.958
Roll command setpoint	0.930
Pitch command setpoint	0.964
Yaw command setpoint	0.940
Average	0.948

Table 5.7: Open loop performance of the seed network policy on the test set.

Policy	**Environment**	**Average episode reward [-]**	**Average track completion [%]**	**Episode success rate [%]**
Seed	Static	177 ± 134	40%	9%
	Randomized	180 ± 125	40%	9%
	Unknown	145 ± 110	33%	3%

Table 5.8: Closed-loop performance of the seed network policy.

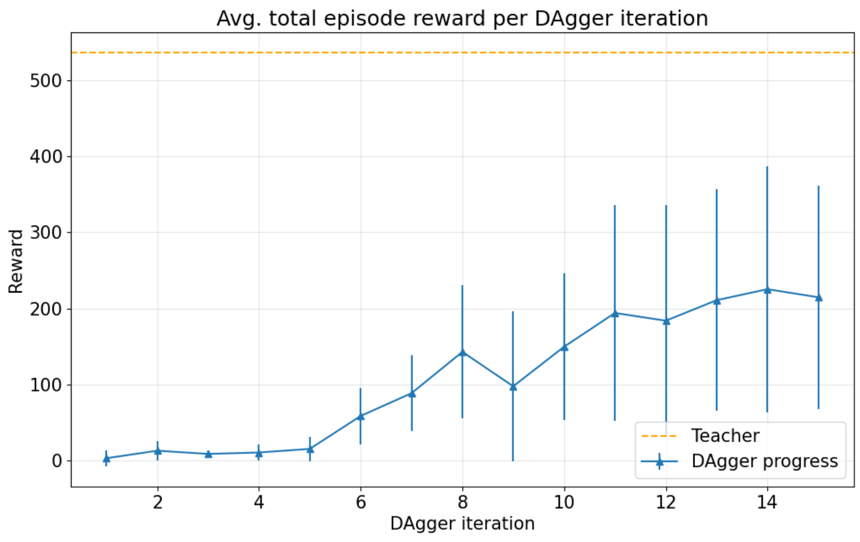


Figure 5.8: Avg. episodic reward progress for DAgger policies.

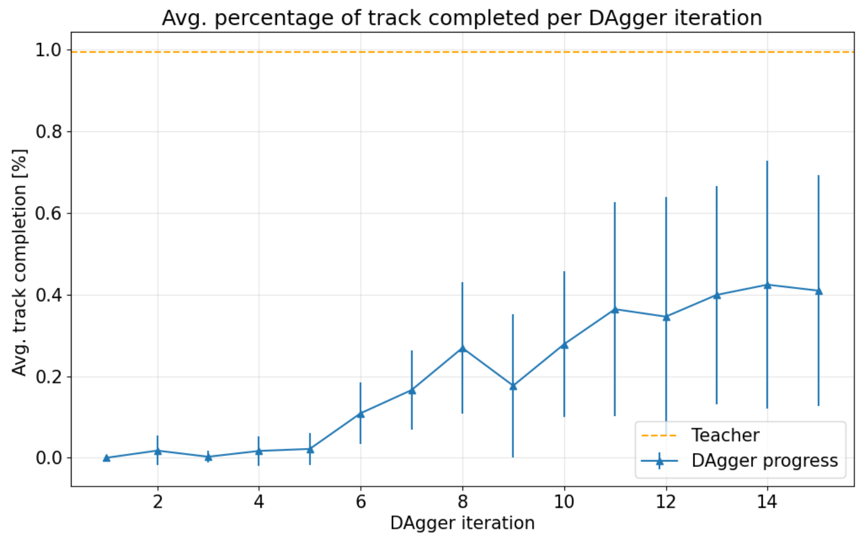


Figure 5.9: Avg. percentage of episodic reward progress for DAgger policies.

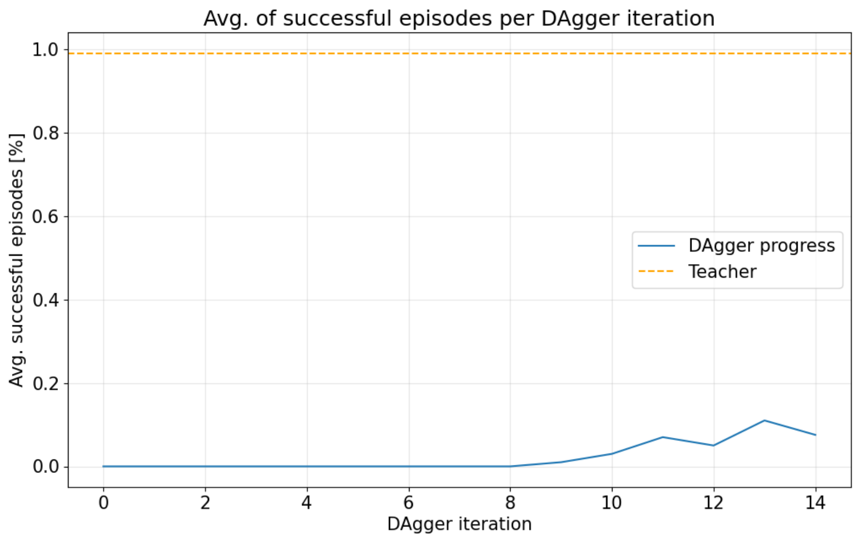


Figure 5.10: Avg. successful episodes for DAgger policies.

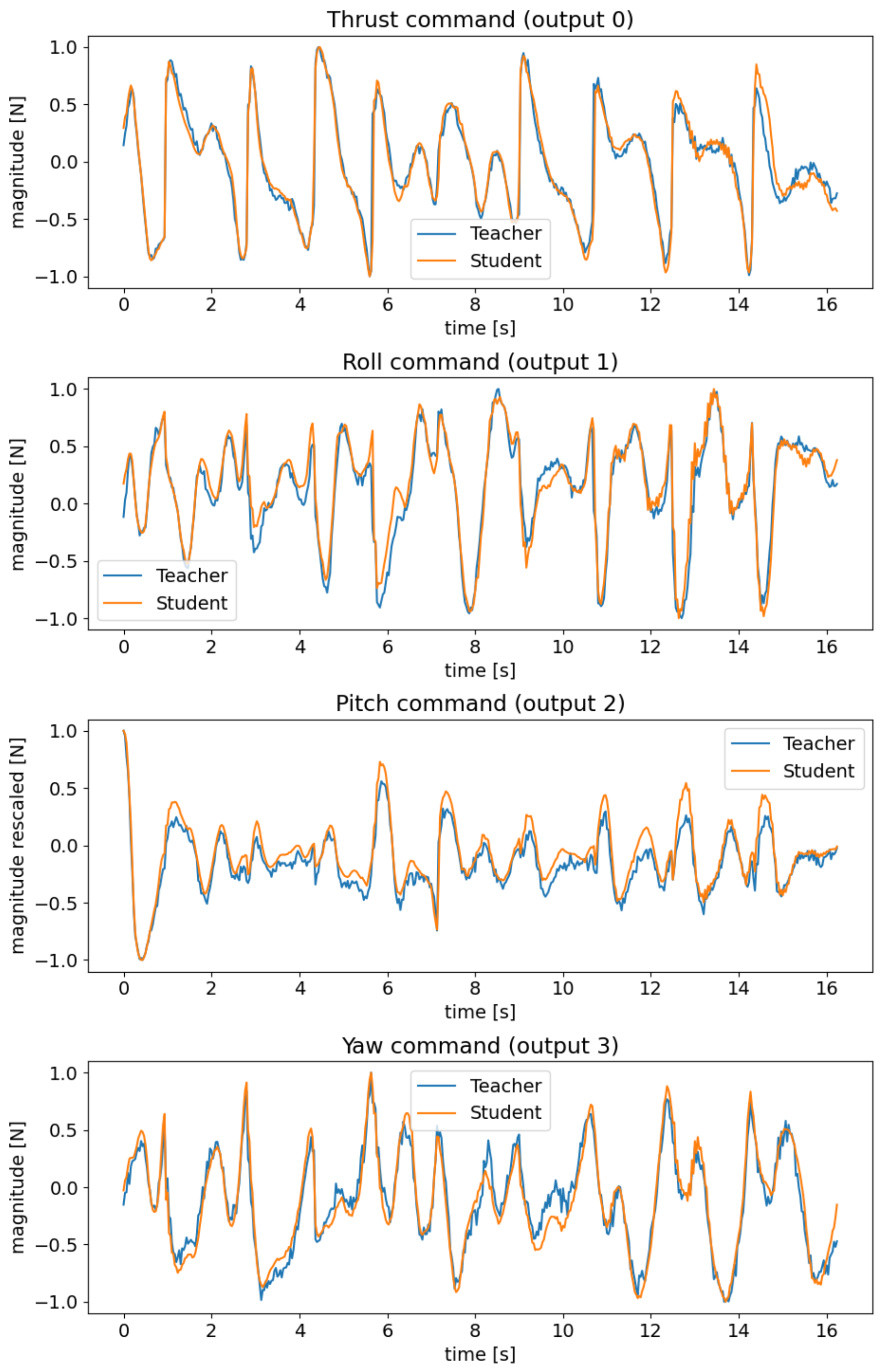


Figure 5.11: Action command comparison between teacher commands and student commands in an example run, where only student's commands were applied. Actions are rescaled in the range (-1,1) for better comparison.

5.4.3 Student Network Architecture Optimization

The NAS optimization is crucial for ensuring that the student policy has the desired number of operations. The next set of experiments and results aim at reducing the computational cost of the student policy trained through IL (which in this context will call seed network).

As explained in sec. 2.7.3, the DNAS training is composed by three different steps: warm-up, search, and fine-tuning. The first phase, in which the full seed network has to be trained, is already done by the previous experiment. The second phase, in which the architecture is optimized, is done with the PIT algorithm [36], employing the DUCCIO regularizer [9]. The training budget for this step was 15 epochs, without any early stopping strategy. The PIT-introduced architecture parameters θ had a separate ADAM optimizer with a constant learning-rate of 0.001. After the search step the chosen network would be the one with the lowest loss among the ones that are compliant with the computational constraints found in sec. 4.4.3. Finally, after the best architecture is chosen, the fine-tuning step consists of a training of the network in which the sole objective is to minimize the task loss. This final step follows the training procedure introduced in the previous experiments.

Results

The search step is presented in fig. 5.12 and fig. 5.13, in which there is presented the loss progression during the training and the progress of the resulting network MACs. The decrease of network's MACs is stable during training, achieving in the 14th epoch the desired number of MACs w.r.t. the constraints imposed. It is intresting to see how the task loss only starts to increase after the 8th epoch, meaning that the lowest number of MACs for which the performance does not suffer is about 40M MACs.

As a result of the search step, the chosen optimized student policy has a computational cost of only 15.3M MACs, which is a 12x improvement over the 183M MACs of the seed network. In tab. 5.9 it is shown a comparison between the seed architecture MACs and the NAS-optimized architecture MACs with a component-level detail and it is evident that the temporal feature extractor optimization had the biggest impact on the computational cost optimization.

The fine-tuning step shows that the NAS optimization dropped the open-loop performance only by 0.01 avg. R2 Score, as shown in tab. 5.10. Also the closed-loop evaluation (tab. 5.11) shows that the optimized network is still able to navigate a race track, keeping a 3% success rate in the randomized visual environment.

Given the results, it is safe to say that the optimized policy can achieve real-time performance on the Bitcraze Crazyflie 2.1. However, similarly to the seed architecture, its closed-loop results show that the NAS-optimized policy still suffers from the distribution-shift problem, presenting a success rate which is lower by 6% w.r.t. the seed architecure, further encouraging the need for a RL fine-tuning step.

Command setpoint	**Seed network MACs**	**Optimized network MACs**	**% of the seed network**
Visual Feature Extractor (Frontnet (80x32))	6'497'216	6'428'414	99%
Projector (FeedForward Network)	590'336	590'336	100%
Temporal Feature Extractor (TCN)	175'906'800	8'276'046	4%
Regressor (Feed-Forward Network)	69'565	69'565	100%
Total	183'063'917	15'364'361	8%

Table 5.9: Seed architecture MACs vs NAS architecture MACs in detail by module.

Command setpoint	**Seed network R2 score**	**Optimized network R2 score**
Thrust command setpoint	0.958	0.949
Roll command setpoint	0.930	0.911
Pitch command setpoint	0.964	0.956
Yaw command setpoint	0.940	0.926
Average	0.948	0.935

Table 5.10: Open loop performance of the NAS policy compared with the Seed policy open loop performance.

Policy	**Environment**	**Average episode reward [-]**	**Average track completion [%]**	**Episode success rate [%]**
NAS	Static	182 ± 115	35%	3%
	Randomized	197 ± 132	37%	3%
	Unknown	127 ± 70	24%	0%

Table 5.11: Closed-loop performance of the NAS policy.

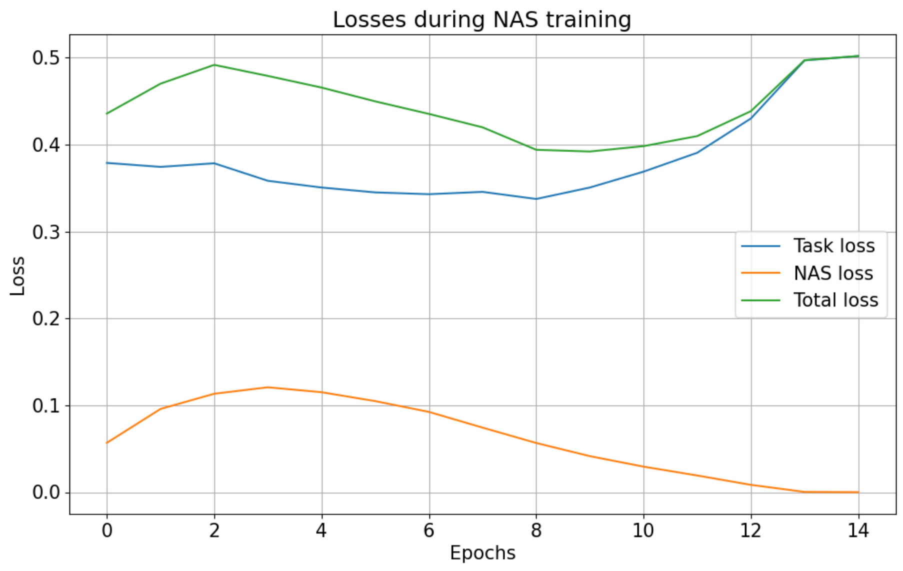


Figure 5.12: NAS Loss progression during training.

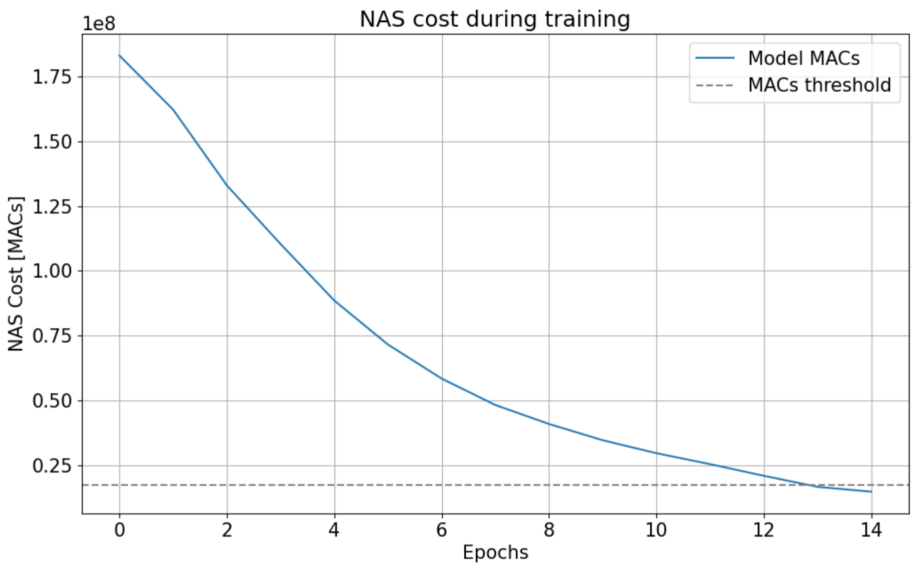


Figure 5.13: NAS Cost progression during training.

5.4.4 Asymmetric Actor-Critic RL Fine-tuning

As shown in the last two experiments with the closed-loop results, both the seed student network and the optimized student network are able to navigate the race tracks, but non in an optimal manner. This last set of experiments is the key to optimize the student policy in order to achieve the best results possible.

Here the results of previous experiments are used: for the actor we use the network resulted of the NAS experiment, which we want to optimize, while for the critic we use the same critic used for the teacher. Before the start of the RL fine-tuning a first fine-tuning of the critic was done, in order to adapt to the new actor and ensure that the value estimates are more precise in the states that the NAS-optimized architecture will encounter. In order to do so, the actor loss was forced to zero for 100'000 rollouts, so it could not update its weights, while the critic could update its weights. Since both the data and the model is more complex, due to computational constraints a lower number of parallel environments was used: 30. Because of this reduction, a higher number of rollouts per update was needed. In this case 2000 rollouts per update were used, thus a total of 60k datapoints per update. At each update, the dataset was divided in 400 mini-batches, for an effective batch-size of 150, which is very similar to the batch size used for the IL training (128). The network is trained for 3 epochs on each update. In order to maintain the visual randomization, the background of the environment changed every 100 rollouts. The rest of the experimental setup is the same as presented in the teacher's training experimental setup, including rewards, learning rate, etc.. The critic fine-tuning was considered a preliminary step and was kept separated from the actual policy optimization, though both follow the same training structure.

Results

The fig. 5.14 and fig. 5.15 shows progress during the RL fine-tuning training. It can be observed that the RL fine-tuning training steadily increases the episodic total reward in time and catastrophic forgetting doesn't occur. The episodic total reward converges at a reward of 410 during training. Given that the same reward function was used for both the RL fine-tuning and the teacher training, the difference in reward between these two policies can be explained by the different observation space, as the teacher uses priviledged informations, which are not usable in the real world.

The the closed-loop evaluation (tab. 5.12), shows that 70% of the runs in the randomized visual environment are successful. More interestingly, the policy reaches a 51% success rate in the unknown environment which that the policy is robust enough to generalize also to unknown environments, thanks to the domain generalization techniques applied. Even though the drop for this metric is significant (−19%) w.r.t. the known visual environments, the average percentage of run completion is more encouraging, showing that it manages to complete 70% of the track on average w.r.t. 76% of the track in a random, known environment.

The results obtained confirm the fact that the RL fine-tuning is needed in order to maximize the performances of the policy, as the DAgger algorithm doesn't fully solve the distribution shift problem, increasing the success rate from 3% of the NAS-optimized policy to 70%, in visually known environments.

Policy	**Environment**	**Average episode reward [-]**	**Average track completion [%]**	**Episode success rate [%]**
RL Finetuning	Static	413 ± 208	75%	66%
	Randomized	417 ± 208	76%	70%
	Unknown	385 ± 199	70%	51%

Table 5.12: Closed-loop performance summary of RL fine-tuning policy.

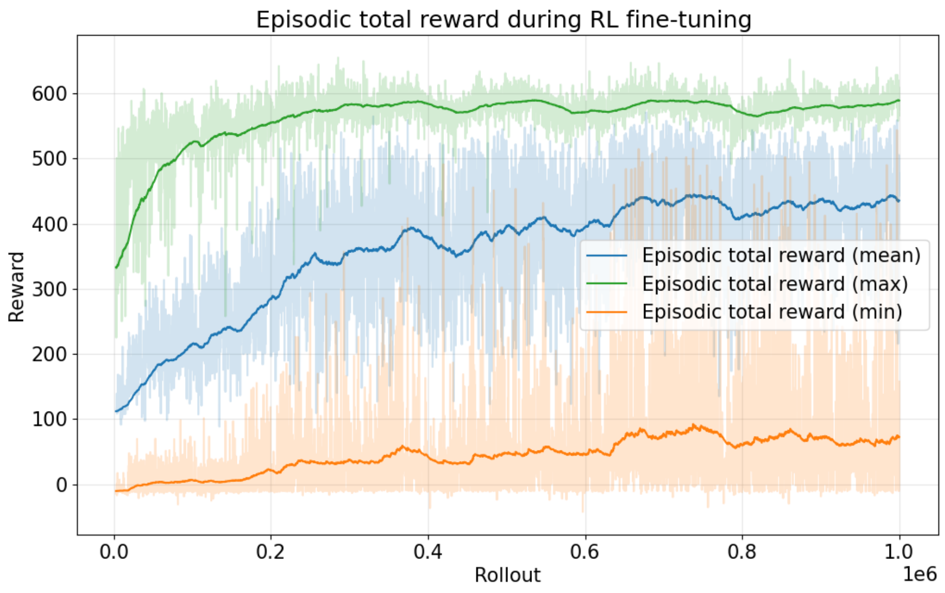


Figure 5.14: RL fine-tuning episodic total reward progress during the training.

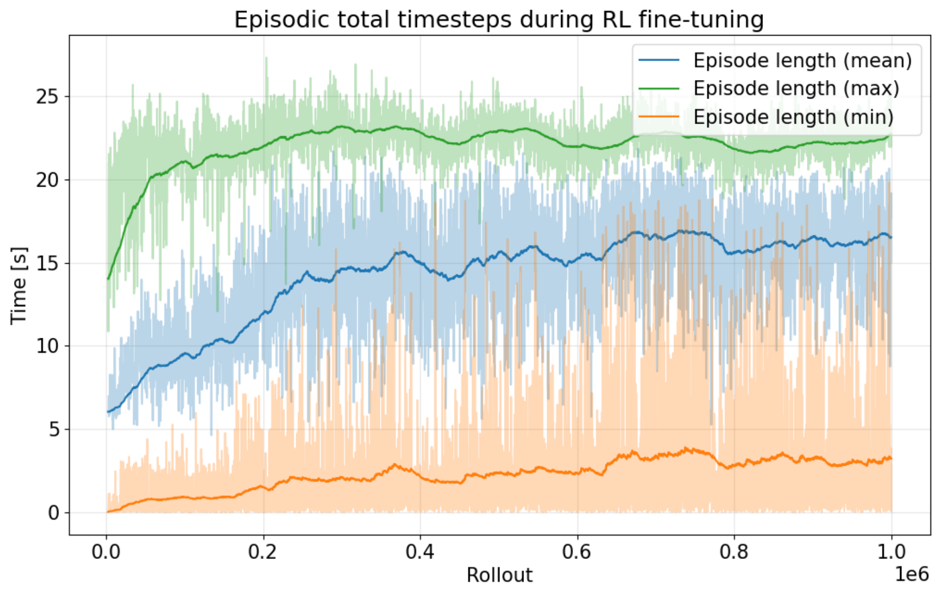


Figure 5.15: RL fine-tuning episode length progress during the training.

5.5 Comparison of Pipeline Steps

The previous experiments led to a policy which can navigate through an unknown track (and eventually an unknown visual environment) with a success rate of 70% (and 51%, respectively). Each step of the procedure played an essential role in the development of the final policy, both in terms of performance and in terms of computational cost. While the teacher policy was deployable on the Crazyflie 2.1 and had optimal performances, reaching a success rate of 99%, it used priviledged information, which is unusable in a real world scenario. It was used to train an vision policy through imitation learning. The vision-based policy resulted from the imitation learning training was able to navigate the race track and could complete 9% of the runs in already seen environments, but it has too costly in terms of computations. This policy was then used as a seed network for NAS optimizations, which pruned the network in order to make it deployable on the Crazyflie 2.1 by respecting the computational constraints. The NAS-optimized policy had a 12x reduction in the number of operations, which led also in a drop in performances, only having 3% of successful runs. Though this performances were good enough to ensure that a RL-based fine-tuning through the asymmetric actor-critic framework would not lead to catastrophic forgetting. The aim of this fine-tuning was to maximize the performances of the policy, achieving a success rate of of 70% in known environments. Moreover the domain generalization techniques used during the training of this policy (namely the visual domain randomization and the pencil-filter), led to a robust agent, which can navigate also in unknown environments achieving 51% of success rate. The summary of these results are presented in tab. 5.13.

Policy	**Environment**	**Average episode reward [-]**	**Average track completion [%]**	**Episode success rate [%]**
Teacher	—	535 ± 50	99%	99%
Seed	Static	177 ± 134	40%	9%
	Randomized	180 ± 125	40%	9%
	Unknown	145 ± 110	33%	3%
NAS	Static	182 ± 115	35%	3%
	Randomized	197 ± 132	37%	3%
	Unknown	127 ± 70	24%	0%
RL Finetuning	Static	413 ± 208	75%	66%
	Randomized	417 ± 208	76%	70%
	Unknown	385 ± 199	70%	51%

Table 5.13: Closed-loop performance summary of the policies trained.

Chapter 6

Conclusions and Future Works

In this thesis, a novel approach for training fully end-to-end deep learning policies for ultra-low-power nano-drones aimed at gate-based autonomous drone racing was proposed. The proposed method consists in multiple steps. First a learning-by-cheating framework is used, in which a priviledged information teacher policy is used to teach a vision-based student policy. Second, the dataset created in the previous step is used to apply neural architecture search techniques in order to reduce the policy's computational cost and ensure its deployability. Finally, an RL fine-tuning through the asymmetric actor-critc framework is employed by using the pretrained post-NAS actor network and the teacher's pretrained critic network, in order to maximize the student network's performances.

The teacher optimal policy taken as an upper bound in this work is able to finish successfully the tracks 99% of the time, but it requires priviledged information which is not obtainable in the real world. Our method, on the other hand, yields a policy which uses only information obtainable from the on-board sensors, such as the camera and the accelerometer, and is able to complete successfully an unknown ADR track up to 70% of the time when in a already seen visual environment. Thanks to the NAS techniques applied, the final policy results deployable at 30Hz entirely onboard the GAP8 SoC, allowing it to process image frames in real-time at the native camera frame rate, requiring only 15M MAC, a 12x reduction w.r.t the pre-NAS seed model (183M MAC). The student policy performances are obtainable thanks to the asymmetric actor-critic RL fine-tuning, which maximized the student network performances, going from a 3% success rate in known visual environment of the NAS-optimized network to the 70% cited previously. Special attention has been put also in minimizing the reality gap, enhancing its domain generalization capabilities by employing techniques such as visual domain randomization and pencil-filtering. Thanks to these techniques the final student policy is able to fly in an unknown track with an unknown visual environment with a success rate of 51%, whereas without these techniques the agent would not be able to fly in an unknown visual environment.

In building the student policy architecture, various aspects of the neural network employed have been analysed in depth. Starting from a neural network family, which has already been proven to have state-of-the-art performances for this task, its performance vs computational cost tradeoffs have studied w.r.t. its hyperparameters choices. Here it has been shown that starting from a network configuration which focuses on performance, which is successively optimized via NAS techniques can still result deployable.

In conclusion this thesis proposes a novel method of developing fully end-to-end deep learning vision-based policies for resource limited deployment targets, showing its effectiveness in developing an on-board sensors-based, accurate and real-time deployable policy. In the scope of autonomous nano-drone racing, this thesis presents the first work which interests both autonomous drone racing and resource limited nano-drones. As a first work tackling end-to-end autonomous nano-drone racing, more work can be done on top of this thesis. Thanks to the capability of our policy to generalize in unseen environments, this work can be used as a baseline for new pipelines or as a pre-trained model for other end-to-end deep learning based control pipelines, in order to investigate different aspects of autonomous nano-drone racing, such as power consumption in ADR scenarios or physical dynamics randomization robustness. Future works include the deployment of the final policy on the Crazyflie 2.1 and, given the drone small size, robustness to physical disturbances such as aerodynamic effects or wind.

Bibliography

[1]

[2] Crazyflie 2.1 — store.bitcraze.io. `https://store.bitcraze.io/products/crazyflie-2-1`. [Accessed 22-09-2024].

[3] The Coordinate System of the Crazyflie 2.X | Bitcraze — bitcraze.io. `https://www.bitcraze.io/documentation/system/platform/cf2-coordinate-system/`. [Accessed 12-09-2024].

[4] Alphapilot ai drone innovation challenge, 2023.

[5] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.

[6] R. W. Beard and T. W. McLain. 2012.

[7] S. Bonato, S. C. Lambertenghi, E. Cereda, A. Giusti, and D. Palossi. Ultra-low power deep learning-based monocular relative localization onboard nano-quadrotors. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, page 3411–3417. IEEE, May 2023.

[8] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70:1253–1268, 2020.

[9] A. Burrello, M. Risso, B. A. Motetti, E. Macii, L. Benini, and D. J. Pagliari. Enhancing neural architecture search with multiple hardware constraints for deep learning model deployment on tiny iot devices. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[10] E. Cereda, L. Crupi, M. Risso, A. Burrello, L. Benini, A. Giusti, D. Jahier Pagliari, and D. Palossi. Deep neural network architecture search for accurate visual pose estimation aboard nano-uavs. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6065–6071, 2023.

[11] Y. Chen and N. O. Pérez-Arancibia. Nonlinear adaptive control of quadrotor multi-flipping maneuvers in the presence of time-varying torque latency. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9, 2018.

[12] c. CORDIS. Low-latency perception and action for agile vision-based flight, 01 2024.

[13] Z. Ding. *Imitation Learning*, pages 273–306. Springer Singapore, Singapore, 2020.

[14] J. Fu, Y. Song, Y. Wu, F. Yu, and D. Scaramuzza. Learning deep sensorimotor policies for vision-based autonomous drone racing. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5243–5250. IEEE, 2023.

[15] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. Richemond, E. Buchatskaya, C. Doersch, B. Avila Pires, Z. Guo, M. Gheshlaghi Azar, et al. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

[16] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza. Autonomous drone racing: A survey. *IEEE Transactions on Robotics*, 40:3044–3067, 2024.

[17] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. Searching for mobilenetv3. *CoRR*, abs/1905.02244, 2019.

[18] P. Imaging. *PMW3901 Optical Flow Sensor Datasheet*. Available at: `https://www.pixart.com`.

[19] D. Jahier Pagliari, M. Risso, B. A. Motetti, and A. Burrello. Plinio: A user-friendly library of gradient-based methods for complexity-aware dnn optimization, 2023.

[20] G. Jocher, J. Qiu, and A. Chaurasia. Ultralytics YOLO, Jan. 2023.

[21] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987, 2023.

[22] E. Kaufmann, L. Bauersfeld, and D. Scaramuzza. A benchmark comparison of learned control policies for agile quadrotor flight. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 10504–10510. IEEE, 2022.

[23] L. Lamberti, V. Niculescu, M. Barciś, L. Bellone, E. Natalizio, L. Benini, and D. Palossi. Tiny-pulp-dronets: Squeezing neural networks for faster

and lighter inference on multi-tasking autonomous nano-drones. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 287–290, 2022.

[24] G. Li, M. Mueller, V. Casser, N. Smith, D. L. Michels, and B. Ghanem. Oil: Observational imitation learning. *arXiv preprint arXiv:1803.01129*, 2018.

[25] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[26] A. Loquercio, E. Kaufmann, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza. Deep drone racing: From simulation to reality with domain randomization. *IEEE Transactions on Robotics*, 36(1):1–14, Feb. 2020.

[27] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 2018.

[28] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, A. Mandlekar, B. Babich, G. State, M. Hutter, and A. Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

[29] H. Moon, J. Martinez-Carranza, T. Cieslewski, M. Faessler, D. Falanga, A. Simovic, D. Scaramuzza, S. Li, M. Ozo, C. De Wagter, et al. Challenges and implemented technologies used in autonomous drone racing. *Intelligent Service Robotics*, 12:137–148, 2019.

[30] M. Muller, V. Casser, N. Smith, D. L. Michels, and B. Ghanem. Teaching uavs to race: End-to-end regression of agile controls in simulation. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0, 2018.

[31] M. Muller, G. Li, V. Casser, N. Smith, D. L. Michels, and B. Ghanem. Learning a controller fusion network by online trajectory filtering for vision-based uav racing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[32] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters. An algorithmic perspective on imitation learning. *Foundations and Trends in Robotics*, 7(1-2):1–179, 2018.

[33] D. Palossi, N. Zimmerman, A. Burrello, F. Conti, H. Müller, L. M. Gambardella, L. Benini, A. Giusti, and J. Guzzi. Fully onboard ai-powered human-drone pose estimation on ultralow-power autonomous

flying nano-uavs. *IEEE Internet of Things Journal*, 9(3):1913–1929, 2022.

[34] H. X. Pham, A. Sarabakha, M. Odnoshyvkin, and E. Kayacan. Pencilnet: Zero-shot sim-to-real transfer learning for robust gate perception in autonomous drone racing. *IEEE Robotics and Automation Letters*, 7(4):11847–11854, 2022.

[35] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *arXiv preprint arXiv:1710.06542*, 2017.

[36] M. Risso, A. Burrello, F. Conti, L. Lamberti, Y. Chen, L. Benini, E. Macii, M. Poncino, and D. J. Pagliari. Lightweight neural architecture search for temporal convolutional networks at the edge. *IEEE Transactions on Computers*, 72(3):744–758, 2022.

[37] L. O. Rojas-Perez and J. Martinez-Carranza. Deeppilot: A cnn for autonomous drone racing. *Sensors*, 20(16), 2020.

[38] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[39] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

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

[41] N. Semiconductor. *nRF51822 Product Specification.* Available at: `https://www.nordicsemi.com/`.

[42] A. Serrano-Muñoz, D. Chrysostomou, S. Bøgh, and N. Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24(254):1–9, 2023.

[43] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza. Autonomous drone racing with deep reinforcement learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1205–1212. IEEE, 2021.

[44] STMicroelectronics. *STM32F405 Microcontroller Datasheet.* Available at: `https://www.st.com/en/microcontrollers-microprocessors/stm32f405rg.html`.

[45] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[46] G. Technologies. *GAP8 IoT Application Processor*. Available at: `https://greenwaves-technologies.com`.

[47] H. Technologies. *Himax HM01B0 Ultra Low Power Image Sensor Datasheet*. Available at: `https://www.himax.com.tw/products/cmos-image-sensor-ultra-low-power/hm01b0/`.

[48] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[49] Webots. http://www.cyberbotics.com. Open-source Mobile Robot Simulation Software.

[50] J. Xing, L. Bauersfeld, Y. Song, C. Xing, and D. Scaramuzza. Contrastive learning for enhancing robust scene transfer in vision-based agile flight. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5330–5337. IEEE, 2024.

[51] J. Xing, A. Romero, L. Bauersfeld, and D. Scaramuzza. Bootstrapping reinforcement learning with imitation for vision-based agile flight. *arXiv preprint arXiv:2403.12203*, 2024.