Alma Mater Studiorum - University of Bologna

SCHOOL OF ENGINEERING

DEPARTMENT OF
ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING
"Guglielmo Marconi"
DEI

**MASTER DEGREE IN
AUTOMATION ENGINEERING**

MASTER THESIS
in
Cyber-physical Systems Programming

Learning Agile Flight Using Massively Parallel Deep Reinforcement Learning

Candidate:
Sebastiano Mengozzi

Supervisor:
Prof. Andrea Acquaviva

Co-supervisors:
Prof. Andrea Bartolini
M. Eng. Luca Zanatta

Academic Year 2021/2022
Session III

A Gigliola e Marcella

Abstract

Recent advancements in deep reinforcement learning opened new strategies for developing intelligent and autonomous robots, such as unmanned aerial vehicles (UAVs). Drones are among the most agile UAVs and can solve highly dynamic tasks with flight policies learned with deep reinforcement learning techniques. Specifically, a neural network can be trained to directly map sensory data to control actions. This end-to-end approach is able to produce outputs with less latency and information loss compared to those approaches with separation between perception, planning, and control. For this reason, represents a valid solution for agile flight. In addition, new parallel software simulators are capable of providing tons of high-quality data in a small amount of time that can be used to rapidly learn control policies robust to domain transfers. In this thesis, a drone is trained with the deep reinforcement learning algorithm PPO, using three different reward functions. The obtained control policy emerges from the imposed constraints and the feedback provided. The outcomes are compared over randomized scenarios. The best policy achieves success rates of 97.52% and 77.30% in the randomized training and test scenarios respectively. The results demonstrate that a learning-based control stack, without any planning and sensing stages, can maneuver the quadrotor and solve the agile task, even in previously unseen scenarios. The results pave the way for future development to implement this solution into real-world applications.

Contents

Introduction

Recent breakthroughs in many engineering fields pointed out what the next future will look like. Thanks to the new discoveries in power management and battery manufacturing we are able to produce mobile platforms that can last days with no service. With advanced mechanical and electronic systems design, powerful actuators, reliable sensors, and strong, lightweight mechanical structures can be built to enclose fast and efficient microcontrollers and give them a physical body.

Last but not least, new Artificial Intelligence (AI) paradigms and sophisticated optimization algorithms can now produce impressive results in achieving superhuman capabilities, leveraging huge data sets, realistic physical simulations, and better hardware accelerators for massively parallel computation.

All of these are key enabling factors to achieve autonomous and intelligent robotic systems that have the possibility to revolutionize humans' everyday life. This new wave of intelligent robots will be able to completely substitute people in carrying out dangerous and repetitive activities. It will expand humans' possibilities, reshaping the future of our societies, starting from how we perceive school and work to how we think of our place in the universe.
We are not there yet. What is missing are reliable algorithms that can make robots able to relate to a human-shaped world and understand reality as we do.

Quadrotors are a hot topic in the robotics community since they pose to engineers many challenges at the same time. Due to their mechanical architecture, they are capable of performing high-speed maneuvers in complex environments making them one of the most agile robotic platforms [1]. Lightness comes with a cost: flight autonomy is very limited due to small batteries and low energy density. Moreover, the capacity to glide over the air requires the ability to control this unstable highly nonlinear system [2] with limited embedded systems' computational power. These facts make quadcopters really demanding benchmarks for testing robotics autonomy and extreme dynamical capabilities.

Deep Reinforcement Learning (DRL) techniques have demonstrated in recent years their ability to master intricate strategies and control complex systems to reach a desired goal thanks to the incredible work of many research laboratories and some of

the biggest tech companies around the world [3], [4].

DRL algorithms are based on a simple intuition: the robot can learn from the experience gained by interacting with the surrounding environment, which guides it toward the solution. This fact makes such algorithms one of the most promising tools for achieving robotics intelligence and specifically drone autonomous navigation. In particular, Proximal Policy Optimization (PPO) [5], a state-of-the-art DRL algorithm, has shown great effectiveness in training Neural Networks (NN) for robotic applications [6], [7], [8]. One of the key enabling factors of the vast adoption of these learning techniques is the recent advancements in the development of simulator software. They partially solve one of the biggest limiting factors: the huge amount of training data and trial and error required to learn a feasible policy, which would otherwise be untreatable if directly applied to real-world scenarios.

The ultimate objective of this thesis is to pose the basis for a robotic learning framework to find complex agile policies by interacting with the surrounding environment like humans and animals do. As a simplified example, animals learn to gather food by imitating their counterparts and satisfying their hunger, thus obtaining a reward. The focus will be on a specific instance of this broad topic. Throughout this work will be studied how a quadrotor can learn agile flight leveraging Deep Reinforcement Learning (DRL) techniques. The agile flight will be examined by testing the drone's ability to traverse a fast-moving gate to reach a target position in space without collisions, similar to how a human pilot would do.

The navigation policy is learned through a neural network module that is trained to map state observations directly into low-level motor control commands for steering the platform toward the goal. No explicit perception, trajectory planning stage, and model-based control will be used.

The agile setup will be recreated leveraging the Nvidia Isaac Gym software [9], a simulation tool that can entirely run in Nvidia GPUs hardware to generate tons of parallel environments and realistic interactions, leading to high speedup in the training convergence of the neural network.

This study is useful since it opens the way to future work in robotics autonomy. The robot is free to discover many types of behaviors and policies, emerging from imposed setup constraints and difficult to explicitly specify by engineers. Moreover, the policy is learned in a high-fidelity reconstructed and randomized setup. This last aspect is fundamental for future developments in implementing zero-shot transfer from simulation to reality and, as consequence, coming up with new robust and general frameworks for fast development and prototyping of new high-performance robotic systems with less effort and trail-and-error tuning required.
The thesis starts with chapter 1, which gives an introduction to the basic theoretical

aspects required to comprehend the experimental setup and the mechanisms that make it possible for robots to learn. Chapter 2 outlines some of the most successful works in the area of agile flight for drones applications and, specifically, the ones that follow learning-based approaches. This will give an overview of how the community is debating this topic and an insight into the research works that have inspired this thesis. The developed setup will be presented in detail in chapter 3, together with the design choices needed to perform the experiments fast and efficiently. The obtained results will be discussed in chapter 4 and the different solutions will be compared to understand how to achieve autonomous and intelligent navigation in a challenging scenario.
The thesis finishes with some concluding remarks and thoughts about future work.

Chapter 1

Background

In this chapter, the key concepts needed to understand and build autonomous robotics systems will be examined. Throughout section 1.1, the core concepts behind artificial intelligence and how it is possible to implement the training of neural networks will be shown. Section 1.2 will describe the reinforcement learning framework and some of the most promising algorithms that implement it. The drone dynamical model will be derived in section 1.3, which will furthermore describe some of the main methodologies to achieve autonomous navigation. To conclude, in section 1.4 simulator basic concepts and simulator software will be reviewed.

1.1 Deep Learning

The core challenge that Artificial Intelligence (AI) tackles, is to make computers able to solve tasks that are easy to perform but hard to describe formally. The solution to this problem is to allow machines to learn from experience and understand the world in terms of a hierarchy of concepts, as depicted in figure 1.1, with each concept defined through its relation to simpler concepts [10]. Drawing a graph to represent this hierarchy it would show how the nodes are built on top of each other, forming a deep graph. For this reason, this AI technique is called Deep Learning (DL).

1.1.1 Deep Neural Networks

Deep Neural Networks (DNN) are the core element of learning techniques. Their goal is to approximate some function $y = f^{\star}(x)$ that relates input x, named features, and output y (labels). Hence, a neural network defines a mapping $y = f(x, \theta)$ which is described by parameters θ. These parameters are the ones that change during the training phase, at the end of which, the ones that better approximate the $f^{\star}$ function are found.

Deep neural networks are called networks because they are typically represented

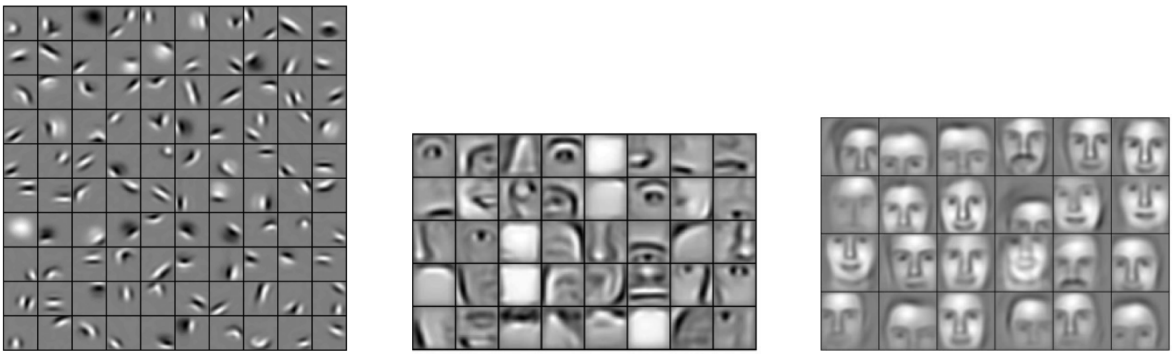


Figure 1.1: Hierarchical representations of faces images features learned with deep networks [11]. The first layer learns to recognize edges (left). The middle layer is able to discern face parts (center). The third layer combines face parts to obtain faces (right).

by composing together many different functions organized in layers. The model is associated with a directed acyclic graph describing how the functions are composed together. The layer where the input enters the network is called the input layer. Its dimension depends on the dimension of the input that the network has to deal with, one unit for each input component. The final layer is called the output layer and, similarly to the input layer, its dimension is equal to the dimension of the output, which depends on the considered application. Between these two layers can be present an arbitrary number of layers, called hidden layers. The higher the number of hidden layers is, the deeper the network will be. Each layer is a vector-to-vector function since the dimension of the input and output could be greater than one.

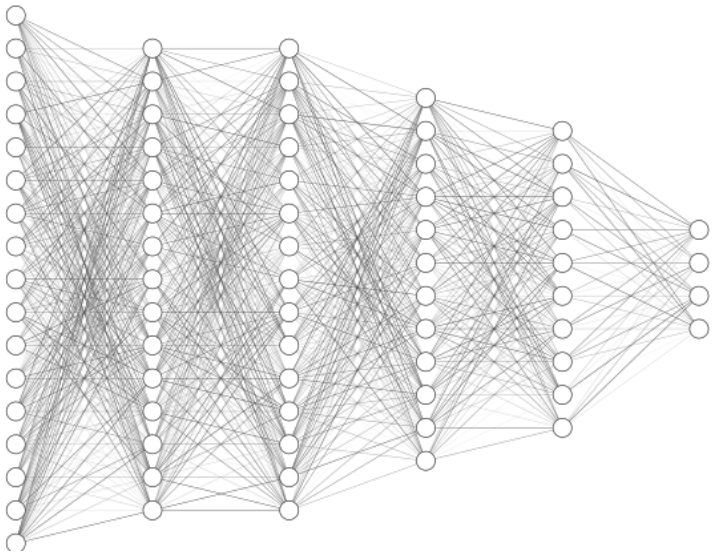


Figure 1.2: Neuron units in a DNN with four hidden layers.

The reason why these networks are called neural is that they are inspired by neuroscience. Rather than seeing these networks as composed of a series of layers, it is

possible to think of a layer as consisting of many units that act in parallel, called neurons, as in figure 1.2. The name neuron derives from the fact that the input-output configuration of these units resembles the physical structure of the biological neuron. They present many input connections and one output, which depends on these inputs and is forwarded to other units. Similarly, biological neurons present many dendrites to receive inputs from other nerve cells and one axon to propagate the output.

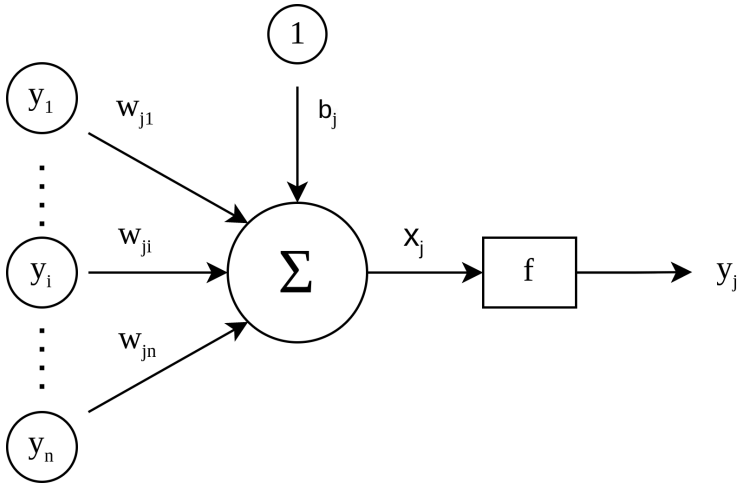


Figure 1.3: Input-output operations done by neuron j.

In figure 1.3 are reported the operations done by a neuron j which belongs to the network. The total input to neuron j, x_j, is a linear function of the weighted sum of the outputs of the neurons y_i, that are connected to j with the addition of an extra input which has always value 1. The weight of each input, w_{ji}, and the bias b_j, which is the weight on this extra input, are the parameters that have to be learned. The total input x_j can then be written as:

$$x_j = \sum_i y_i w_{ji} + b_j \tag{1.1}$$

The output of a unit, y_j, is a nonlinear, real-valued function of its total input

$$y_j = f(x_j) \tag{1.2}$$

where $f(x_j)$ is the so called *activation function*.

The activation function is fundamental for deep neural networks since without its nonlinearity, they could be always transformed into two-layer linear networks. In order to become universal function approximators at least one hidden layer is needed, as the universal approximation theorem states [12], thus the need for the activation function.

To improve the estimate of the parameters θ and come up with a suitable mapping function that solves a desired task, the network, or agent, could either need feedback or not. Based on this fact and, if required, the type of feedback received it is possible to define three main different learning paradigms:

- unsupervised learning;

- reinforcement learning;
- supervised learning.

In unsupervised learning, the agent learns patterns in the input without explicit feedback supply. The most common unsupervised learning task is clustering: detecting potentially useful hidden relations between input examples and clustering them.

In supervised learning, the agent observes some example input-output pairs and finds the underlying relation between them [13]. To achieve this, some training examples accompanied by a label are provided to the network. The training examples and label pairs are called the dataset, and the label represents the right output to the specific training example it is associated with. Training examples specify directly what the output layer must do, but do not say anything about the behavior of the hidden layers. It is the learning algorithm that has to specify how to use those layers to obtain the desired output.

In reinforcement learning the agent learns from a series of feedback reinforcements, such as rewards, or punishments. It has no dataset at its disposal as in supervised learning but it creates its own by gathering interactions with the environment. It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it and consequentially learn them.

1.1.2 Deep Neural Networks Training

In supervised and reinforcement learning techniques the algorithm adopted to learn the parameters θ, which characterize $y = f^{\star}(x)$, is based on the evaluation of the output that the neural network produces, $\hat{y} = f(x, \theta)$, named predicted value.

To obtain the output, an input x, named sample, is fed into the NN and flows from the input layer to the output layer producing a predicted value. This process is called forward propagation and is carried out as described by the equations 1.1 and 1.2.

The evaluation of the predictions is performed by defining a cost function $L(y, \hat{y}(\theta))$, called loss, which is a scalar function of the output of the network $\hat{y}$ and some feedback y (labels or rewards). The cost function represents some metric that has to be minimized, such as an error term between the output of the network and the provided label, or maximized, e.g. the obtained reward by the agent during the interactions with the environment. Usually, the total loss function used to evaluate a neural network will often combine a primary term, e.g. one of the metrics described above, with a regularization term, which is used to help the network to better generalize the behavior in unseen scenarios. If the network is not able to generalize, is said that the network has overfitted. Generalization is fundamental for obtaining usable and reliable algorithms in real-world cases.

Given a loss function is possible to change the values of a network's parameters by minimizing the loss, or its negation, which is

$$\min_{\theta} L(y, \hat{y}(\theta)) \tag{1.3}$$

in order to match the desired outputs. The minimization is obtained through a gradient descent method: this means that the parameters are changed in the direction of the steepest descent on the loss function surface in search of a global minimum. The steepest direction is given by the gradient of the loss function and the parameters are updated through the rule

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} L \tag{1.4}$$

The parameter α is known as the learning rate and defines the size of the update. Computing the gradient is essential to perform the update but is not an easy procedure in the case of a neural network. Fortunately, the backpropagation algorithm [14] does so using a simple and inexpensive procedure [10].

During a training step, the generated output produces a scalar cost as a result of the cost function. The backpropagation algorithm allows the information of the cost to flow backward through the network in order to minimize L by gradient descent. The gradient, which is the mathematical tool needed to represent the influence of each learnable neuron parameter on the final output, is found by the calculating partial derivative of the loss with respect to the network parameters.

The backward pass starts by computing the partial derivative of the loss function with respect to the outputs of each output unit y_j. For this reason, the cost function has to be differentiable and, as an example, can be chosen as the total error

$$L = \frac{1}{2} \sum_{c} \sum_{j} (y_{j,c} - d_{j,c})^2 \tag{1.5}$$

where c is an index over each input-output prediction of the network, j is an index over the output units, y is the output of an output neuron and d is the desired output. Equation 1.6 is obtained by differentiation of the equation 1.5 with respect to the outputs of each output unit y_j:

$$\frac{\partial L}{\partial y_j} = y_j - d_j \tag{1.6}$$

Then, in the equation 1.7, the chain rule is applied to compute the partial derivative of the loss with respect to the total input x_j to those output neurons, giving

$$\frac{\partial L}{\partial x_j} = \frac{\partial L}{\partial y_j} \cdot \frac{dy_j}{dx_j} \tag{1.7}$$

To complete this computation is required that the activation function is selected to be differentiable. Differentiating the nonlinear activation function to get the value of dy_j/dx_j and substituting it gives the function that describes how a change in the total

input x to an output unit will affect the cost function. But this total input is just a linear function of the states of the lower-level units and it is also a linear function of the weights on the connections between the units. For this reason, it is possible to compute how the cost will be affected by changing these states and parameters. Using the equation 1.1, the partial derivative of the loss for a weight w_{ji}, from neuron i to neuron j is

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial x_j} \cdot \frac{\partial x_j}{\partial w_{ji}} = \frac{\partial L}{\partial x_j} \cdot y_i \tag{1.8}$$

and for the output of the i^{th} unit the contribution to the loss resulting from the effect of i on j is

$$\frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial x_j} \cdot \frac{\partial x_j}{\partial y_i} = \frac{\partial L}{\partial x_j} \cdot w_{ji} \tag{1.9}$$

Taking into account all the connections that start from unit i the final equation become

$$\frac{\partial L}{\partial y_i} = \sum_j \frac{\partial L}{\partial x_j} \cdot w_{ji} \tag{1.10}$$

These equations are needed to compute the partial derivative of the loss with respect to the output y for any neuron in the penultimate layer when $\partial L/\partial y$ is given for units in the last layer. Therefore it is possible to repeat this procedure in order to compute this term for successively earlier layers, and finally, find the influence of all the parameters of the network on the loss function.

1.2 Deep Reinforcement Learning

Deep reinforcement learning is an AI technique where an agent learns to accomplish a given task by interacting with the environment via trial and error.
This technique does not fall among either supervised or unsupervised approaches. It differs from supervised learning because the feedback to train the network is not generated by exploiting some ready-to-use labeled dataset, on the contrary, unlabeled data to learn from are gathered during the training phase. It also differs from unsupervised learning due to the fact that reinforcement learning can make use of feedback, or supervision while training.

Reinforcement Learning (RL) problems can be expressed as a system consisting of an agent and an environment, as shown in figure 1.4. An environment produces what is called a state, the information describing the state of the system. The state belongs to the state space, which is the set of all possible states that the environment can encounter. It can be defined in many different ways, e.g. as integers, real numbers, vectors, matrices, and structured or unstructured data.

The agent interacts with the environment by selecting an action to perform on the basis of the state of the system that it observes. Sometimes the overall state of the

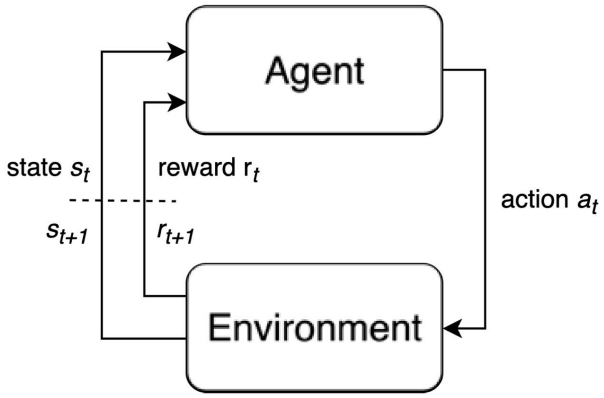


Figure 1.4: The reinforcement learning control loop [15].

system cannot be known by the agent, which can only observe some part of it from its point of view. The chosen action belongs to the action space, defined as the set of all possible actions that the agent can perform, which are often represented as scalar or vectorial quantities. The action space can be continuous or discrete, meaning that actions, e.g. actuators control inputs, can assume values belonging to $\mathbb{R}$ in the first case, or just assume a finite number of values in the second one.

Once the action is selected, the environment receives it and transitions to the next state, returning feedback to the agent, called reward, which can be either a positive, null, or negative scalar depending on the obtained outcome. When a cycle state, action, and reward completes, one time step t has passed. This sequence repeats until the problem reaches a terminal state or the maximum time step T.

In reinforcement learning problems the goal of the agent is to maximize the objective, which consists of the sum of the rewards obtained. To achieve this result it has to select good actions, or rather the actions that led to positive rewards, thus reinforcing the formers. The action-producing function of the agent is called policy. This function maps states or observations into actions and it is what the agent has to discover. As explained in section 1.1, a neural network can be used to approximate this nonlinear function, becoming the core component of the agent in the RL loop.
This exchange of signals between the agent and the environment unfolds in time: the tuple (s_t, a_t, r_t) called experience, corresponds to state, action, and rewards signals exchanged at the time step t. During the time horizon, called episode, from $t = 0$ until termination, many experiences occur. The ordered sequence of experiences over an episode $\tau = [(s_0, a_0, r_0), (s_1, a_1, r_1), ...]$ is called trajectory.
The interaction agent-environment can be thought of as a sequential decision-making process. The mathematical framework to model this kind of problem is called Markov Decision Process (MDP).

1.2.1 DRL as Markov Decision Process

The transition function is the rule that specifies how the environment transitions from one state to the next one. This function can be seen as a probability distribution that depends on all the preceding states and actions that occurred since the beginning of the episode.

$$s_{t+1} \sim P(s_{t+1}|(s_0, a_0), (s_1, a_1), ..., (s_t, a_t)) \tag{1.11}$$

As described by the formula 1.11, the next state is obtained by sampling the probability distribution P. For an episode that lasts many steps, this problem becomes untractable, making it impossible to model a transition function that takes into account all the past effects and to find a suitable policy that can capture all these complex interactions.

To make the transition function more practical it is assumed that the Markov property holds: this property states that the transition to the next state s_{t+1} only depends on the previous state s_t and action a_t. The new transition function then becomes

$$s_{t+1} \sim P(s_{t+1}|s_t, a_t) \tag{1.12}$$

Despite this assumption, which simplifies the transition function formulation, many real-world processes can be expressed in this way, since the state can be defined to include any necessary information required to make the transition function Markov, including previous states.

As pointed out before, the state known by the agent, called observation, could not correspond to the actual state of the environment. In this case, the environment is described as Partially Observable MDP, or POMDP, and does not satisfy the Markov property. To tackle this category of problems some other methods have to be implemented, such as providing past observations as input.

The MDP formulation of a reinforcement learning problem is defined by 4-tuple:

- $\mathcal{S}$, the states set;
- $\mathcal{A}$, the actions set;
- $P(s_{t+1}|s_t, a_t)$, the state transition function of the environment;
- $\mathcal{R}(s_t, a_t, s_{t+1})$, the reward function of the environment, which associates to a tuple (s_t, a_t, s_{t+1}) the scalar reward r_t.

The agent does not have access to the transition and reward functions. It can only get information about these functions through the provided feedback, the reward, as a result of its action choices in a particular state.

As explained in the introductory part of this section, the goal of the agent is to maximize an objective, which takes into account the reward gathered during an episode.

It is possible to formalize this concept by defining the return $R(\tau)$, with τ the trajectory over an episode that ends at $t = T$.

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... + \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t \quad \text{with} \quad \gamma \in [0, 1] \tag{1.13}$$

As it is possible to note from equation 1.13, the return is the weighted sum of the rewards in a trajectory, being the discount factor γ the summation weight. The discount factor plays an important role in reinforcement learning problems because changes the value of future rewards. The smaller it is, the less value will be given to rewards received in future time steps. On the contrary, the larger γ is, the more value will be given to future rewards.

The objective $J(\theta)$ that the agent has to maximize, is defined as the expectation of the returns over many trajectories sampled from a policy π

$$J(\tau) = \mathbb{E}_{\tau \sim \pi}[R(\tau)] = \mathbb{E}_{\tau} \left[\sum_{t=0}^{T} \gamma^t r_t \right] \tag{1.14}$$

The discount factor will modify the behavior of the agent, which will care to maximize the short-term reward for small values of γ and maximize the long-term reward for higher values of γ. The right value of the discount factor depends on the problem to solve.

1.2.2 Q- and V- Functions

There are three primary functions in reinforcement learning that can be learned during the training:

1. The policy π, which maps states into actions.
2. The environment model $P(s_{t+1}|s_t, a_t)$.
3. The value function, $V^{\pi}(s)$ or $Q^{\pi}(s, a)$, useful to estimate the expected return and get information about the objective.

The policy is what the agent learns in order to maximize the objective. A common choice is to make it stochastic. This is a very important solution to balance exploration and exploitation during the training phase. The exploration component is what makes the agent try different kinds of actions, even if it risks bad rewards, while the exploitation component is what makes the agent use gathered knowledge and take safe actions that in the past led to good rewards. The right amount of exploration is fundamental to experiencing states that otherwise would have never been visited and that could lead to the solution of the problem. On the other hand, exploitation is what

makes the agent progress in the training and obtain high scores. This is called the exploration-exploitation trade-off.

The transition function provides information about the environment. By learning this function the agent is able to predict the next state given the current state and the chosen action. This is useful to plan good actions.

The value functions condense information about the objective. They are used by the agent to understand the goodness of states and actions available in a particular state considering the expected future return.

The value function V^π estimates how good or bad a state is and is defined as

$$V^\pi(s) = \mathbb{E}_{s_0=s,\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right] \tag{1.15}$$

V^π measures the expected return starting from state s to the end of the episode and acting according to the same policy π. It is a forward-looking measure since all the rewards received before state s are ignored.

The value function Q^π estimates how good or bad a state-action pair is and is defined as

$$Q^\pi(s,a) = \mathbb{E}_{s_0=s,a_0=a,\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right] \tag{1.16}$$

Q^π measures the expected return from taking the action a in a particular state s assuming that the agent acts following the same policy and is a forward-looking measure too. As it is possible to note from the corresponding equations, the Q- and V- functions are similar in their formulation in fact, these two functions are closely related. $V^\pi(s)$ can be seen as the expectation over the Q-values for all the possible actions a available in a state s under the policy π:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}\left[\, Q^\pi(s,a)\,\right] \tag{1.17}$$

while $Q^\pi(s,a)$ can be rewritten as:

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\, r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n}\,\right] + \gamma^{n+1} V^\pi(s_{t+n+1}) \tag{1.18}$$

By combining V^π with the rewards from a trajectory it is possible to compute Q^π. The influence of the action on the Q-value comes from considering the one-step ahead V-function, whose value depends on the chosen action at state s. Thanks to these two relations one function can be computed starting from the other one.

1.2.3 DRL Algorithms

There exist three major families of deep reinforcement learning algorithms, each one corresponding to one of the learnable functions: policy-based, value-based, and model-based.

Algorithms belonging to the policy-based class learn directly the action-producing function, the policy π. The advantage of policy-based algorithms is that they are a very general class of optimization methods because they can be applied to problems with any type of action, e.g. discrete, continuous, or both. They also directly optimize the objective of the agent, since the $J(\tau)$ depends on the return, which is maximized by adopting an optimal policy. Moreover, this class of algorithms is guaranteed to converge to a locally optimal policy as proven by the Policy Gradient Theorem [16]. One disadvantage is that these methods have high variance [17], meaning that the training dynamics, which is environment-dependent, changes a lot among the episodes and the training process can become unstable. Another drawback is that these algorithms are sample-inefficient because they can use only fresh interactions to train on, discarding past trajectories.

Value-based algorithms use the learned value function to evaluate state-action pairs to generate a policy. Due to this approach, they solve the problem indirectly. They are more sample-efficient with respect to policy-based algorithms because they make better use of gathered data, however, there is no guarantee that these algorithms will converge to an optimal solution. Moreover, the most adopted versions of these algorithms are only applicable to discrete action spaces.

Model-based algorithms learn the environment's transition dynamics or use a known model to make predictions. By knowing a model of the environment, many trajectories can be computed offline and then only the best is applied. This is an advantage in situations where is difficult to gather experiences from the environment. Fewer interactions are needed to achieve good results compared to the previous classes, but the major disadvantage is that good models are often difficult to come by. Because of this, predictions that go many steps into the future progressively lose accuracy, making the model unreliable.

The ideas behind these classes of algorithms can be mixed to obtain combined methods. These methods learn two or more of the primary reinforcement learning functions and combine the strengths of each algorithm to compensate for individual drawbacks. The most widely adopted combined algorithms are called Actor-Critic (AC) algorithms. The name derives from the fact that these algorithms learn the value function, which provides more informative feedback compared with the return, to critique how the agent acts, shaping the learning phase of the policy.

1.2.4 Advantage Actor-Critic (A2C)

The Advantage Actor-Critic (A2C) algorithm belongs to the class of combined algorithms of the Actor-Critic type [18] [19].

The core structure of all the AC algorithms is composed of two parts, the actor

which learns a parametrized policy, and the critic which learns a value function to evaluate state-action pairs to provide a reinforcing signal to the actor. The value function is more informative with respect to the reward since the latter often is received only when the task is solved and not along the path to solving it. By guessing the future, the learned value function makes the reinforcing signal less sparse, guiding the agent toward the goal.

Both the policy and the value function, are two nonlinear functions of the state of the environment and two deep neural networks are adopted to learn and approximate those functions. These networks are called policy network and value network. In the AC algorithms usually, the policy network and value network are not implemented as two separate networks but rather as one deep neural network with two heads, one for each component. In this way, the actor and the critic share the lower layers of the network, while they have distinguished deeper layers with different output spaces, one corresponding to the action space and the other being a scalar quantity.

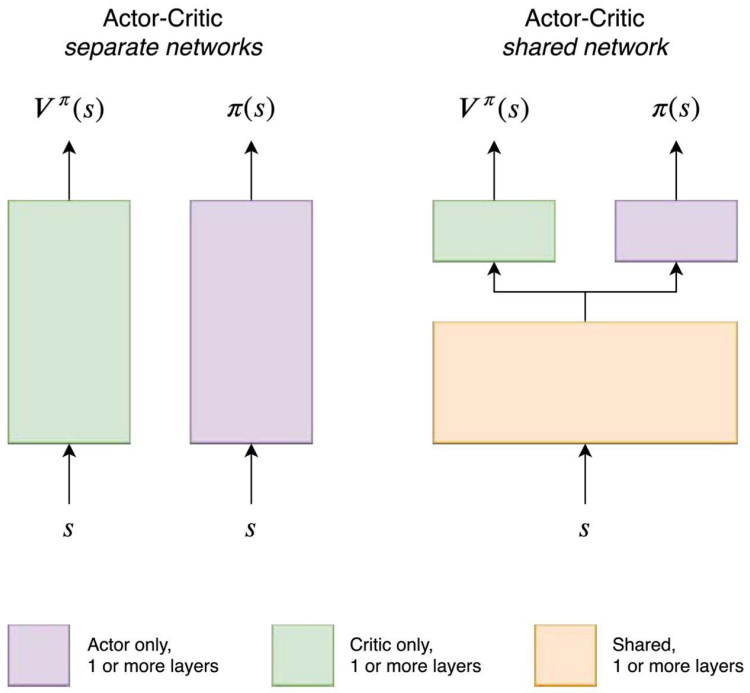


Figure 1.5: Actor-Critic network architectures: shared vs. separate networks [15].

The benefit of sharing the lower-level layers is reasonable since both the actor and the critic have to learn a common representation of the state space, maintaining a number of specialized layers to carry out different tasks. Using the AC structure makes learning the policy depend on how well the value network is able to judge the chosen actions. But at the beginning of the training loop, the critic has no clue about how to produce the reinforcing signal. For this reason, the learning process of the policy cannot give good results until the value function can generate reasonable signals for the actor. This particular architecture makes the learning during the initial iterations

faster, with fewer parameters to train and less memory occupied by the network with respect to the two separate networks implementation. One downside of this approach is that it can make learning more unstable due to different contributions to the gradient of the two components. Balancing them requires one more hyperparameter to tune.

Actor

The actor is what learns the parametrized policy π_θ, being θ the parameters of the neural network. To learn those parameters the policy gradient algorithm is implemented. Formally, the policy gradient algorithm finds a solution for the following problem:

$$\max_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{1.19}$$

This is a maximization problem, solved through gradient ascent on the policy parameters θ by updating the parameters in the direction of the steepest ascent.

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\pi_\theta) \tag{1.20}$$

As described in the equation 1.20, the direction of the steepest ascent is given by the gradient of the objective $J(\pi_\theta)$, with α being the learning rate.

The objective's gradient $\nabla_\theta J(\pi_\theta)$ is what is called the policy gradient. It is obtained by differentiation of both terms in equation 1.20.

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] \tag{1.21}$$

Since the rewards composing the return function are generated by an unknown function, the second term of the equation 1.21 cannot be differentiated. The way in which policy parameters θ influence the return function $R(\tau)$, is by changing the state and action distributions, expressed as MPD, that, as consequence, modify the rewards obtained by the agent. For this reason, equation 1.21 needs to be rewritten in order to explicit such a relation and compute the gradient.

Considering a function $f(x)$ parametrized by a probability distribution $p(x|\theta)$, it is possible to prove that the gradient of the expectation of that function is equivalent to the expectation of the gradient of the log probability multiplied by the original function.

$$\nabla_\theta \mathbb{E}_{x\sim p(x|\theta)}\left[f(x)\right] = \mathbb{E}_x\left[f(x)\nabla_\theta \log p(x|\theta)\right] \tag{1.22}$$

This identity is applicable to the return function, expressing the probability distribution, which parametrizes it, as the agent's action probability $\pi_\theta(a_t|s_t)$.
With this, it is possible to rewrite the policy gradient in a form that can be differentiated.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{1.23}$$

The policy gradient is the mechanism by which action probabilities produced by the policy, or actor network, are scaled up or down depending on the obtained return value, respectively greater or smaller than zero. Over many episodes and parameter updates the policy will learn to produce actions that result in higher values of $R(\tau)$.

In advantage actor-critic, the function used to re-shape the action's probability distribution is not the return as in the above formulation, but the value function called advantage A^{π}. This function is what the critic network learns, to guide the learning of the policy through the policy gradient.

To conclude, the formulation of the policy gradient used in the A2C algorithm is shown in equation 1.24. Instead of sampling many trajectories per policy and computing the expectation, the gradient of several steps in the same trajectory is averaged, simplifying the problem.

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_t \left[(A_t^{\pi} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) \right] \tag{1.24}$$

Considering the case of continuous action space, usually, there will be several actions that can be performed, and each one of them can take a value from some continuous range. As an example, in the LunarLander ATARI game, the lander is equipped with two side thrusters to control the stability and one main engine to slow down and land. Each one of these actuators (or actions) can be fired with different power intensities, which are continuous values belonging to a finite range. To implement such a continuous action space and balance exploration-exploitation, the actor neural network learns the mean and variance of normal distributions, one for each continuous action, that once sampled give the actual control actions value. The probability distribution is described by means of its probability density function

$$f(x \,|\, \mu,\, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{1.25}$$

with μ the mean value and σ the variance. To improve numerical stability [17], this expression is simplifyed applying the logarithm:

$$\log \pi_{\theta}(a_t|s_t) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2} \tag{1.26}$$

During the training phase, the mean and variances returned by the actor head change, since the network parameters are updated. Because of this, the shape of the distribution is altered. As the agent learns, those probability distributions will have their means centered on the values of the actions that, considering a particular state, led to positive rewards. Furthermore, the variance will be reduced so that the action value near the mean will be sampled more often. This behavior is reported in figure 1.6.

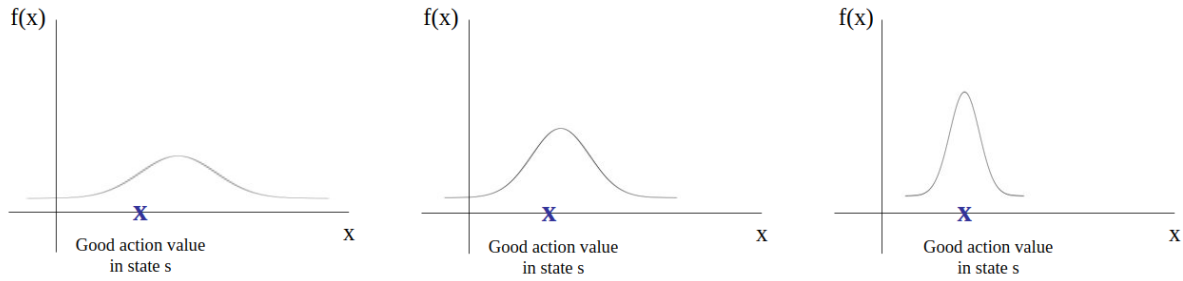


Figure 1.6: Actor learning the policy represented by a normal probability density function. The agent improves from left to right, centering the mean value on the action value that led to positive rewards in a particular state and reducing the variance.

Critic

The critic task is to learn the advantage function to evaluate state-action pairs and guide the actor training. The advantage function is defined as follows:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{1.27}$$

This value function measures how better or worse an action is with respect to the other actions available in the state where the agent is. Evaluating the policy with this approach provides some benefits. First of all, when, in a particular state, is estimated that all of the actions bring the same rewards, the advantage function is 0. In this case, no choice will be discouraged. This property comes from the fact that $\mathbb{E}_{a \in \mathcal{A}}\left[(A^{\pi}(s_t, a)\right] = 0$.

Another benefit of using the advantage function is that it avoids penalizing or boosting an action when the agent is in a particularly bad or good state because the value function V, which captures the quality of the state, is subtracted and so ignored. In this way, the agent is encouraged to do the best that he can without considering the surrounding situation. The taken action can only affect the future steps within the trajectory, making the advantage able to capture the long-term effects of an action within the time horizon selected using the discount factor γ.

To estimate the A function it is necessary to estimate before the Q and V functions. In section 1.2.2 it is shown how these two functions are related to each other. Since learning both would be less efficient and prone to inconsistencies, only one is learned. The value function Q is more complex and require many more sample to find a good guess due to its dependence on both actions and states. Moreover, the estimation of V starting from Q, as in equation 1.17, is computationally expensive since requires computing the values for all possible actions in a state and then doing a weighted average among them. In environments where the action space is continuous, this problem is untractable. For these reasons, it is convenient to estimate the Q-function starting from V. To make the equation 1.18 applicable, the expectation is removed by

considering only one trajectory, and the V-function adopted is the one learned by the critic network, $\hat{V}^{\pi}(s)$, as in equation 1.28.

$$Q^{\pi}(s,a) \approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1}\hat{V}^{\pi}(s_{t+n+1}) \tag{1.28}$$

The equation 1.28 is known as n-step forward returns. This formulation makes a tradeoff between bias and variance of the estimation of the Q-value. The n steps of rewards are unbiased because they are ground-truth rewards values obtained along the steps, but they have high variance since they come from a single trajectory that could have been particularly good or bad in solving the problem. On the other hand, $\hat{V}^{\pi}(s)$ has lower variance since it embodies the expectation over all the trajectories followed so far, but is biased because its value is computed using a neural network as a function approximator. Close to t the overall estimate is unbiased, outweighing the introduced variance, which remains limited because of its proportionality to the number of steps n. Further from t, the variance is limited by the estimate of the V-function at the cost of an introduced bias. The hyperparameter n controls the tradeoff between these two behaviors.

In the end, the advantage function estimation is implemented in the following way:

$$\begin{aligned} A^{\pi}(s_t,a_t) &= Q^{\pi}(s_t,a_t) - V^{\pi}(s_t) \\ &\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1}\hat{V}^{\pi}(s_{t+n+1}) - \hat{V}^{\pi}(s_t) \end{aligned} \tag{1.29}$$

The estimate of the V-function is learned by the critic neural network. To train the network, Temporal Difference (TD) learning is used. This technique generates a target value to compare the prediction of the network with and produces an error term. This is minimized using regression loss, such as Mean Square Error (MSE). The target is computed in the equation 1.30 adopting the n-step estimate.

$$V^{\pi}_{tar}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1}\hat{V}^{\pi}(s_{t+n+1}) \tag{1.30}$$

Putting actor and critic together the overall A2C algorithm is obtained.

1.2.5 Proximal Policy Optimization (PPO)

The main challenge when training agents with policy gradient algorithms is that they are prone to performance collapse. After a network update, the agent starts to perform badly. This is a situation that ruins future training iterations because behaving badly will result in poor trajectory generation which will be then used to further update the network.

The performance collapse is caused by the fact that a policy is searched in an indirect way, through parameters of the network inside the parameters space and not

directly in the policy space. But the mapping between the two spaces is not always congruent and distances in both spaces don't necessarily correspond. Since the mapping is nonisometric a small update of the parameters of the network could result in a completely different policy, which could perform really badly.

Proximal Policy Optimization (PPO), by Schulman et al. [5], is a class of optimization algorithms that addresses this common issue of policy gradient algorithms.

The key idea is to guarantee monotonic policy improvement by modifying the objective function of the A2C algorithm. The new objective is called the surrogate objective and is the ratio between the new and the old policy scaled by the advantages computed with the old policy, as in equation 1.31.

$$J(\pi_\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t^{\pi_{\theta_{\text{old}}}} \right] \tag{1.31}$$

However, maximizing this objective without any constraint may lead to very large updates to the policy parameters which could cause riskier changes in the behavior of the new policy. To avoid this the clipped surrogate objective is adopted. By defining

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{1.32}$$

the clipped objective can be written as

$$J^{\text{clip}}(\pi_\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t) \right] \tag{1.33}$$

This objective limits the ratio between the old and new policy to be in the interval $[1-\epsilon, 1+\epsilon]$, so varying ϵ it is possible to limit the size of the update. The more the new policy will be different from the old one, the more $r_t(\theta)$ will deviate from 1. Large policy updates will be discouraged.

1.3 Drone Modeling and Control

Most of the research in autonomous drone navigation has the objective to push the maximum physical limit of these systems and achieve what is called agile flight. This is a very demanding goal because of the difficulties in controlling a resource-constrained flying platform.

Exploiting the hardware's maximum performances is useful to accomplish missions in highly dynamic scenarios such as outdoor environments, or is needed to minimize the amount of time required to execute the task, for example, to win a drone race competition. Executing a task faster is also a way to limit the battery life constraint and is vital in life-threatening situations, for example when quadrotors are used in search-and-rescue scenarios.

1.3.1 Drone Dynamics

In order to be able to design a control stack for agile drone flight, the dynamical behavior of the hardware platform has to be characterized. A quadrotor can be modeled using the following dynamical equations [20]:

$$\dot{r} = v \tag{1.34}$$

$$\dot{v} = g + R \cdot c \tag{1.35}$$

$$\dot{R} = R \cdot \hat{\omega} \tag{1.36}$$

$$\dot{\omega} = J^{-1} \cdot (\tau - \omega \times J\omega) \tag{1.37}$$

The vectors r and v are respectively the position and velocity vectors defined as $r = [\, x \; y \; z \,]^T$ and $v = [\, v_x \; v_y \; v_z \,]^T$ and expressed in world coordinates. R is the rotation matrix which defines the orientation of the body frame with respect to the world frame, and $\omega = [\, p \; q \; r \,]^T$ denotes the body rates (roll, pitch, and yaw respectively) in body coordinates. The skew-symmetric matrix relative to the angular velocity vector ω, is $\hat{\omega}$ and is defined as

$$\hat{\omega} = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix} \tag{1.38}$$

The gravity vector is defined as $g = [\, 0 \; 0 \; -g \,]^T$ and $J = diag(J_{xx}, J_{yy}, J_{zz})$ is the second-order moment-of-inertia matrix of the quadrotor.

The drone can be seen as a rigid body moving in space and which is controlled by four single rotor thrusts f_i as illustrated in figure 1.7.

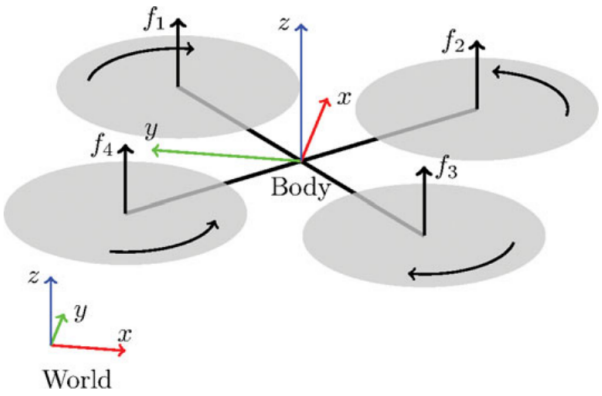


Figure 1.7: Quadrotor with coordinate system and single rotor forces [21].

By changing each one of these four single rotor thrusts, a three-axis resulting torque τ and a mass normalized collective thrust, or lift, c can be applied on the quadrotor's frame. The relation between the single rotor thrusts, the collective thrust, and the

body torques can be formulated following the coordinate system and numbering of figure 1.7, as in equations 1.39 and 1.40.

$$\tau = \begin{bmatrix} \frac{\sqrt{2}}{2} l(f_1 - f_2 - f_3 + f_4) \\ \frac{\sqrt{2}}{2} l(-f_1 - f_2 + f_3 + f_4) \\ k(f_1 - f_2 + f_3 - f_4) \end{bmatrix}, \tag{1.39}$$

$$mc = f_1 + f_2 + f_3 + f_4 \tag{1.40}$$

The quadrotor's arm length is denoted with l and is the distance between the center of mass and the axis of the propeller. Its total mass is referred to as m, and k is the rotor-torque coefficient. It is chosen as an experimentally determined constant as done in [22], [23], and [24], and relates the torque and thrust produced by each rotor.

1.3.2 Autonomous Drone Navigation

Once the dynamic model of the drone is known it is possible to develop a software pipeline to autonomously control the platform's flight behavior.

Many approaches have been developed to sense and interpret the surrounding environment and take autonomous decisions but is possible to highlight two main paradigms: model-based approaches and learning-based ones.

Model-Based Approach

The model-based architecture has become over the years the most popular one. Based on the first principle methods this architecture divides the control stack into three major components: perception, planning, and control as in figure 1.8.

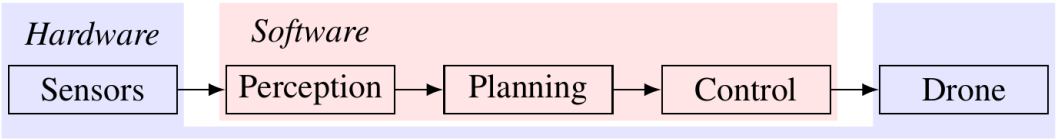


Figure 1.8: A classic control architecture for an autonomous system programmed using model-based approach [25].

The perception control block is what makes it possible for the robot to sense the surrounding environment and estimate its state. It receives the information coming directly from the robot's onboard sensors.
Typical onboard measurements that the drone has at its disposal are data coming from the Inertial Measurements Unit (IMU), such as acceleration, and orientation in space. These measurements are received at high frequencies and are integrated over time to obtain position and velocity. This estimation is reliable just for a few steps because

the measurements are noisy and the integration makes the estimate drifts from reality over time. Because of this other sensors are needed to grasp the actual environment state.
A second source of information, a very common and cheap solution, is camera measurements. The provided data is much more complex and rich than the IMU data, but comes with lower frequencies and embeds higher-level information that has to be retrieved with complex algorithms. Moreover, the quality of camera information depends on environmental conditions and sometimes cannot be reliable, for example in low-light conditions or when experiencing motion blur. Combining these two main sources of information to compensate for respective flaws, a reliable state estimation of the drone and surrounding environment can be obtained.

The data gathered by the sensors are processed by the perception block and then passed to the planning module. Its role is to plan an optimal trajectory with respect to some desired metric, e.g. minimum time, able to satisfy the constraints imposed by the task setup and the hardware platform. Once the system has come up with a feasible path to achieve the goal, it has to follow it. The planned trajectory is obtained by controlling the thrusters accordingly.

The controller subsystem is often divided into two cascade structures referred to as high-level controller and low-level controller. Typically the high-level controller receives as input the desired space trajectory and the velocity and acceleration profiles coming from the planning algorithm and generates desired virtual inputs, or lower level of abstraction reference signals, such as linear velocities or body rates and collective thrust, which then are passed down to a low-level flight controller that directly controls the individual rotors of the quadrotor [26].

Model-based approaches rely on the knowledge of the system's mathematical model but is not always simple to come up with a model for complex systems without introducing undesired approximations. Moreover, these control techniques have been demonstrated to achieve outstanding performances in scenarios where the system and environment models are considered to be perfectly known, but they miss the ability to adapt to model mismatches and unseen, noisy scenarios such as outdoor environments.

The model-based control pipeline has to be carefully designed depending on the application and needs fine-tuning adjustments guided by experience and heuristics. This is a time-consuming procedure that has to be repeated for every new setup.

For this reason, learning-based approaches have gained more and more traction, requiring less engineering effort during the tuning phase and being more robust to inaccurate models.

Learning-Based Approach

These approaches replace the perception, planner, and controller blocks with a neural network. As explained in section 1.1.1 deep neural networks are the most powerful tools to perform function approximation [27] and for this reason, can substitute some of the model-based blocks or even all of them finding input-output mapping functions.

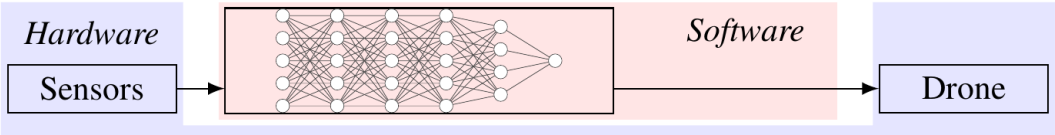


Figure 1.9: End-to-End learning approach [25].

Neural networks can deal easily with both high-dimensional and low-dimensional input data such as images, position, and velocity. The biggest challenge for learning-based methods is to collect enough data that guarantee the neural network training success. Moreover, the gathered data should be generated with the right amount of randomization to guarantee the generalization property and avoid overfitting a particular setup. Finally, interactions should be as similar as possible to real-world scenarios so as to reduce differences with reality and simplify the deployment on the actual hardware.

Figure 1.9 outlines one of the most attractive autonomous navigation paradigms, the one that most resembles the human pilot approach. It is called end-to-end flight and consists in one or more neural networks that map raw sensory data directly to control commands. In this particular architecture is the neural network that directly has to build a hierarchical internal representation of the task setup and relate observations to perceive what is happening in the surrounding, understand its goal and execute proper actions to achieve it as an emerging behavior of the artificial neurons interactions. The problem with a learning-based controller is that it can be difficult to prove its robustness as done with traditional methods. While this approach may provide superior performance to classical methods, there is the possibility that they cannot be used in practice due to the inability to provide stability properties. Even though their limitations, research is evolving in the direction of learning-based and end-to-end approaches thanks to recent advancements in simulation technologies, which can guarantee a great amount and variety of data to train neural networks, and their proven adaptability to large disturbances and parameter variations [28].

1.4 Simulators

Simulators are powerful software tools that are used across all engineering fields to develop and test solutions to complex real-world problems without the need for actual

hardware implementation, saving time and money. They are essential to implement robot learning solutions based on reinforcement learning algorithms. This is because the robot learns by interacting with the environment via trial-and-error, and so it needs to fail. Failing in the real world would mean damaging the hardware platform which could then lead to the partial or total replacement of it. For example, when a quadcopter learns to fly using RL in the first training iterations will continuously crash trying to achieve flight stability. Robotics simulators avoid the unnecessary effort of sacrificing the hardware that would be otherwise unaffordable.

The ideal simulator for robotics applications is fast, to collect a large amount of data since RL algorithms are not sample-efficient and need a lot of interactions to find a suitable policy. To cope with this, simulators are able to parallelize the workload across many computing units speeding up the time required to gather interactions and feedback and making the training iterations converge rapidly to a solution. Most of the simulator architectures rely on a combination of CPUs and GPUs to run a reinforcement learning system. CPUs are used to simulate environment physics, calculate rewards, and run the environment, while GPUs are used to accelerate neural network models during training and inference, as well as render graphics if required. Another property of an ideal simulator is to be physically accurate and to represent the dynamics of the real world with high fidelity. Simulators are designed to mimic real-world scenarios by adopting detailed physical simulations. Rigid and flexible body motion is reproduced by simulating all the forces that act on the recreated system. The last important property required is to be photo-realistic, to minimize the discrepancy between simulated and real-world sensors' observations when the use of a camera is necessary. Being photo-realistic means the ability of the simulators to reconstruct different lighting conditions with refractions, reflections, and visual distortion effects and to reproduce different kinds of textures on the simulated bodies' surfaces.

Those objectives are generally conflicting in nature: for example, the more a simulation is realistic, the slower it is. Therefore, achieving all those objectives in a single simulator is challenging. In the following sections, there will be given an overview of some simulators.

IsaacGym

The Isaac Gym [9] learning platform provides a high-performance solution for training RL agents to perform diverse robotics tasks on an Nvidia GPU. This platform enables both the physics simulation and neural network policy training to be executed on the GPU and communicate with each other directly, thereby eliminating any CPU bottlenecks. Consequently, complex robotics tasks can be trained on a single GPU in significantly less time than traditional RL training methods that rely on a CPU-based simulator and GPU for neural networks. The scheme in figure 1.10 highlights

the differences in training times between the two approaches.

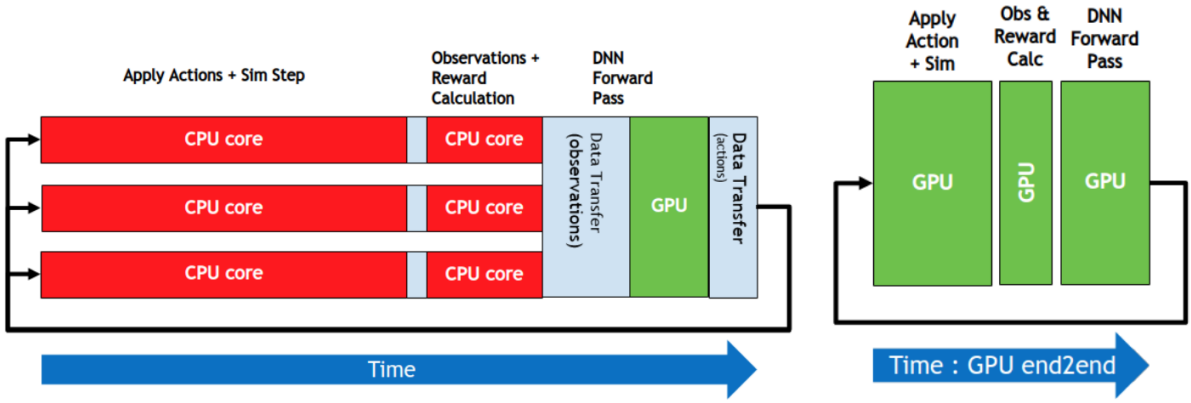


Figure 1.10: (Left) Traditional RL experience collection pipeline. (Right) Isaac Gym RL experience collection pipeline [9].

Isaac Gym provides utilities that can wrap the raw data buffers, implanted in C++ and CUDA, as tensor objects in common machine-learning frameworks like PyTorch. Moreover, this simulator provides an API for creating and populating a scene with robots and objects that are rendered in a photo-realistic way. Each environment can be duplicated many times to parallelize the workload and data-gathering while preserving the ability for variations between copies needed to randomize the setups and avoid overfitting. Thanks to this feature Isaac Gym has been demonstrated to be one of the most promising photo-realistic simulators for reinforcement learning applications [29]. Since it is a relatively new platform, there are not many ready-to-use task-specific features, e.g. flight controllers o different kinds of sensors, contrary to other simulation software.

Gazebo

Robot Operation System (ROS) [30] is the most popular software framework for robot programming and control. Generally, it is used with the Gazebo simulator for various robot simulations such as manipulation and mobile robot path planning. Gazebo is designed to accurately reproduce the dynamic environments a robot may encounter but cannot reproduce photo-realistic environments [31]. Moreover does not have built-in features to parallelize the workload across many computing units [31], essential in RL applications.

Flightmare

Flightmare [32] is a flying robot simulator that is composed of two main components: a configurable rendering engine built on Unity that can provide photo-realistic simulations and a flexible physics engine for dynamics simulation. Those two components are

decoupled and can run independently of each other. Flightmare comes with a sensor suite, including RGB images, IMU, depth, segmentation, and API for RL that can simulate quadrotors in parallel using the classical simulation pipeline that exploits CPU and GPU.

Chapter 2

Related Work

Many recent research works investigate advanced AI algorithms and optimal control techniques to achieve agile flight. Developing quadrotors that are fully autonomous and that can approach, or outperform, the agility of birds or human drone pilots is very challenging and, to date, there is no framework that completely solves this problem.

State-of-the-art work faces this problem by splitting the control stack into consecutive perception, planning, and control blocks [25], [33], [34], [35]. This division of the navigation pipeline enables parallel progress on each component and makes the overall system interpretable. This is because it leverages the use of analytically proven control algorithms' stability properties, making it possible to implement such solutions on safety-critical systems [25].

As an example, [33] uses this approach to investigate agile flight in a drone racing setup, in which a quadrotor has to traverse moving gates along the race track navigating with visual information. To solve the task, a Convolutional Neural Network (CNN) is combined with a state-of-the-art path-planning and control system [36]. The CNN, whose architecture was first proposed in [37], directly maps raw images into a representation in the form of a waypoint and desired speed. This information is then used by the planner to generate a short, minimum-jerk trajectory segment that is tracked by a low-level controller to reach the desired goal. Although simple and effective, such an approach typically discards interactions among the different blocks and requires each block to make over-simplifying assumptions [38]. Moreover, a dedicated perception pipeline could have high computational costs, as in [39], which introduces a feature-based monocular Simultaneous Localization And Mapping (SLAM) system that operates in real-time, that requires a high-end processor coupled with laptop-sized RAM. Finally, due to the presence of sequential processing blocks between sensors and actuators, the latency to go from observation to action increases at the cost of agility, which is the fundamental requirement in a dynamical obstacle avoidance setup [40]. Due to these intrinsic limitations, most of the research has moved towards end-to-end learning-based approaches, where the policies are trained directly from data without

explicit perception and planning stages. Neural networks have a strong potential to control agile platforms like quadrotors. In comparison to traditional methods, neural policies are more robust to noise in sensor observations and can deal with imperfection in the actuation [38].

The authors of [41], use deep reinforcement learning for training vision-based navigation policies entirely in simulation, and then, by highly randomizing the simulated settings, transfer them into the real world to achieve flight with only synthetic training images. The collision avoidance policy is represented by a deep convolutional neural network that directly processes raw monocular RGB images and outputs velocity commands. This approach has been successfully deployed in the real world but the obtained policies result in slow navigation speeds and discrete action space, making it unable to meet the agile navigation requirements. In [42] the authors demonstrate that end-to-end policies trained in simulation enable high-speed autonomous flight through challenging environments and outperform traditional obstacle avoidance pipelines.

2.1 End-to-End Control

The work [26] shows that learning-based end-to-end policies, trained with the PPO algorithm, outperform model-based solutions, in agile flight tasks with non-nominal scenarios, where some physical parameters are uncertain. Moreover, the authors compared three different control strategies, depicted in figure 2.1, for agile quadrotor flight called: linear velocity control (LV), collective thrust and body rates control (CTBR), and single-rotor thrust (SRT).

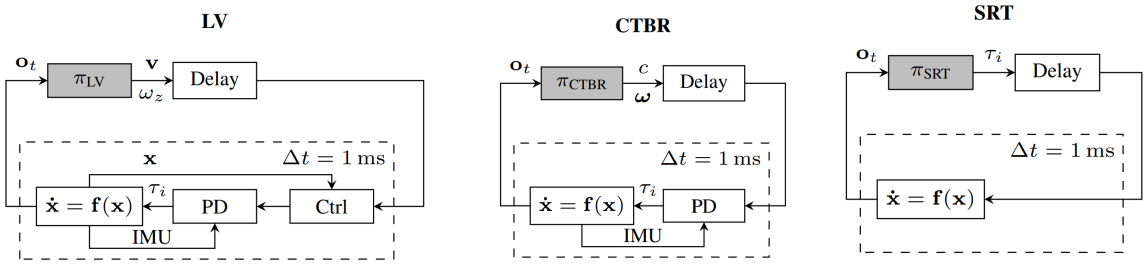


Figure 2.1: Different control strategies for end-to-end NN [26].

Control policies specifying linear velocity are the most high-level of the three approaches and do not take into account the dynamic constraints of the platform. They rely on the existing flight control stack, and while being robust, cannot achieve the best performance. On the other hand, single-rotor thrust policies are the most capable of performing aggressive maneuvers but present a lack of robustness in case of parameter mismatches.

In the end, the authors show that the CTBR approach is more robust in sim-to-real transfers compared to the SRT approach and is more performant with respect to LV

control, representing a good trade-off between safety and agility. For this reason, the experiments carried out in this thesis work will adopt the CTBR approach to achieve autonomous end-to-end navigation.

2.2 Agile Flight Benchmark

In work [43], a hybrid model-based and learning-based approach, has been applied to solve a challenging agile flight benchmark in the real world. This task consists of a quadcopter that has to traverse a swinging gate to reach a goal hovering position behind it, starting from different initial positions in space. This proposed environment setup is the one that has been chosen to develop this thesis work.

The authors managed to solve this environment with a proposed state-of-the-art solution: training a neural network to produce high-level decision variables, instead of training a neural network to map directly raw input data to control action. The decision variable, which corresponds to a forecasted traversal time of the gate, is passed to a model-based optimal controller, Model Predictive Control (MPC), that produces actions that minimize a loss function. This approach has been proven to work both in simulation, with multiple gate traversals, and in reality, with one swinging gate.

This thesis work aims to answer the question if the same benchmark can be solved using an end-to-end learning-based approach to obtain a solution without the need for separate planning and control modules. Moreover solving the task in that way would be desirable since the MPC algorithm is resource hungry and cannot always be run on an embedded platform, such as a nano-drone or other resource-contained robotic platforms. Furthermore, it requires more time to produce control commands with respect to the inference of a neural network, due to the fact that MPC has to solve iteratively an optimization problem [44]. In this thesis, the policy able to solve the task is found with a reinforcement learning algorithm as an emergent behavior from the physical setup and reward function implemented. The training is guided toward the solution using different reward functions. Once the policy is obtained, and so are the weights of the network, no optimization algorithm has to be solved.

Chapter 3

Methods

The main goal of this thesis work is to recreate the setup of agile flight through a fast-moving gate, proposed in [43], and train a DNN to predict actions in the collective thrust and body rates action space [26] starting from the state of the system, without explicit planning and perception stages and resource hungry optimization procedures. To learn an end-to-end policy, the reinforcement learning loop depicted in figure 3.1 has been set up. Its main function is to collect the necessary data from the parallel simulator and update the policy parameters. NN agents are tested along the training phase to save the best-performing model.

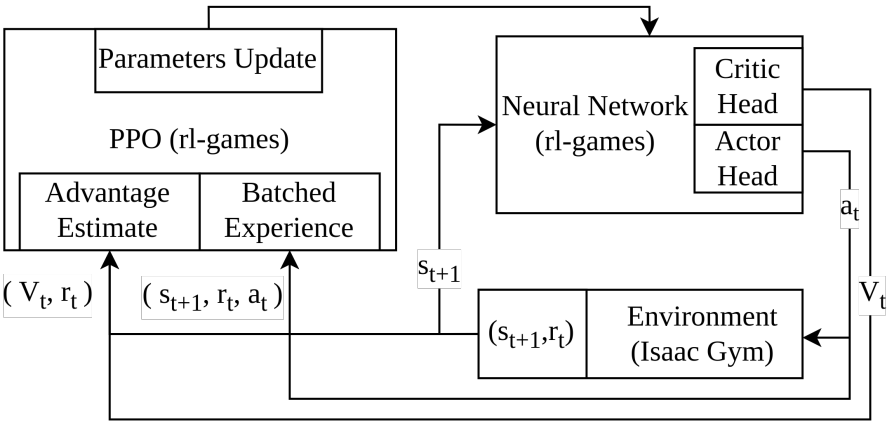


Figure 3.1: Reinforcement learning framework to train the neural network solving the task.

Isaac Gym divides the simulation into three main steps: *create simulation* step, *pre-physics* step, and *post-physics* step. Once the environment is built in the *create simulation* step, the software transitions to the *pre-physics* step, where the agent computes an action a_t and the value V_t based on the state of the system. The action is then transformed into forces or torques and passed to the physical simulator that returns the new state of the system s_{t+1}. Finally, in *post-physics* step, the new state is used to

compute the reward value r_t and is fed to the agent which can choose again a new set of actions to perform.

In the meantime *rl-games* [45], a highly-optimized RL library compatible with the Isaac Gym framework, gathers experiences and advantage values, estimated through the critic outputs, to compute the loss function and performs batched updates of the NN parameters.

3.1 Environment setup

The Isaac Gym simulator has been used to recreate the agile benchmark of the work [43] and is depicted in figure 3.2.

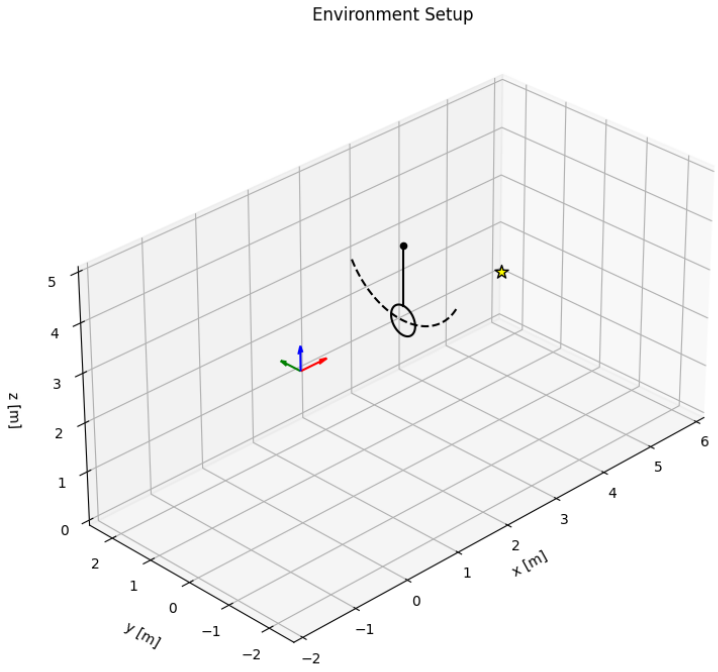


Figure 3.2: Simulated Environment setup with drone frame, swinging gate and goal poses.

The setup consists of a circular gate attached to a pendulum, that can swing along a plane, and a drone platform, that starting from one side of the obstacle has to traverse the gate to reach a target hovering position behind it. The pendulum, in figure 3.3, is composed of a bar that has one of its extremities attached to a revolute joint fixed in space at position $[x = 2.0\,m,\ y = 0.0\,m,\ z = 4.5\,m]$, while on the other extremity, there is attached a circular gate.

When the simulation is started, the pendulum starts swinging in free fall conditions and during the motion is subjected only to friction forces. The hinge and the pendulum

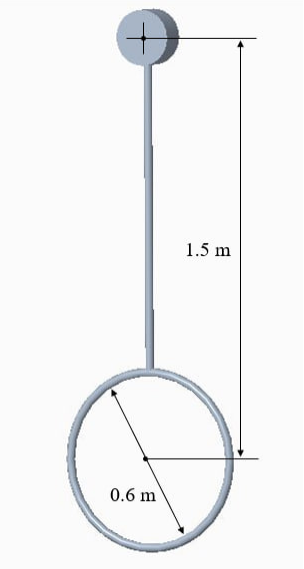


Figure 3.3: Gate model with diameter 0.6 m and bar length 1.5 m.

3D models have been created via CAD software and have been composed together using the Universal Robotic Description File (*.urdf*). This file describes the dynamical properties of the mechanical system (e.g. mass, moment of inertia, joint friction) and defines how the mechanical parts are geometrically composed together.

The drone platform considered in the experiments is the Crazyflie 2 [46], an open-source nano-drone vastly adopted in the aerial robotics community. The considered physical drone parameters ([47], [48]) are reported in table 3.1, and its simulated model has been made available by [48]. This platform is adopted because of the availability of its physical characteristics and model. When transferring this work into a real-world application further investigations on the right quadrotor platform for agile flight should be done.

Parameter	Value
Mass [kg]	0.027
Inertia [kgm^2]	[2.3951e-5, 2.3951e-5, 3.2347e-5]
Gravity [ms^{-2}]	[0.0, 0.0, -9.81]

Table 3.1: Physical parameters of the simulated drone.

The quadrotor starting position corresponds to $[x = 0.0\,m,\ y = 0.0\,m,\ z = 3.0\,m]$, while its initial orientation is parallel to the ground with the x-axis aligned with the world's one. The target to reach is position $[x = 4.0\,m,\ y = 0.0\,m,\ z = 3.0\,m]$, behind the moving pendulum.

This custom environment has been replicated 4096 times inside the simulation. After hyperparameters tuning, this was the highest number of environments that contributed to faster training iterations, while an additional increase did not involve any

successful improvement. The designed environments are played in parallel by the agent using one Nvidia GPU RTX A6000 on a remote cluster which can be accessed through *ssh* network communication protocol. This GPU is compatible with Isaac Gym which can generate tons of data in minutes by leveraging the full GPU pipeline.

3.1.1 Reset Policy

In reinforcement learning is crucial to constrain the agent's physical behavior to avoid unrealistic and dangerous actions due to the approximation done by the physics simulator. Moreover, an accurately designed reset strategy, together with the reward function, guides the agent in learning the desired policy.

The environment is restored whenever the drone flies too far from the target, to limit its movements around the point of interest, or when its altitude is too low with respect to the ground, to create a safety boundary. A reset occurs also when the quadcopter collides with the gate, and when roll or pitch orientations overcome a safe inclination angle with respect to the ground. All the considered reset conditions boundaries are reported in table 3.2.

Around the simulated drone is defined a collision mesh larger than the mechanical structure occupancy. This surface is exploited by the simulator to detect potential collisions and is useful to create a safety margin during traversal moves.

Condition	Value
Max drone-gate distance [m]	20.0
Min altitude [m]	0.2
Max roll module [rad]	$\pi/2$
Max pitch module [rad]	$\pi/2$

Table 3.2: Reset conditions boundaries.

3.1.2 Training Randomization

All the simulated environments have been randomized at training time in order to avoid overfitting a particular configuration of the starting pose of the drone, the starting height of the pendulum. The hovering target remains fixed.

The randomization is carried out in two stages. The first one is the one used to generate randomization of the setup among all the environments, in particular by varying the initial inclination of the pendulum, which will remain the same for all the remaining training iterations.

The second type of randomization is the one that occurs each time that an agent arrives at a terminal condition (e.g., the termination of the episode length, or crashing).

In this case, an environment has to be restored to its specific initial condition. When this occurs, the new initial pose of the quadrotor is varied with respect to the starting pose of the first iteration.

All the randomized variables are sampled from a uniform distribution in a symmetric range about the nominal values, as specified in table 3.3.

Parameter	Value	Randomization Range
Initial gate angle θ [rad]	0.0	$[-\pi/2, \pi/2]$
Initial drone x position [m]	0.0	$[-3, 1]$
Initial drone y position [m]	0.0	$[-1, 1]$
Initial drone z position [m]	3.0	$[-1.5, 0.5]$

Table 3.3: Nominal values and randomization ranges of the environment configuration parameters at training time.

3.2 Agent Setup

The agent is controlled by a deep neural network. It is initialized by the reinforcement learning library used to train it, *rl-games*, via the configuration file. It specifies the hyperparameters for the neural network architecture, such as the hidden layer dimensions, or the learning rate, and the RL algorithm, e.g. the actor-critic architecture.

3.2.1 NN Architecture

The deep neural network is a fully-connected neural network with 3 hidden layers of dimensions [256, 256, 128] and separated actor and critic heads. The input layer receives from the simulator the state of the system which consists of the position of the drone, its orientation, its distance from the swinging gate center, the drone's linear velocities and body rates, and the linear velocity of the gate center. The total dimension of the input layer, which has to be equal to the input dimension, is 19 since each piece of information in input is a 3-dimensional vector, except the quadrotor orientation, which is expressed through a quaternion and has 4 dimensions. The output layer of the critic's head has size 1 and outputs the estimated value function while, the output layer of the actor's head has size 4 and outputs the lift of the drone and the roll, pitch, and yaw rates. The control commands are then passed to the low-level flight controller to find the resulting torque and lift applied by the chassis.

3.2.2 NN Training

The training procedure is implemented using the reinforcement learning algorithm PPO. This is executed by the *rl_games* library, which manages all the data coming from the simulated environments. The hyperparameters used for the PPO algorithm can be found in table 3.4.

Parameter	Value
γ (discount factor)	0.99
Learning rate	1e-3
Entropy regularization	0.0
ϵ (importance ratio clipping)	0.2

Table 3.4: Training hyperparameters.

The total duration of the training phase has been chosen as 1000 epochs since additional iterations resulted in the agent making no progress. Each epoch consists in playing 16 simulation steps among all the 4096 parallel environments, so in the end the total amount of training steps is equal to 65 million. To conclude, the maximum game duration selected equals 700 steps and, since the simulation runs at 60Hz, the maximum total time that an episode could last is almost 12 seconds. This time is sufficient for the drone to traverse the gate and reach the final position.

3.2.3 Low-Level Flight Controller

The low-level flight controller is what connects the neural network to the drone actuators and is implemented to produce actions in the CTBR action space explained in section 2.1. For this reason, the low-level controller has to be able to translate roll, pitch, and yaw body rates and the desired lift into actual rotor thrusts. To compute them, the equations 1.39 and 1.40 have to be solved. In the experiments performed the resulting torque and linear force are directly applied to the drone's center of mass since from a mechanical point of view force applied to a rigid body can be always described by a resulting force and a total torque. This is done to overcome a bug in the API provided by Isaac Gym that verifies when is required to apply a particular force into a point of the rigid body and, at the same time, simulate with the full GPU pipeline.

The mass normalized lift c is directly computed by the network, while the low-level flight controller, designed starting from [23], is in charge of mapping body rates to torques. It implements the following equation

$$\tau_{des} = J \cdot P \cdot (\omega_{des} - \hat{\omega}) + \hat{\omega} \times J\hat{\omega} - J \cdot D \cdot \frac{d}{dt}(\omega_{des} - \omega_{des,t-1}) \tag{3.1}$$

where $\hat{\omega}$ are the measured body rates, ω_{des} are the body rate signals coming from the network, and $P = diag(p_{pq}, p_{pq}, p_r)$ is a diagonal matrix containing the proportional

gains of the roll, pitch, and yaw rates. Roll and pitch gains are equal since the considered drone platform is symmetric along the planes xz and yz. The proportional gains have been selected as $p_{pq} = 25$, $p_r = 10$, following a trial and error procedure. The matrix $D = diag(d_{pq}, d_{pq}, d_r)$ is a diagonal matrix containing the derivative component gains of the roll, pitch, and yaw input rates, selected as $d_{pq} = 1$, $d_{pq} = 1$, $d_r = 1$.

Finally, equation 3.1 implements a feedback linearizing control scheme, since both, the inertia matrix of the drone J and the term $\hat{\omega} \times J\hat{\omega}$, go in cancellation when plugging the desired torque τ_{des} into equation 1.37.

The overall low-level control scheme is depicted in figure 3.4 and consists of a PD controller in parallel form with feedback linearization. The derivative term is applied on measurement [49] to avoid the *derivative kick* phenomenon, which is a spike in the derivative of the error when any change in the setpoint occurs. This really high derivative is then fed into the PD equation, which results in an undesirable spike in the output.

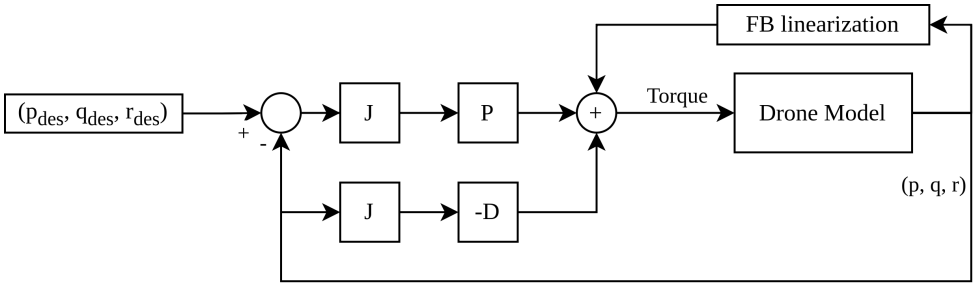


Figure 3.4: Block scheme of the low-level flight controller.

This controller has been implemented from scratch using the PyTorch python package [50] to maintain the end-to-end GPU pipeline implemented by the Isaac Gym simulator. This package is built to make the most efficient use of the GPU parallel architecture, providing custom methods to execute parallel operations. In particular, the *torch.einsum* method has been used, since it allows computing many multi-dimensional linear algebraic array operations based on the Einstein summation convention in parallel, together with the *torch.cross* method for fast parallel matrix-vector products. Removing sequential *for* loops operations make it possible to execute every operation, among thousands of environments, inside the GPU without CPU bottleneck and achieving training phases that last minutes.

3.3 Reward Functions

The agent learns indirectly from the gathered rewards during the interactions with the environment. The design of a reward function to guide the training is crucial for obtaining good results. In the experiments, three different reward functions have been

compared to study the best way to guide the agent during the training and obtain a strategy that is able to adapt to different unseen scenarios.

These functions are built on top of each other with successive functions that add terms to further instruct the agent to discover the policy that solves the task. The reward is computed at every time step depending on the state of the system and the total score is the sum of every reward received until a termination condition is reached.

3.3.1 Target Reward

The first reward function considered, called target reward, is the one that gives positive feedback to the agent when it minimizes its distance from the hovering target and achieves stable flight. No information or reward is returned for passing through the gate. Even if is unlikely that the agent learns the desired behavior, it is useful to have a baseline to compare with and to check if the agent is able to figure out how to pass the obstacle without any guidance.

The formulation of this first function is the following:

$$r_{target,t} = r_{dist,t} + r_{dist,t} \cdot (r_{up,t} + r_{spin,t}) \tag{3.2}$$

The agent neural network is initialized with random weights and for this reason, the output actions at the beginning of the training phase are random. This fact means that the quadrotor is not even able to stabilize itself and hover. The reward terms r_{up} and r_{spin} are what encourage the agent to learn stability.

The r_{up} component is defined as

$$r_{up} = \frac{1}{1 + tilt^2} \tag{3.3}$$

where $tilt = 1 - k \cdot \hat{k}$, represents the amount of misalignment between the unit vector k of the quadrotor z-axis, with the unit vector $\hat{k}$ of the world z-axis. When the two z-axes will be aligned this term will be equal to 1, while the greater the misalignment is the lower will be its value. The agent is encouraged to stay parallel to the ground with the rotors pointed upward each time step to cumulate points and, as a result, achieve flight stability.

This reward term does not control the yaw body rate, meaning that the drone is free to spin around its z-axis. This condition has to be avoided because would result in an impossible flight primitive. For this reason, the component r_{spin} is introduced.

$$r_{spin} = \frac{1}{1 + r^2} \tag{3.4}$$

The yaw body rate of the drone, r, has to be selected as low as possible to maximize the fraction, similar to the misalignment coefficient.

The combination of these two reward terms is what makes the agent able to achieve stable flight since there is no underlying controller that solves this problem.

Aside from being able to hover parallel to the ground, the quadrotor has to reach the target position. The reward term r_{dist} is what pushes the agent toward that goal and is defined as

$$r_{dist} = \frac{1}{1 + dist^2} \tag{3.5}$$

where the variable $dist = \sqrt{(p_{target})^2 - (p_{drone})^2}$ is the distance between the target position and the drone position. The agent receives positive reinforcement if it is able to minimize its distance from the goal position. Moreover, r_{dist} appears as the weight of spin and uprightness terms. Since could get values that belong to the interval [0, 1], it scales down the importance of getting high scores for spin and uprightness until the goal distance is minimized, encouraging the maneuvers needed to reach the final position.

3.3.2 Sparse Reward

This second reward function candidate is composed of the target reward plus the term r_{gate}, which reinforces those agents that manage to traverse the gate:

$$r_{sparse,t} = r_{gate,t} + r_{target,t} \tag{3.6}$$

The component r_{gate} has been designed as a one-time score obtained when the following condition is met:

$$\|p_{gate} - p_{drone}\| \leq 0.1\, m \tag{3.7}$$

where p_{gate} and p_{drone} are respectively the positions of the gate centers and the drone's CoM in space. If this condition is satisfied, it means that the drone has traversed the gate, thus it obtains a positive reinforcement. The traversal is reinforced when occurs with a $0.1\, m$ error around the gate center to guarantee safety margins between the drone and the gate's internal boundaries.

This reward term has been chosen two orders of magnitude higher compared to the other terms to stimulate agents in taking the risk of traversing the gate. The quadrotor is discouraged to fly near the obstacle since a collision would result in the reset of the environment and, consequently, in a less cumulative final score.

One last design choice for this reward term is in its sparsity. The reason for this implementation is to not give any bias to the drone on how to reach the gate and traverse it. Rather than heuristically guide the quadrotor toward a solution, the agent is left free to discover how to reach it and obtain the flight trajectory as an emergent solution from the constraints.

3.3.3 Synchronization Reward

The third and last reward function is defined as

$$r_{sync,t} = r_{sync_y,t} + r_{sync_z,t} + r_{sparse,t} \tag{3.8}$$

where, aside from the previously presented component $r_{sparse,t}$, two more terms are introduced: $r_{sync_y,t}$ and $r_{sync_z,t}$. Similarly to what has been done already, they are defined so as to have maximum in one:

$$r_{sync_y} = \frac{1}{1 + dist_{gy}^2} \ , \tag{3.9}$$

$$r_{sync_z} = \frac{1}{1 + dist_{gz}^2} \tag{3.10}$$

The agent is encouraged to minimize its distance z and y components to synchronize with the gate oscillation that occurs in the zy plane. More information on how to solve this task is given with respect to the sparse reward. In the end, the motion along the x-axis is mainly governed by the distance from the target reward, while the motion along the zy plane is reinforced to be similar to the one of the dynamic gate. This is the most informative function of the three.

Chapter 4

Experimental Results

In this chapter, the training outcomes and the experimental results are presented by comparing the policies trained with the reward functions described in section 3.3: the target reward, which scores only the flight stability and the distance to the goal, the sparse reward, which in addition to the previous one, scores the gate traversal, and finally the sync reward, which sums to the former functions a term that reinforces the synchronization with the swinging motion of the gate. Furthermore, the results of a random policy baseline are displayed. In particular, is shown that the sync reward outperforms both, the sparse reward and the target reward functions in guiding the agent during the training phase, proving great performances in task generalization. The obtained results are promising for further development of end-to-end autonomous agile flight tasks and open the way to new experiments needed to transfer this work into the real-world scenario.

4.1 Training Results

The agent training converged successfully for each of the three designed reward functions since all the agents reached their respective scores to solve the environments. The related learning curves are reported in figure 4.1.

The reward scores depend on the function used to score the agent's policy, so they cannot be used to compare the agent's performance. For this reason, an objective experimental setup is required. Furthermore, at training time, the agent keeps exploring the action space sampling the probability function learned by the agent, as explained in section 1.2.4, which does not lead always to the best choices. At test time the policy will output only the most probable action without any stochastic behavior, leading to a consistently good score.

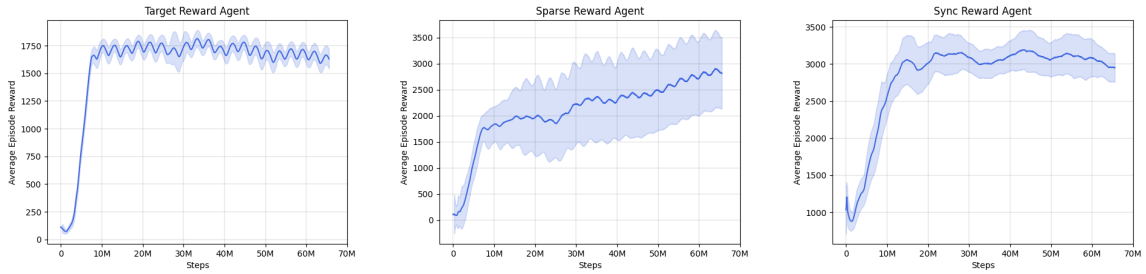


Figure 4.1: Learning curves of the agents trained with the three different rewards functions, showing smoothed mean performance and standard deviation. Each agent is trained for a total of 65 million environment interactions.

4.2 Experimental Setup

The learned policies have to be tested and compared in order to evaluate their performance and find the best reward function needed to solve the task considered.

The test procedure consists of each agent trying to solve the environment 5000 times with two different configurations. For the first round of tests, the environment is randomized using the training ranges in the selection of the drone's starting position. For the second one, the randomization ranges are increased to evaluate generalization to unseen initial poses. Whenever the agent passes through the gate and terminates the episode hovering around the target, one success is counted. If the agent crashes or does not traverse the gate will result in a failure. Crashes are detected by checking whether the duration of the episode is inferior to 700 steps.

4.3 Training Scenario Evaluation

Table 4.1 shows the evaluation results for the agents trained with the three different reward functions plus the random baseline. The reported tests have been performed with the randomization ranges of the training scenario, as reported in section 3.1.2.

The agent initialized with random weights ends each test episode with a crash. This confirms the fact that the quadrotor without any underlying high-level flight controller has to learn to stabilize itself before starting to solve the task. On the contrary, the agent trained with the target reward function is not able to solve the task but has learned to fly toward the goal and hover in that position. The obtained results demonstrate that if the agent is not instructed to traverse the gate it will learn to avoid it. As a matter of fact, the number of crashes represents only 3.38% of the total tests with respect to the number of agents that finished the episode with no traversal, corresponding to 96.62%. Even if the plane where the gate swings is along the path, the drone does not collide. This behavior is emerging due to the fact that an agent

that crashes will accumulate less reward with respect to an agent that is able to end the game near the hovering position.

Reward Function	Success Rate	Crashed	Not Passed
Random	0.00% (0)	100% (5000)	0.00% (0)
Target Reward	0.00% (0)	3.38% (169)	96.62% (4831)
Sparse Reward	61.08% (3054)	15.04% (752)	23.88% (1194)
Sync Reward	97.52% (4876)	1.14% (57)	1.34% (67)

Table 4.1: Evaluation of the policy learned with different reward functions on the environment in training configuration. Success is counted when the agent traverses the gate and finishes the episode hovering. The "crashed" column reports the number of those agents that have not finished the episode. The "not passed" column counts those agents that managed to finish the episode hovering in the goal position but did not traverse the gate.

A quadrotor trained with the sparse reward function at the testing time is able to traverse the gate and hover 61.08% of the time, but crashes in 15.04% of cases: a higher percentage with respect to the 3.38% of the policy discussed above. This occurs because the agent has discovered that when passes through the gate, gathers one high reward and consequently becomes more prone to risk a crash trying to get it.

Finally, the agent educated with the synch reward reached a 97.52% success rate, solving the environment almost in all the encountered diverse situations. As it is possible to see in figure 4.2, regardless of the starting position the drone is able to execute the necessary maneuvers to anticipate the pendulum movements and traverse it. It is able to adapt its linear velocity along the path and to encounter the gate at the right time. This emergent behavior is shown in the trajectory plot 4.2c, where the drone starts far from the gate with a high velocity, then, near the traversal point, slows down to stall and, at the right time accelerates as fast as possible to traverse the gate without colliding. This is done with an end-to-end neural network that directly maps sensory information into action to take, with no intermediate planning stages. It demonstrates that a neural network can be used to control a quadrotor effectively in a high dynamical scenario. Once passed through the gate, the drone redirects itself toward the hovering goal. By looking at another sampled trajectory, depicted in figure 4.2f, it is possible to note that in some situations the quadrotor is not able to hover steadily at a point. This behavior is due to the fact that even when it reaches the final position it continues to minimize its misalignment with the gate center. Further studies have to be done to try to reduce this oscillatory behavior, such as policy retraining or modifications to the rewards function that should let the agent ignore the synchronization reward term once the gate is traversed, similarly to what [43] did with their cost functions.

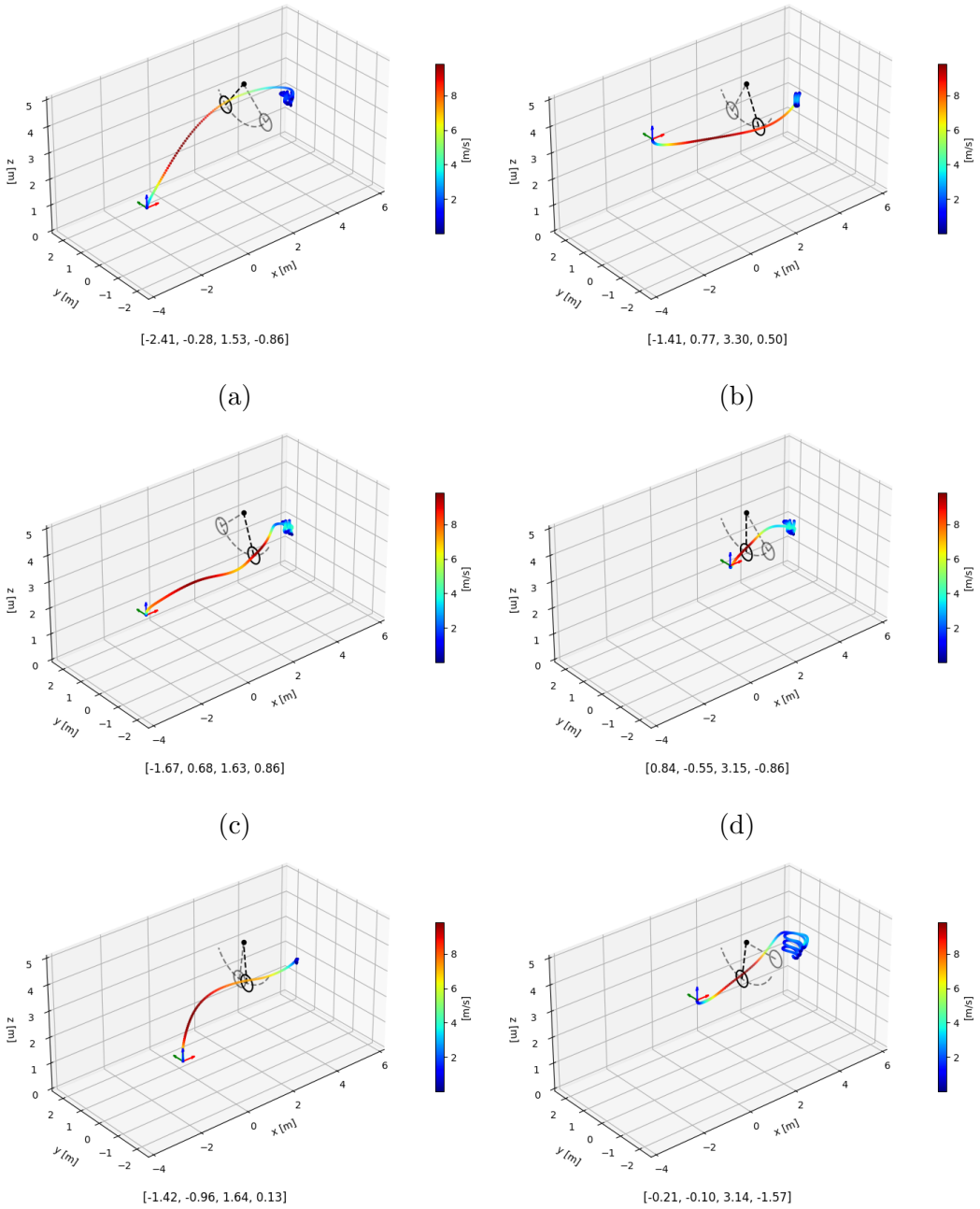


Figure 4.2: Evaluation trajectories of the trained policy. The quadrotor trajectories are colored by their speeds indicated in the color bar. The grey gate indicates the gate starting pose. The figures' titles specify the starting coordinates, respectively: the initial drone position (x, y, z) and the gate's initial angle in radians.

4.4 Generalization

The generalization capabilities of each policy candidate have been assessed by repeating the previous tests with wider randomization ranges, summarized in table 4.2.

Parameter	Value	Randomization Range
Initial gate angle θ [rad]	0.0	$[-\pi/2, \pi/2]$
Initial drone x position [m]	0.0	$[-4, 1.5]$
Initial drone y position [m]	0.0	$[-2, 2]$
Initial drone z position [m]	3.0	$[-2, 2]$

Table 4.2: Nominal values and randomization ranges of the environment configuration parameters in the generalization test.

The results obtained are reported in table 4.3. Even in this setup, the success rate of the random baseline and target reward policies has remained irrelevant, as expected. This is a more challenging case with respect to the one considered in section 4.3. It is possible to note that the success rates of the sparse reward and sync reward have dropped by 20% with respect to the tests in the training ranges.

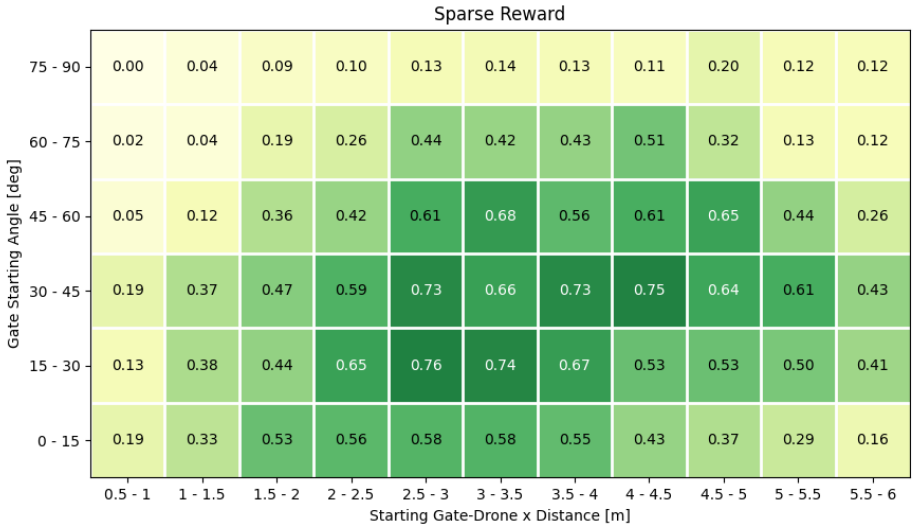
Reward Function	Success Rate	Crashed	Not Passed
Random	0.00% (0)	0.00% (5000)	0.00% (0)
Target Reward	0.12% (6)	8.20% (410)	91.68% (4584)
Sparse Reward	37.58% (1879)	26.04% (1302)	36.38% (1819)
Sync Reward	77.30% (3865)	10.52% (526)	12.18% (609)

Table 4.3: Evaluation of the policy learned with different reward functions on the environment in testing configuration. Success is counted when the agent traverses the gate and finishes the episode hovering. The "crashed" column reports the number of those agents that have not finished the episode. The "not passed" column counts those agents that managed to finish the episode hovering in the goal position but did not traverse the gate.

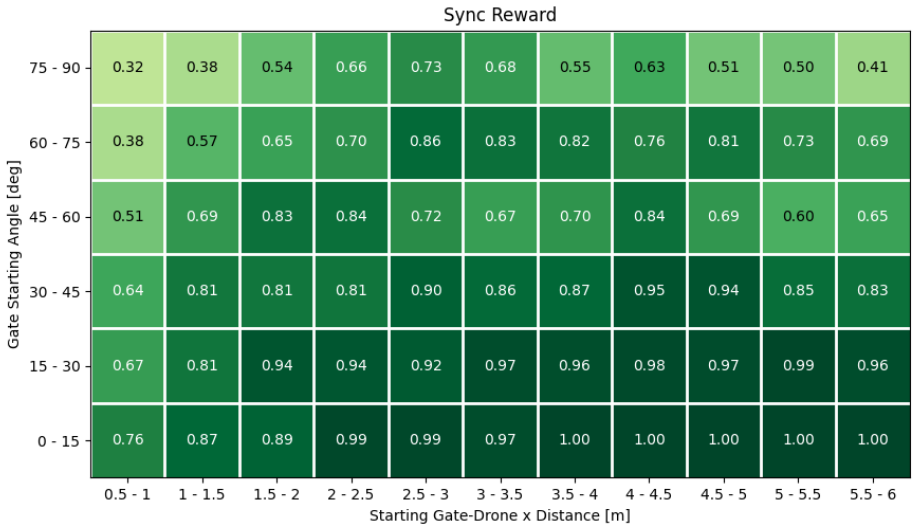
The number of trajectories that ended in crashes and the ones that did not pass through the gate increased by almost the same quantity respectively. In this circumstance, the evaluation is further investigated with the aid of heatmaps, in figure 4.3 and 4.4, to find evidence of which configurations of the environment are the most critical for the quadrotor.

Along the horizontal axis are reported separate slices of the 3D space obtained by varying, one at a time, the x, y, and z coordinates in intervals of 0.5 meters and leaving free the remaining two. Along the y-axis are captured different starting angular positions of the gate, from 0 to 90 degrees with steps of 15 degrees.

The approach using the sync reward function outperforms the sparse reward in all the possible starting configurations.



(a)



(b)

Figure 4.3: Evaluation heatmaps for different initial drone poses reported along the horizontal axis and for different gate starting angles reported along the vertical axis.

In figure 4.3b it is possible to notice that the sync reward agent has high success rates for those configurations with a small gate starting angle and a high gate-drone x distance. When the gate starts low will reach a limited maximum velocity making it simpler for the drone to center it. Moreover, a greater distance from the pendulum plane leaves more time for the agent to anticipate the oscillation and results in success in almost all cases, even in the unseen ones. On the other hand, the cases that most contribute to performance drop are the ones with higher gate starting angles and low gate-drone distances along the x-axis. This is due to the fact that the drone cannot accelerate fast enough to intercept the gate, or passes without traversing it, going

directly toward the target. The agent trained with the sparse reward function, as can be noticed from figure 4.3a, generalizes poorly to unseen situations. It retains good results in the distance range it was trained on and in those situations with a low inclination of the gate, which consequently reaches a slower swinging velocity.

The heatmaps in figure 4.4 report the success rates when the quadrotor has different offsets with respect to the gate center along the y and z directions and they show coherent behaviors with the one observed above, when the distance along the x-axis is changed. The drone trained with the sparse reward function (heatmaps 4.4a and 4.4b), has higher success rates when aligned with the gate resting position, again with difficulties in generalizing to unseen scenarios.

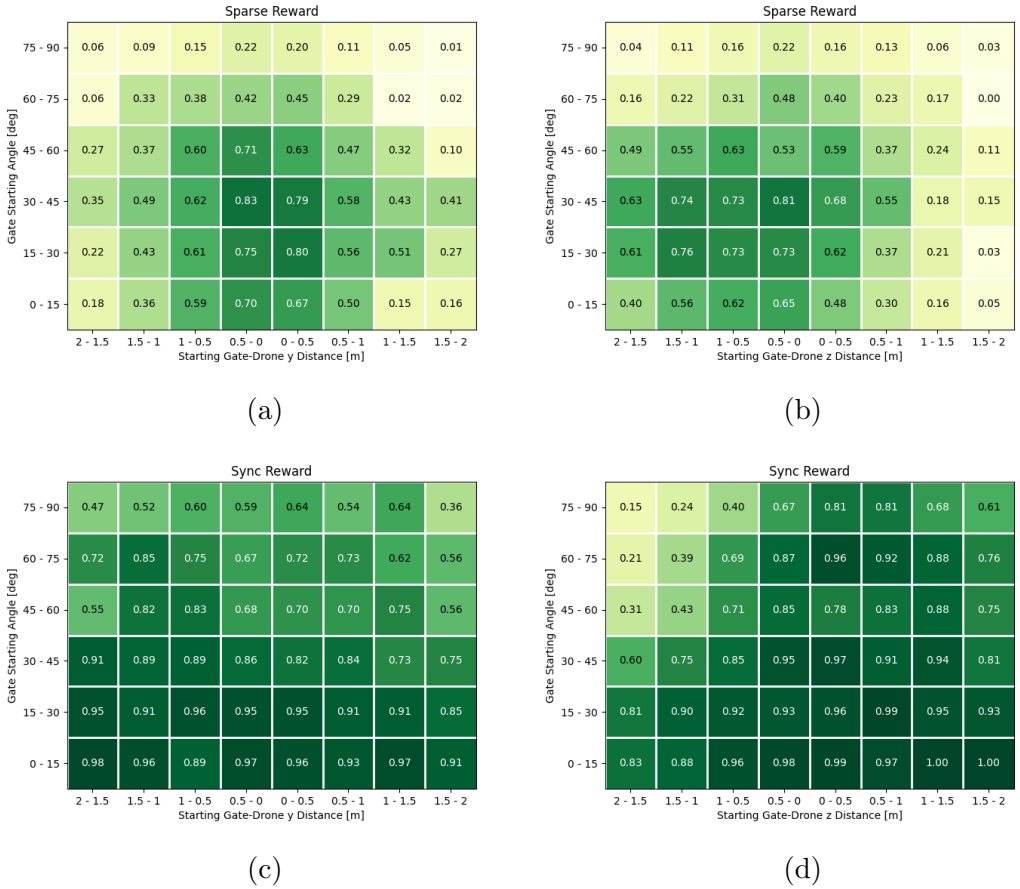
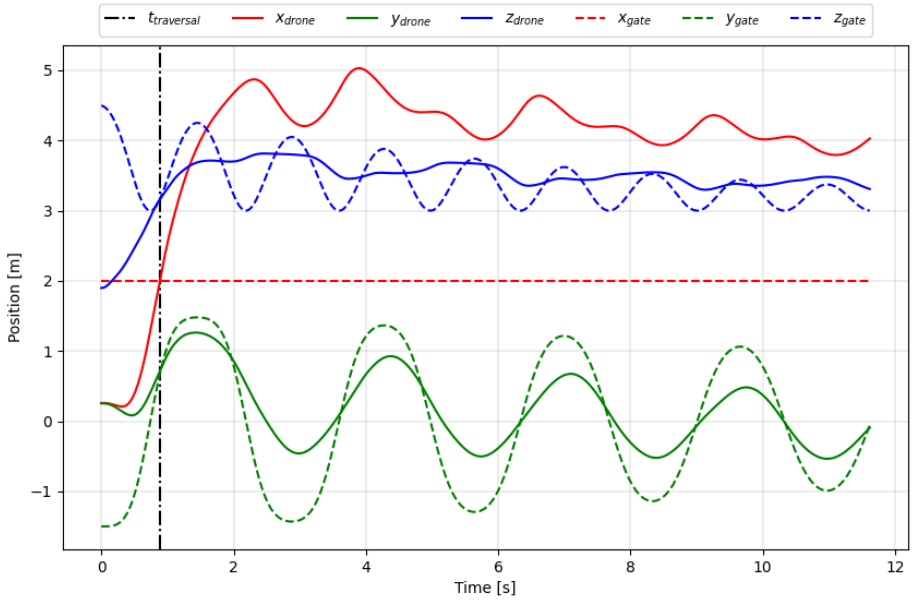
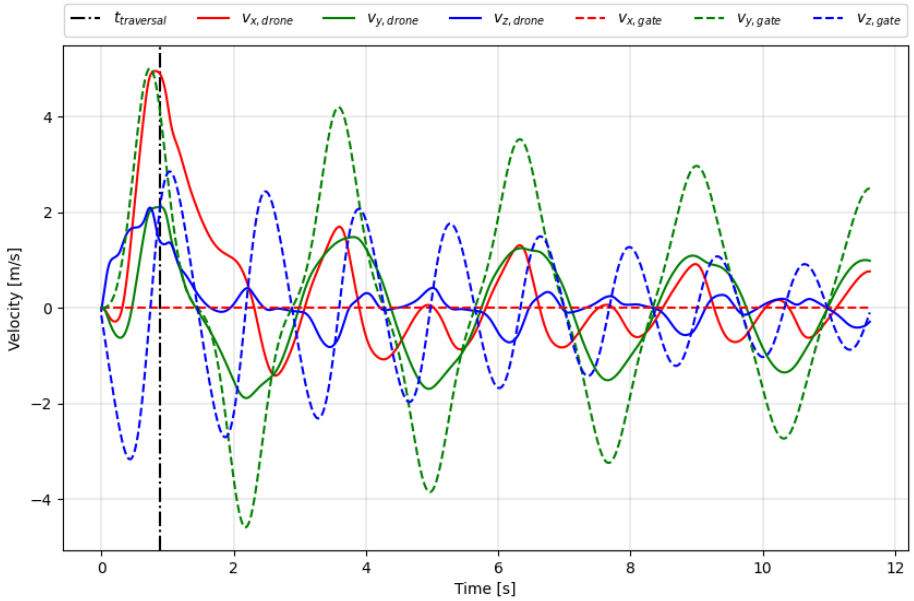


Figure 4.4: Evaluation heatmaps for different initial drone poses reported along the horizontal axis and for different gate starting angles reported along the vertical axis.

The left upper corner of the heatmap in figure 4.4d, represents the most critical condition for the agent trained with the sync reward function, and again together with figure 4.4c, underlies the difficulties in traversing the gate when it swings really fast and the starting point it too close to the pendulum plane. This could be a potential problem for those applications that present many fast-moving obstacles really close to each other.



(a)



(b)

Figure 4.5: Trajectories of the gate and the quadrotor with traversal time instant outlined by the black vertical line. The plots are obtained adopting the policy trained with the synch reward function.

To conclude, the trajectories followed by the gate and the drone are documented in figure 4.5. The traversal occurs at the time in which all three positions are aligned and is outlined by the black line. Before that time instant, the drone accelerates abruptly and then, after it is passed, rapidly decelerates to reduce the overshoot of the hovering target. Once there it continues moving synchronously with the pendulum along the y direction, trying to minimize the distance with the gate center in the y-z plane, while, at the same time, attempting to keep a steady position along the x and z directions that slowly converge toward the goal situated at (3, 0, 4).

Conclusions and Future Work

This thesis explored a novel state-of-the-art framework in which massive parallel simulation is adopted to learn an agile control policy. The latter is able to maneuver a drone in a highly dynamic environment: a fast-moving gate.

The policy is found with deep reinforcement learning PPO algorithm which trains a neural network fed with synthetic data coming from the randomized agile flight task setup. The dynamic gate setup is recreated and parallelized inside the Isaac Gym reinforcement learning simulator. Moreover, to interface the neural network control with the simulated quadrotor and, at the same time to maintain the simulation software full GPU pipeline, a low-level flight controller is implemented from scratch using the PyTorch framework. Different strategies have been tested to guide the agent toward the goal, starting from a reward that only stimulates a stable flight primitive to an increase of the informative task-related content in the next reward functions. As a matter of fact, while all the trained policies were able to learn a stabilizing flight controller, only the one trained with the most informative reward function was able to solve the agile benchmark in the 97.52% of cases and generalize across unseen configurations with 77.30% of success rate.

In conclusion, this thesis remarks the efficacy of the end-to-end approach in quadrotor control, which is able to follow a path that traverses a fast-moving gate without any planning and preception stages. The obtained results can serve as starting point for future work, for example testing the policy on even more challenging scenarios obtained by adding more gates and reducing the sensory information. On top of that, new and more sophisticated reward functions could be designed to avoid undesired oscillatory residual movements. Finally, other neural network architectures could be tested in the context of agile flight. For example, recursive neural networks, e.g. long short-term memory networks, aiming to better exploit the temporal nature of such dynamical tasks. In addition to this, even smaller and minimal neural networks, that can be quantized and loaded on low-power and low-cost embedded quadrotor platforms, could be analyzed in the direction of fully onboard sensing autonomous flight. Performing these tests will be valuable in pursuing the goal of sim-to-real transfer of the robust learned policy on the actual drone platform making it able to navigate an unconstrained and dynamic environment with only onboard sensing.

Bibliography

[1] Jon Verbeke and Joris De Schutter. "Experimental maneuverability and agility quantification for rotary unmanned aerial vehicle". In: *International Journal of Micro Air Vehicles* 10.1 (2018), pp. 3–11.

[2] Moses Bangura, Robert Mahony, et al. "Nonlinear dynamic modeling for high performance control of a quadrotor". In: (2012).

[3] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[4] Jonas Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602.7897 (2022), pp. 414–419.

[5] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[6] Ilge Akkaya et al. "Solving rubik's cube with a robot hand". In: *arXiv preprint arXiv:1910.07113* (2019).

[7] Nikita Rudin et al. "Learning to walk in minutes using massively parallel deep reinforcement learning". In: *Conference on Robot Learning*. PMLR. 2022, pp. 91–100.

[8] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2017, pp. 3389–3396.

[9] Viktor Makoviychuk et al. *Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning*. 2021. DOI: 10.48550/ARXIV.2108.10470. URL: https://arxiv.org/abs/2108.10470.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] Honglak Lee et al. "Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks". In: *Commun. ACM* 54.10 (2011), pp. 95–103. ISSN: 0001-0782. DOI: 10.1145/2001269.2001295. URL: https://doi.org/10.1145/2001269.2001295.

[12] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[13] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[15] Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning*. Addison-Wesley Professional, 2019.

[16] Richard S Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999.

[17] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[18] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[19] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077.

[20] Matthias Faessler et al. "Autonomous, vision-based flight and live dense 3D mapping with a quadrotor micro aerial vehicle". In: *Journal of Field Robotics* 33.4 (2016), pp. 431–450.

[21] Matthias Faessler, Davide Falanga, and Davide Scaramuzza. "Thrust Mixing, Saturation, and Body-Rate Control for Accurate Aggressive Quadrotor Flight". In: *IEEE Robotics and Automation Letters* 2.2 (2017), pp. 476–482. DOI: 10.1109/LRA.2016.2640362.

[22] Sergei Lupashin et al. "A platform for aerial robotics research and demonstration: The Flying Machine Arena". In: *Mechatronics* 24.1 (2014), pp. 41–54. ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2013.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0957415813002262.

[23] Matthias Faessler et al. “Automatic re-initialization and failure recovery for aggressive flight with a monocular vision-based quadrotor”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 1722–1729. DOI: 10.1109/ICRA.2015.7139420.

[24] Matthias Faessler et al. “Autonomous, Vision-based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle”. In: *Journal of Field Robotics* 33.4 (2016), pp. 431–450. DOI: https://doi.org/10.1002/rob.21581. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21581. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21581.

[25] Drew Hanover et al. *Autonomous Drone Racing: A Survey*. 2023. DOI: 10.48550/ARXIV.2301.01755. URL: https://arxiv.org/abs/2301.01755.

[26] Elia Kaufmann, Leonard Bauersfeld, and Davide Scaramuzza. “A Benchmark Comparison of Learned Control Policies for Agile Quadrotor Flight”. In: *2022 International Conference on Robotics and Automation (ICRA)*. 2022, pp. 10504–10510. DOI: 10.1109/ICRA46639.2022.9811564.

[27] Yuxi Li. “Deep reinforcement learning: An overview”. In: *arXiv preprint arXiv:1701.07274* (2017).

[28] Dingqi Zhang et al. *A Zero-Shot Adaptive Quadcopter Controller*. 2022. DOI: 10.48550/ARXIV.2209.09232. URL: https://arxiv.org/abs/2209.09232.

[29] Taewoo Kim, Minsu Jang, and Jaehong Kim. “A Survey on Simulation Environments for Reinforcement Learning”. In: *2021 18th International Conference on Ubiquitous Robots (UR)*. 2021, pp. 63–67. DOI: 10.1109/UR52253.2021.9494694.

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

[31] Taewoo Kim, Minsu Jang, and Jaehong Kim. “A Survey on Simulation Environments for Reinforcement Learning”. In: *2021 18th International Conference on Ubiquitous Robots (UR)*. IEEE. 2021, pp. 63–67.

[32] Yunlong Song et al. “Flightmare: A Flexible Quadrotor Simulator”. In: *Proceedings of the 2020 Conference on Robot Learning*. 2021, pp. 1147–1157.

[33] Elia Kaufmann et al. “Deep Drone Racing: Learning Agile Flight in Dynamic Environments”. In: (2018). DOI: 10.48550/ARXIV.1806.08548. URL: https://arxiv.org/abs/1806.08548.

[34] Andrew J Barry, Peter R Florence, and Russ Tedrake. “High-speed autonomous obstacle avoidance with pushbroom stereo”. In: *Journal of Field Robotics* 35.1 (2018), pp. 52–68.

[35] Boyu Zhou et al. “Robust and efficient quadrotor trajectory generation for fast autonomous flight”. In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 3529–3536.

[36] Mark W Mueller, Markus Hehn, and Raffaello D’Andrea. “A computationally efficient algorithm for state-to-state quadrocopter trajectory generation and feasibility verification”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2013, pp. 3480–3486.

[37] Antonio Loquercio et al. “DroNet: Learning to Fly by Driving”. In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 1088–1095. DOI: 10.1109/LRA.2018.2795643.

[38] Antonio Loquercio and Davide Scaramuzza. “Agile Autonomy: High-Speed Flight with On-Board Sensing and Computing”. In: ().

[39] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. “ORB-SLAM: a versatile and accurate monocular SLAM system”. In: *IEEE transactions on robotics* 31.5 (2015), pp. 1147–1163.

[40] Davide Falanga, Suseong Kim, and Davide Scaramuzza. “How fast is too fast? the role of perception latency in high-speed sense and avoid”. In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1884–1891.

[41] Fereshteh Sadeghi and Sergey Levine. “CAD^{2}RL: Real Single-Image Flight without a Single Real Image”. In: *CoRR* abs/1611.04201 (2016). arXiv: 1611.04201. URL: http://arxiv.org/abs/1611.04201.

[42] Antonio Loquercio et al. “Learning high-speed flight in the wild”. In: *Science Robotics* 6.59 (2021), eabg5810.

[43] Yunlong Song and Davide Scaramuzza. “Policy search for model predictive control with application to agile drone flight”. In: *IEEE Transactions on Robotics* 38.4 (2022), pp. 2114–2130.

[44] Jemin Hwangbo et al. “Control of a quadrotor with reinforcement learning”. In: *IEEE Robotics and Automation Letters* 2.4 (2017), pp. 2096–2103.

[45] Denys Makoviichuk and Viktor Makoviychuk. *rl-games: A High-performance Framework for Reinforcement Learning*. https://github.com/Denys88/rl_games. May 2021.

[46] Wojciech Giernacki et al. “Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering”. In: *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*. IEEE. 2017, pp. 37–42.

[47] Julian Förster. "System Identification of the Crazyflie 2.0 Nano Quadrocopter". en. Bachelor Thesis. Zurich: ETH Zurich, Aug. 2015. DOI: 10.3929/ethz-b-000214143.

[48] Wolfgang Hönig and Nora Ayanian. "Flying Multiple UAVs Using ROS". In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Springer International Publishing, 2017, pp. 83–118. ISBN: 978-3-319-54927-9. DOI: 10.1007/978-3-319-54927-9_3. URL: https://doi.org/10.1007/978-3-319-54927-9_3.

[49] *PX4 User Guide*. URL: https://docs.px4.io/main/en/config_mc/pid_tuning_guide_multicopter.html#rate-controller.

[50] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

Aknowledgements

Più passa il tempo, più la lista delle persone da ringraziare si allunga. Se così accade penso ci si possa ritenere fortunati.

Innanzitutto vorrei ringraziare il Prof. Andrea Acquaviva ed il Prof. Andrea Bartolini per i loro preziosi consigli e per aver creduto in me fin dall'inizio, permettendomi di sviluppare questo studio.

Un ringraziamento speciale va a Luca, che da più di un anno mi segue in tutte le mie attività e mi ha introdotto con passione al mondo dell'AI e della ricerca.

Grazie a tutto l'ECS Lab ed in particolare ad Alberto, Emanuele, Francesco e Martin per la loro disponibilità nell'aiutarmi e per i momenti condivisi.

L'esito di questo difficile percorso non sarebbe stato egualmente positivo senza gli amici a fianco, con i quali ho affrontato, tra gli altri, i momenti più faticosi: in particolare Alice, con cui ho ormai alle spalle undici anni di studi, Alberto, Nicola e Nicola (non è un refuso). Inoltre un ringraziamento di cuore va a tutti gli amici del gruppo di Bertinoro allargato ed agli amici di sempre con cui ho condiviso tanti momenti di felicità.

Grazie a Patrizia, Marco, Andrea, Arianna e tutta la famiglia "giù dal colle" per essere stati la mia seconda casa in questi ultimi anni.

Vorrei poi ringraziare tutta la mia famiglia, i miei zii, i miei cugini, i loro compagni e le mie nonne Gigliola e Marcella alle quali dedico i miei studi di più alto grado. Ringrazio i miei genitori Aurora ed Alberto, che hanno sempre fatto il massimo per spronarmi ad andare avanti e mi hanno dato conforto nei momenti più incerti, e mio fratello Leonardo, con il quale ho condiviso studi, pandemie, veleggiate e risate.

Per finire, grazie Bianca per essere sempre al mio fianco, in ogni momento di gioia o tristezza. Grazie per essere stata la mia compagna di vita e di studi universitari, che ora volgono al termine, e per lo splendindo futuro che ci aspetta assieme.