Titre: Title:	Deep Reinforcement Learning in Real-Time Environments
Auteur: Author:	Yann Bouteiller
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bouteiller, Y. (2021). Deep Reinforcement Learning in Real-Time Environments [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/6658/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6658/
Directeurs de recherche: Advisors:	Giovanni Beltrame
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Deep Reinforcement Learning in Real-Time Environments

YANN BOUTEILLER
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juin 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Deep Reinforcement Learning in Real-Time Environments

présenté par **Yann BOUTEILLER**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Daniel ALOISE, président
Giovanni BELTRAME, membre et directeur de recherche
Quentin CAPPART, membre

DEDICATION

For Lou.

ACKNOWLEDGEMENTS

The author thanks Giovanni Beltrame for his supervision and support at Polytechnique Montreal during this dual degree in Canada, and Valeria Borodin for her supervision and support at Ecole des Mines de Saint-Etienne in France. The author also thanks Maria Giesen for her remote administrative help, and more generally all the people in Polytechnique Montreal and Ecole des Mines de Saint-Etienne who contributed to make this dual-degree possible. Thanks in particular to Bernard Dhalluin, without whom none of this would have been possible. Thanks to everyone in the MIST Lab for making these two years an enjoyable experience. The author also thanks Pierre-Yves Lajoie, Yoshua Bengio and the anonymous reviewers of the paper for their helpful feedback, which helped greatly improve the readability of the article. The core of this thesis is a collaboration between the MIST Lab and the MILA, and has been accepted at the International Conference on Learning Representations (ICLR 2021). The author thanks ElementAI and Compute Canada for providing the consequent computational resources used to run the experiments featured in the second part of this thesis. Finally, the author especially thanks Simon Ramstedt, Jonathan Binas, Amine Bellahsen, Sanae Lotfi, Ricardo de Azambuja, Edouard Geze and Dong Wang, who at different points in time have all closely collaborated to the work presented in this thesis.

RÉSUMÉ

L'Apprentissage par Renforcement regroupe une famille d'algorithmes permettant de découvrir des contrôleurs performants. Ces algorithmes fonctionnent sur le principe d'essai-erreur, en maximisant la somme cumulative d'un signal dit de "récompense", conçu par le praticien. Dans des environnements simples et idéalisés vérifiant certaines propriétés, nombre de ces algorithmes sont mathématiquement garantis de découvrir un contrôleur optimal.

Les récents progrès de l'Apprentissage Profond ont permis d'étendre avec succès les algorithmes d'Apprentissage par Renforcement à des environnements beaucoup plus complexes. Néanmoins, ces succès demeurent le plus souvent cantonnés à des applications très cadrées telles que le jeu de go, d'échecs, ou encore les simulations informatiques. Dans ce mémoire, nous nous intéressons à étendre le domaine d'applicabilité de la discipline aux environnements réels, afin de faciliter par exemple l'apprentissage de contrôleurs pour la robotique.

Il est en effet difficile d'appliquer avec succès les algorithmes d'Apprentissage par Renforcement dans de tels environnements. En particulier, les environnements utilisés dans la littérature sont conçus pour être assimilables à des Processus Décisionnels Markoviens, sur lesquels se base toute la théorie de l'Apprentissage par Renforcement. Cependant, le monde réel est en général beaucoup trop complexe pour être naïvement assimilé à de tels objets idéaux. En particulier, il est vraisemblablement impossible d'observer l'intégralité de l'univers, le monde réel est non-stationnaire, et les évènements s'y déroulent de manière continue en temps-réel. Dans le cadre de ce mémoire, notre objectif est plus particulièrement d'étendre la théorie et la pratique de l'Apprentissage par Renforcement au domaine du temps-réel.

Les Processus Décisionnels Markoviens fonctionnent par nature au tour-par-tour. En effet, ils consistent schématiquement en un ensemble d'états fixes entre lesquels l'agent transitionne. Pour cette raison, les environnements utilisés par les théoriciens sont systématiquement mis en pause entre chaque transition, y compris ceux censés mimer des robots contrôlés en continu. Or, il est évidemment impossible de mettre le monde réel en pause, et il est donc fréquent que de nombreuses sources de délais apparaissent dans les applications pratiques.

Dans la première partie de ce mémoire (chapitre 3), nous présentons un travail préliminaire durant lequel nous avons développé, sous la forme d'un "jeu vidéo" de course aérienne ludique à deux joueurs, un logiciel destiné à la collecte de jeux de données d'Apprentissage par Renforcement Multi-Agents Hors-Ligne pour les drones autonomes. Le développement de ce logiciel met en lumière un certain nombre de problèmes liés à la présence inéluctable du temps-réel à de nombreux niveaux de l'application. Ces différents problèmes illustrent

la motivation sous-jascente de notre contribution principale, présentée dans les chapitres suivants.

La seconde partie de ce mémoire (chapitre 4) expose notre principale contribution théorique. Nous décortiquons mathématiquement l'anatomie des environnements réalistes dans lesquels apparaissent des délais aléatoires, par exemple au niveau du temps d'inférence, de transmission et d'activation des actions, ou encore au niveau du temps de capture et de transmission des observations. Nous démontrons qu'il est possible, dans de tels environnements, de transformer des fragments de trajectoires collectés par une politique obsolète ("off-policy") en fragments de trajectoire compatibles avec la politique actuelle ("on-policy"). Cette transformation est effectuée par le biais du ré-échantillonnage partiel des fragments obsolètes, sous certaines conditions portant sur les délais mesurés dans lesdits fragments. Les fragments ainsi générés nous permettent d'estimer la fonction de valeur par propagation à pas multiples sans biais ni explosion de variance. Nous appliquons ce principe pour créer Delay-Correcting Actor-Critic (DCAC), un algorithme basé sur Soft Actor-Critic avec de bien meilleures performances en présence de délais. Ce gain de performance est montré mathématiquement et par le biais d'expériences réalisées dans les environnements MuJoCo classiques, que nous avons modifiés afin d'y introduire des délais aléatoires. Ce chapitre central a été accepté en tant qu'article de conférence à International Conference on Learning Representations (ICLR 2021).

La troisième et dernière partie de ce mémoire (chapitre 5) est consacrée à l'application pratique et présente notre travail en cours et à venir. D'une part, nous introduisons un utilitaire python, publié sur PyPI, que nous avons développé afin de faciliter la création d'environnements Gym dans le monde réel. À l'aide de cet utilitaire, nous développons des environnements pour des applications variées, telles que la robotique et les jeux vidéos de course en temps-réel. D'autre part, nous présentons une infrastructure logicielle que nous avons développée afin de permettre l'entraînement à distance de nos robots et autres agents exécutés localement. Il nous est ainsi possible d'utiliser des serveurs de calcul haute-performance afin d'entraîner nos agents collectant des expérience en parallèle dans le monde réel. Cette infrastructure, largement inspirée de la bibliothèque RLlib développée par l'Université de Berkeley, nous offre un contrôle beaucoup plus étendu et nous permet en particulier d'implémenter des pipelines de traitement et de transmission des données spécifiques à chaque tâche. Nous présentons en particulier des résultats prometteurs dans le domaine de la conduite sportive autonome.

ABSTRACT

Whereas all environments commonly used in the Reinforcement Learning (RL) literature are paused between transitions, it is simply not possible to pause the real world. Thus, action and observation delays commonly occur in many practical RL applications.

In our central contribution, we study the anatomy of randomly delayed environments, and show that partially resampling trajectory fragments in hindsight allows for unbiased and low-variance off-policy multi-step value estimation. We apply this principle to derive Delay-Correcting Actor-Critic (DCAC), an algorithm based on Soft Actor-Critic with significantly better performance in environments with delays. This is shown theoretically and also demonstrated practically on a delay-augmented version of the MuJoCo continuous control benchmark. This contribution, presented in central chapter of this thesis, has been accepted as a conference paper at the International Conference on Learning Representations (ICLR 2021).

In our second and more practical contribution, we develop RL environments in real-time applications. We provide a python helper, Real-Time Gym, that enables implementing delayed RL environments in the real world with minimal effort. We demonstrate this helper on applications such as robotics and real-time video-games. We further introduce a framework that we developed in order to train our real systems distantly on a High Performance Computing server, and present promising results on autonomous car racing tasks.

The aforementioned contributions are motivated by early work that we present in the preliminary chapter of this thesis. In this side-contribution, we develop a two-player drone-racing 'video-game' on top of a photo-realistic simulator, in order to collect drone-racing datasets for Offline Reinforcement Learning applications. In particular, this early work highlights several difficulties related to Real-Time Reinforcement Learning that we address in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

MDP	Markov Decision Process
CDMDP	Constantly Delayed Markov Decision Process
RDMDP	Randomly Delayed Markov Decision Process
POMDP	Partially Observable Markov Decision Process
MRP	Markov Reward Process
RL	Reinforcement Learning
RTRL	Real-Time Reinforcement Learning
MARL	Multi-Agent Reinforcement Learning
SAC	Soft Actor-Critic
RTAC	Real-Time Actor-Critic
DCAC	Delay-Correcting Actor-Critic
GPU	Graphics Processing Unit
CPU	Central Processing Unit
AI	Artificial Intelligence
ML	Machine Learning
NN	(artificial) Neural Network
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
SOTA	State-Of-The-Art
ADRL	Airsim Drone Racing Lab
UML	Unified Modeling Language

CHAPTER 1 INTRODUCTION

Reinforcement Learning (RL) is a field of Machine Learning (ML) inspired from the concept of Reinforcement, which comes from behavioral psychology. Reinforcement partially explains living organisms' behavior as the result of an history of rewarding and punishing stimuli. Simply put, in a given situation, the organism would try certain actions and experience certain outcomes. When later facing similar situations, the organism would become more likely to retry the actions that yielded the best outcomes in the past, and less likely to retry the actions that yielded the worst outcomes.

Throughout this thesis, we assume that the reader is familiar with the basics of Reinforcement Learning, and to a lesser extent with the basics of Deep learning. For an introduction to the field of Reinforcement Learning and definitions of most of the vocabulary used in this work, we refer the reader to the second edition of "Reinforcement Learning: an introduction" by R. Sutton and A. Barto [1]. For an introduction to the field of Deep Learning, we recommend "Deep Learning" by I. Goodfellow, Y. Bengio and A. Courville [2].

RL has attracted a lot of attention over the past few years, in particular because the recent advances in parallel computing and Deep Learning enabled spectacular successes of RL-based approaches. Probably the most renowned of these successes is still the DeepMind team's performance at playing go [3] after their work led to a Monte-Carlo Tree Search algorithm outperforming Lee Sedol, known as the best human go player in the world.

As a side note, the reputation of this story highlights an observation that we made during this work: when it comes to RL, people who are not practitioners in the field often tend to exhibit a very strong bias around the human-level performance, and start abruptly showing interest when an agent surpasses this threshold. However, arguably the most impressive success here was to use self-play and a non-heuristic reward. Indeed, AlphaGo learnt the game from scratch by playing against itself, without any prior human knowledge. More precisely, the only prior human knowledge was contained in an extremely simple and sparse reward signal: +1 when the game was won, and -1 when it was lost.

Regarding more general applications, several observations should be made about this example and will serve as a starting point for this thesis. First, thanks to the recent advances in Deep Learning, deep RL approaches are able to learn complex strategies in very complex state-spaces, even with naturally sparse rewards [4]. Indeed, the go state space is a 361-dimensional vector that can take approximately $2 \cdot 10^{170}$ values each turn, for a typical game depth of 150 moves, which would have completely prohibited an exhaustive search. In particular, this

illustrates the potential of deep RL in real-world scenarios where the state-space can be even much larger. This is the case when e.g. using images as observations, which was already done in 2015 with human-level performance in simple video games by Mnih et al. [5]. Second, through e.g. self-play, deep RL approaches can adapt to multi-agent scenarios with no prior knowledge about the other agents' policies. In fact, when a RL agent trains using self-play, it explores a part of the policy-space that eventually includes the policies of other agents such as human players. Of course, instead of training against itself, a RL agent can train against or with other agents, possibly also learning, in both adversarial and cooperative settings. This makes RL promising for a wide range of real-world scenarios where several learning agents interact, being either machines or living beings. Third and more importantly for the matter of this thesis, the setting in which AlphaGo takes place can be considered turn-based. This means that each game turn can be represented as a fixed state, from which an action is computed that leads to another fixed state, and so on.

In fact, this turn-based scenario is not limited to AlphaGo but is a characteristic of most RL approaches. This is because they rely on Markov Decision Processes (MDPs) as their underlying mathematical framework, and therefore describe the world as a series of fixed states linked by decisional transitions. From one given state, an action is selected and applied, yielding a new state, and so on.

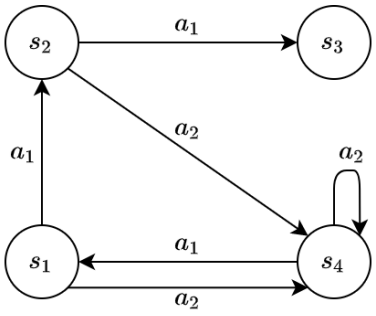

Figure 1.1 A simple Markov Decision Process

The MDP presented in Figure 1.1 is a very small, deterministic example with a discrete state space of 4 states and a discrete action space of 2 actions. If the agent is in e.g. state s_4, it needs to choose between actions a_1 and a_2. After applying either of these actions, it will find out that selecting a_1 (respectively a_2) leads to state s_1 (respectively s_4), and can repeat this experience over and over to build its internal understanding of the world. Of course, the main assumption is that the world can be described by this type of dynamics. This assumption is compressed into the Markov assumption, essentially saying that the state

observed by the agent at any point in time contains enough information to predict future states with no additional knowledge from the past. This is often a reasonable assumption to make as long as it is approximate in practice, because except in very simple turn-based games or simulations, the actual state of the universe is impossible to observe, and evolves continuously. This is especially true in real-world robotics and complex, real-time simulations or video games. In such situations, the world is practically described as a Partially Observable MDP [6], which essentially makes the same underlying assumptions about its state and dynamics but acknowledges that only a portion of the state is observed by the agent.

The reason why it often makes sense to consider the world's dynamics as a series of fixed states and transitions is that, in practice, many algorithms work in this turn-based fashion. In other words, they produce a single output, which in our case will be an action, from a single input, which in our case will be an observation of the world. There exists work toward continuous-time RL trying to adopt a continuous-time analysis instead [7]. Yet, since the vast majority of current RL algorithms are turn-based [1], we will focus on the usual modeling of the world using discrete time-steps in this thesis.

In its mathematical definition, an MDP also contains an intrinsic judgement about which states or transitions are good or bad, called a reward signal. For a given MDP and a given fixed strategy, each state can therefore be associated to an expected future reward. This augmentation is called a Markov Reward Process (MRP). Implementation-wise, an MRP is represented by an environment with which an agent interacts.

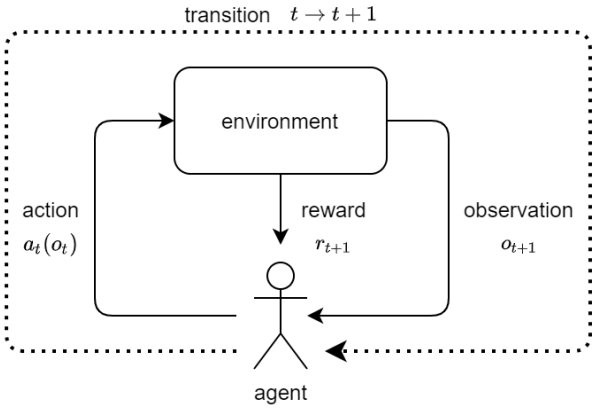

Figure 1.2 Usual diagram of a Markov Reward Process

In the MRP depicted by Figure 1.2, the agent observes the environment at a given time-step, computes an action from this observation, applies this action in the environment, and the environment is stepped to the next time-step, which yields a reward and a new observation.

Notice that the word 'time-step', widely used in RL literature [1], is misleading when we are interested in real-time control. Indeed, the concept of time-step does not actually refer to any concept of elapsed time. Instead, it refers to a transition between two fixed states, which should not naively be interpreted as a fixed duration between two instantaneous points in time. Nevertheless, this is exactly what has been done by most RL theorists for a long time [8], as they were more interested in solving general mathematical problems framed as MDPs than in questioning the real-world relevance of MDPs themselves.

The reality is, MDPs are in fact well-suited to describe purely turn-based Markov chains of decisions, such as single-player board games where time has no actual impact on whatever happens in the considered universe (e.g. peg solitaire).

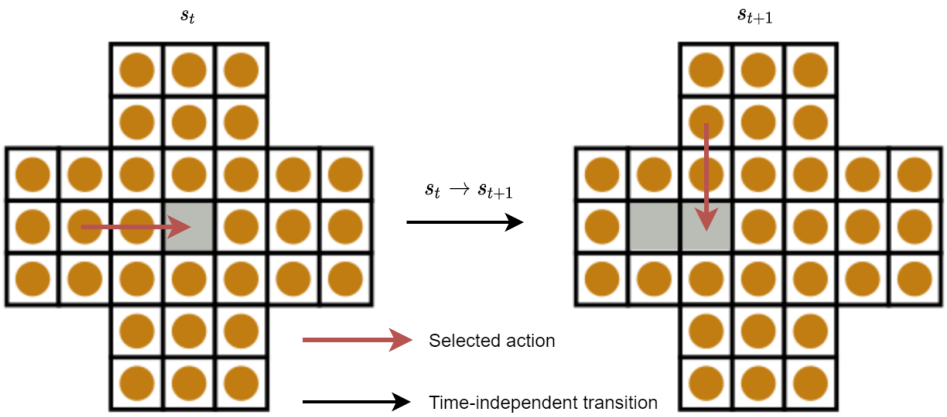

Figure 1.3 Example of a purely time-independent Markov Decision Process.

In the example of Figure 1.3, the amount of time taken to select an action (which marble to move) and the time it takes to actuate this action (move the marble) has no impact on the state of interest (the marbles' positions at each turn).

On the other hand, using MDPs to describe two-players board games such as go is already an approximation. Indeed, a human opponent, for instance, thinks in real-time and their internal state is continuously evolving. In particular, the longer the agent takes to play, the longer the other player will be able to think about their next move, and other similar subtle dynamics such as their level of stress will be impacted. When naively modeling the problem as an MDP with no further precaution, these subtle dynamics are de facto ignored, and actually violate the Markov assumption. Of course, in the case of go, such dynamics have so little impact compared to the mechanics of the actual game that they can safely be considered negligible.

However, this issue has much more impact when RL algorithms are applied to real-time

settings such as real-world continuous control, real-time simulators, or real-time video games [9]. For a long time, RL theorists have used fakely turn-based environments to simulate continuous control tasks, which are currently part of the very popular OpenAI Gym RL benchmark suite [8]. Notably, one can think of historical tasks such as Pendulum, where an agent tries to balance a solid pole against gravity around a single axis, or of the current MuJoCo continuous benchmark suite [10], which was introduced more recently as the aforementioned historical tasks became too easy for the State-Of-The-Art (SOTA) deep RL algorithms.

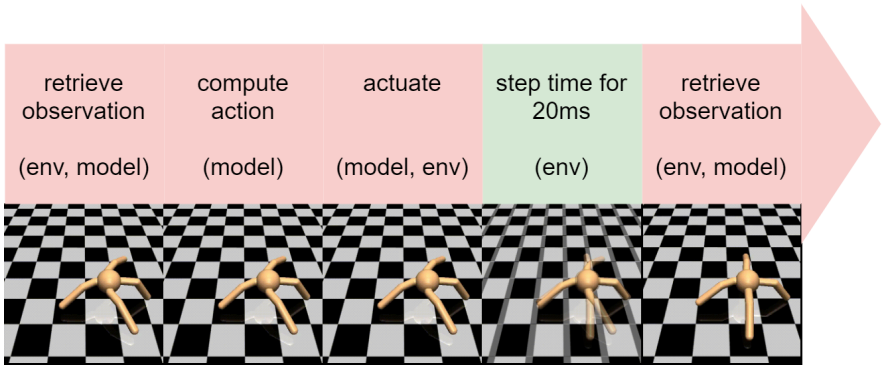

Figure 1.4 Typical operation of the Gym continuous control benchmark tasks.

Figure 1.4 depicts one of the classic MuJoCo continuous control benchmark tasks. It typically illustrates the approach that RL theorists have taken for a long time when dealing with robotic control. Namely, adapting the task to the theory rather than the other way around. Indeed, as we have previously highlighted, MDPs are essentially fixed states linked by transitions. A consequence is that, when using an MDP, one mathematically considers that the universe doesn't evolve between transitions. In other words, everything that happens between transitions is instantaneous, which practically means that the environment is paused. This is why the Gym benchmarks are turn-based, even for continuous control tasks close to robotic applications such as the MuJoCo tasks. Forcing this time-stepping scheme between paused states comes with convenient advantages. Not only it allows to practically consider observation capture and action inference as instantaneous, but it enables other computations while the environment is paused, e.g. training a neural network.

The real world, however, cannot be paused. While some articles in robotic RL treat time delays as negligible [11] or design their systems to be effectively paused between time-steps [12], this is of course not suitable in general, as practitioners often need their systems to act and react as fast as possible. In this thesis, we design advanced Gym environments for real-time applications, and RL algorithms that deal with this common scenario.

The structure of the thesis is as follows:

In Chapter 2, we provide an overview of the existing literature that is relevant to our work. The specific sub-field of Reinforcement learning in the presence of explicit delays is still rather under-explored. This is because it is mainly relevant to practical real-world applications such as autonomous driving, in which RL algorithms are only starting to attract serious attention.

In Chapter 3, we briefly introduce preliminary work in which we develop a Gym environment featuring two drones, racing in a photo-realistic simulation. We use this environment to build a two-player drone-racing 'video game' that records players' input, full state-transitions, and a reward signal for each player. This software is effectively a dataset collector for Multi-Agent Real-Time RL, suitable for Offline RL, Inverse RL and Imitation Learning altogether. We highlight time-related issues that naturally arise during the development of this software. These issues illustrate the underlying motivation of the contribution presented in the next chapters. Our code for this software is open-sourced at the following URL: https://github.com/yannbouteiller/gym-airsimdroneracinglab.

In Chapter 4, we present our main theoretical contribution, published at ICLR 2021 [13]. Being particularly interested in the real-world relevance of our approach, we study the anatomy of environments featuring realistic, random delays. We prove that, in such environments, it is possible to transform off-policy trajectory fragments into actual on-policy fragments, which effectively enables unbiased and low-variance multi-step value estimation. This is done by using the current policy to recursively re-sample the history of actions that is part of the full Markov state in delayed environments. We apply this principle to derive Delay-Correcting Actor-Critic, an algorithm based on Soft Actor-Critic with significantly better performance in the presence of random delays. Our code for this chapter is open-sourced at: https://github.com/rmst/rlrd.

Finally, in Chapter 5, we introduce our current and future work toward real-world RL. We present Real-Time Gym, a small python helper that enables implementing delayed Gym environments in the real world with minimal effort. Real-Time Gym is open-sourced at: https://github.com/yannbouteiller/rtgym. We demonstrate this helper in practical applications such as robotics and autonomous racing in real-time video-games with no insider access. We further develop our own framework in order to easily build ad-hoc training pipelines. This allows us to collect RL samples locally on real-world agents, while training their policies distantly on a High Performance Computing system. In particular we show promising results in our autonomous racing tasks.

CHAPTER 2 LITERATURE REVIEW

2.1 Algorithms and models

The whole history of RL and of Deep Learning are somehow relevant to this thesis, and the most relevant knowledge about both fields can be found in "Reinforcement Learning: an introduction" by R. Sutton and A. Barto [1] and "Deep Learning" by I. Goodfellow, Y. Bengio and A. Courville [2]. We work within the very well-established framework of Markov Decision Processes [14], which is fundamentally the base of most current RL algorithms [1, 15]. The most important class of algorithms whose understanding is required to comprehend this thesis is the class of Temporal Difference Learning approaches, such as TD(lambda) [16]. More specifically, we propose a Temporal Difference approach for environments with real-time delays, from which we derive Delay-Correcting Actor-Critic (DCAC), an improved version of Soft Actor-Critic (SAC) [17, 18] in such environments. The reason why we choose to improve SAC in particular is that it is well-suited for real-world applications.

2.2 Reinforcement Learning in the real-world

Back in 2013, Kober et al. published a survey highlighting how RL could be used in some robotic applications with a few notable achievements at the time [19]. The recent advances in the field enabled more spectacular successes in difficult tasks such as learnt legged locomotion, in particular by Haarnoja et al. [20] using SAC. There are three main reasons why SAC is well-suited for real-world applications such as robotics. First, it is off-policy. In other words, it enables experience replay, which is known to greatly improve sample-efficiency [21]. This is important in robotics, where sample collection is costly [19]. Second, SAC deals with continuous policies, thus enabling continuous actions, which are common in robot control [19]. Third, SAC trains a stochastic policy and maximizes its entropy for exploration purpose. This has been shown to produce policies that are robust, for example to the stochasticity of real-world terrains in legged locomotion [20]. Apart from legged locomotion, RL strategies recently achieved notable successes in various real-world tasks such as complex robotic arm control [12, 22] or learning of low-level flight controllers [23, 24]. These real-world policies are often trained in simulation. Indeed, simulation provides a safe and convenient environment where robots can try dangerous actions without damaging themselves, and it can be sped up to collect samples at a fast rate.

2.3 Simulators and environments

In this thesis, we are specifically interested in the time-related aspects of real-world RL. These aspects are difficult in simulation, and they are usually approached rather naively when not avoided altogether. In particular, the MuJoCo simulator [10] is typically used by RL theorists as part of the Gym [8] benchmark to demonstrate algorithms for continuous control. This simulator being paused between time-steps, it effectively enforces instantaneous delays in practice. Since MuJoCo is popular in the RL community, we modify this benchmark thanks to a Gym wrapper that artificially introduces user-defined delays in any turn-based Gym environment. This allows us to provide results that compare to the literature in Chapter 4.

In Chapter 3, we choose the Microsoft AirSim Drone Racing Lab (ADRL) simulator [25] in order to build a multi-agent RL dataset collector for quadrotor racing. AirSim is a photo-realistic simulator for cars and quadrotors. It compares to CARLA [26] and, in our experience, both simulators share the same weaknesses in terms of computational heaviness, hardware-dependency and poor control over time. We chose ADRL because it readily offers a quadrotor racing simulation, which is an interesting and rich multi-agent task, and because it emulates real-world flight controllers.

In Chapter 5, we demonstrate our real-time framework in image-based autonomous car racing tasks. We use the TrackMania 2020 [27] video game as a virtual universe. Fusch et al. [28] conducted related work with a simulator built with an insider version of the Gran Turismo video game. This simulator provided many ground truth inputs to their model, was presumably paused between transitions, and allowed the authors to gather samples from eighty cars in parallel. As opposed to this work, we do not have insider access to the TrackMania video game. Instead, we use the actual game in real-time like humans do, and we collect samples from one single car. We are in this sense training in the real-world setting, while still using a virtual game for safety. The reason why we chose TrackMania specifically is that it can be used for free, provides a replay editor that enables shooting qualitative results, and can be used with a community framework that allows us to retrieve positions in order to build our reward function.

2.4 Multi Agent Reinforcement Learning

The environment presented in Chapter 3 features two learning agents in the same quadrotor simulation. While this setting is difficult due to the real-time setting, it is even more difficult due to the presence of several learning agents. Indeed, from a single-agent point of view, the other agent is part of the environment. However, since this other agent is learning over time,

such environment is not stationary, which violates the Markov assumption over training. In practice, this scenario is very relevant to real-world control, where the environment is often non-stationary (e.g. self-driving cars learning in parallel). This poses the following dilemma: a non-stationary environment calls for the use of on-policy algorithms such as Proximal Policy Optimization [29], which are naturally well-suited for this setting because they can adapt to the slow changes of the environment in an online fashion. However, on-policy algorithms are generally not suited for real-world scenarios because of their sample-inefficiency, which calls for off-policy algorithms. For this matter, Lowe et al. [30] introduced the idea of a centralized critic training decentralized actors. The Multi Agent Reinforcement Learning setting has been further and intensively explored by Jakob Foerster's team [31].

2.5 Offline approaches

Although this is not the core subject of this thesis, it is useful from a practical point of view to gather some basic knowledge about methods that enable learning well-performing policies from datasets collected offline. Indeed, in Chapter 3, we present a software that we developed in order to collect multiagent drone-racing datasets, including a reward signal. Additionally, in Chapter 5, we introduce a framework that we developed in order to demonstrate the approach presented in the central part of this thesis on any real-world applications. In particular, this framework enables the collection of similar datasets. Once collected, there are 3 possible directions for using these datasets.

The first direction is to ignore the reward signal. Indeed, although in our case we provide a reward function, it is often the case that such function is not available. In this situation, one often uses Imitation Learning approaches, which can be separated in two categories: Behavioral Cloning and Inverse Reinforcement Learning. Behavioral Cloning tries to 'copy' an expert's policy. In other words, it is a supervised learning approach that predicts the expert's actions given the observations which were recorded while the expert was controlling the device. Inverse Reinforcement Learning is conceptually more advanced: it tries to infer the reward function that the expert was trying to optimize, without actual access to the reward signal. A comprehensive survey of the field is provided up to 2019 by Torabi et al. [32], and this is still an active area of research [33–36]. In particular, Imitation learning is often used in robotic settings, such as autonomous driving [37].

The second direction is to use the reward signal in order to learn a policy that performs better than Imitation Learning policies. This method, known as Offline Reinforcement learning (also often called Batch Reinforcement Learning), is close to Inverse Reinforcement Learning. Yet it is more efficient and straightforward, because the reward signal is available and thus doesn't

need to be inferred. The main difficulty in this setting is that, since the training algorithm has only access to a finite amount of data and cannot further explore the environment, it tends to erroneously extrapolate its available information to unexplored states [38]. In particular, this results in an overestimation of the value of unexplored states, which leads pure RL off-policy algorithms to perform poorly. Current approaches dealing with this issue regularize the RL policy by using a trade-off factor between the RL policy and the Behavioral Cloning policy [39], which exhibits good results in practice.

The third direction is to use either one or a combination of the aforementioned approaches in order to initialize the model or the algorithm before allowing it to explore the environment. This is particularly relevant to the algorithm that we present in the central part of this thesis, because this algorithm is based on Soft Actor-Critic (SAC, which uses a replay memory and is therefore compatible with the Offline RL setting), and is designed for real-world scenarios. The end-user is likely to be interested in achieving the best possible final performance within the smallest possible amount of training time. In this regard, it is interesting to either initialize the replay memory with expert samples instead of random samples, initialize the actor network with a Behavioral Cloning policy, initialize both the actor and the critic with an Offline RL algorithm, or combine some of these tricks.

2.6 Reinforcement Learning with delays

In this thesis, we design efficient deep RL algorithms for real-time environments. Such environments are typically encountered when developing continuous controllers for real robots, or for real-time simulations. Their main characteristic is that time cannot naively be considered paused between transitions, and therefore time-delays need to be considered as part of the the analysis. We trace our line of research back to 2003, when Katsikopoulos et al. [9] provided the first discussion about Delayed Markov Decision Processes. Their work showed that, in the presence of delays and under certain assumptions, it is sufficient to consider a buffer of past actions long enough to cover the duration of the delays as part of the state-space of an MDP, in order to ensure its Markov property. This finding is nowadays the solution commonly adopted by non-naive approaches in settings where delays should not be ignored [18, 28, 40]. In particular, Katsikopoulos et al. discussed three types of delays: action delays, observation delays, and asynchronous rewards. Asynchronous rewards are trivial in the setting of this thesis because they can simply be re-synchronized off-policy, so we are not concerned with these. However, we use action and observation delays extensively.

In 2008, Walsh et al. [41] re-introduced the notion of "Constantly Delayed Markov Decision Process". While recent advances in deep learning enable implementations of what the authors

call “the augmented approach”, which is the approach described by Katsikopoulos et al., this was considered intractable at the time because the size of the action buffer grows with the considered delay length. Instead, they studied the case where observations are retrieved with a constant delay while actions are generated instantaneously, and developed a model-based algorithm to predict the current state of the environment. Considering that inference is instantaneous is naive, but this is not a problem in practice when delays are constant because, in this specific situation, observation and action delays are equivalent. However, this approach suffers from the uncertainty that is inherent to the explicit state-predictive process. Most of the following work in RL with constant delays derives from Walsh et al., and inherits this drawback. Similarly in 2010, Schuitema et al. [42] developed “memory-less” approaches based on SARSA and vanilla Q-learning, taking advantage of prior knowledge about the duration of a constant control delay.

In 2013, Hester et al. [43] released an interesting article that is quite different from state-predictive approaches. They adopted the action-buffer-augmented approach from Katsikopoulos et al. to handle random delays, and they relied on a decision-tree algorithm to perform credit assignment implicitly. By comparison, the approach presented in this thesis is a Deep Learning approach that relies on delay measurements to perform credit assignment explicitly, which allows us to propagate delayed information with no uncertainty.

More recently, Firoiu et al. (2018) [44] introduced constant action delays to a video game, in order to train agents whose reaction time compares to humans. Similar to previous state-predictive approaches, the authors used a state-predictive model, but were the first to do so with a Deep Learning approximator, based on a Recurrent Neural Network (RNN) architecture. Similarly, in 2021, Derman et al. proposed Delayed-Q [45], an algorithm for constantly delayed environments based on a recurrent state-predictive architecture.

In 2019, Ramstedt et al. [46] formalized the framework of Real-Time Reinforcement Learning (RTRL) that we generalize to all forms of real-time delays in this thesis. Initially designed to cope with the fact that inference is not instantaneous in real-world control, the RTRL setting is equivalent to a constantly delayed MDP with an action delay of one time-step and no observation delay. We instead study stochastic, multi-step action and observation delays.

In 2020, [47] adopted an alternative approach by considering the influence of the action selection time when action selection is performed within the duration of a larger time-step. However their framework only allows delays smaller than one time-step, whereas large time steps are not compatible with high-frequency control. In this thesis, we instead present an analysis that is agnostic to the duration of the time-step, and which still holds when the time-step is made arbitrarily small.

CHAPTER 3 PRELIMINARY: REAL-TIME QUADROTOR RACING

This preliminary chapter briefly presents a multi-agent drone racing Gym environment built on top of Microsoft AirSim Drone Racing Lab (ADRL) [25]. This project is advanced enough to serve as a tool for Multi Agent Reinforcement Learning (MARL) and Robotics research, in particular in the Offline Reinforcement Learning setting. This environment currently works in a non-real-time fashion. This means that, like other usual RL benchmarks for continuous control, it is paused between time-steps and uses arguable hacks to do so. This poses practical issues that we will further describe in this chapter. These issues will then provide a motivation for the next chapters of this thesis, and be alleviated as a conclusion.

3.1 Gym environments

Most of the current work in RL research uses the very popular OpenAI Gym [8] python interface, and the vast majority of RL algorithms are implemented to work along this framework. There are two main reasons for this success. The first reason is that OpenAI Gym comes with a number of readily implemented environments, serving as benchmarks. These benchmarks are widely adopted by the community of RL researchers to evaluate and compare the performance of their algorithms. In particular, Gym natively contains historical tasks with either discrete or continuous action and observation spaces, Atari video games with discrete inputs as action spaces and images as observation spaces, and some continuous control tasks using the MuJoCo simulator [10] with continuous action and observation spaces. The second reason for this success is that the Gym python interface itself is very simple and, more importantly, provides a unified standard across RL implementations. In other words, as long as a RL algorithm is fully compatible with the Gym python interface, this algorithm will work on all Gym environments without any required change.

The principle of this interface is depicted in Figure 3.1. The Gym interface consists of two main python abstract methods, `step` and `reset`, and a few optional abstract methods such as `render`. Additionally, when implemented, a Gym class must implement two required attributes: `action_space` and `observation_space`. The RL algorithm, depicted as 'Model' in Figure 3.1, first retrieves the `action_space` and `observation_space` in order to know the shape of the observations that it must expect from the environment (e.g. images, lists, dictionaries...), and the shape of the actions that the environment expects (usually a numpy array). The RL algorithm then calls the `reset` method, which outputs an initial observation o_0. In the episodic setting, `reset` is called at the beginning of each episode. The RL

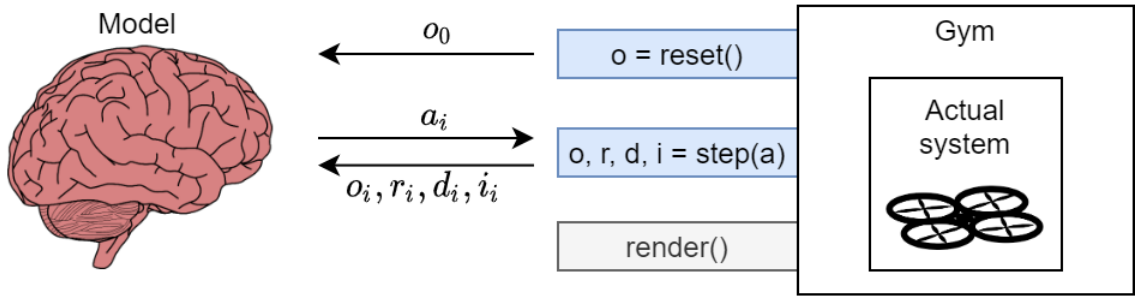

Figure 3.1 Principle of the OpenAI Gym unified interface

algorithm then computes an action a_1 from the initial observation o_0, and passes it as an argument to the `step` method. In response, the `step` method steps the environment for 1 time-step, and outputs a new observation o_1, a transition reward r_1, a Boolean signal d_1 that tells whether the task is 'done' in the episodic setting, and an optional i_1 dictionary usually containing debugging information. Action inference and calls to `step` then happen alternatively until the d_i signal is 'True', in which case the `reset` method must be called before calling `step` again.

As previously pointed out, this stepping scheme assumes that the environment is not evolving in real-time. Instead, the environment is assumed to be paused between transitions and, when time is part of the environment, it is assumed to be simulated within the `step` method.

3.2 gym-airsimdroneracinglab

AirSim Drone Racing Lab (ADRL) [25] is a software built on top of the Microsoft AirSim simulator [48], which relies on the Unreal Engine to provide photo-realistic simulations of quadrotors or cars, as illustrated in Figure 3.2. ADRL provides a few readily available maps in which the user can place gates that serve as a track for drone racing, as illustrated in Figure 3.3. The simulation lives on a local server and is controlled externally through an API.

In this project, we built a Gym environment for quadrotor racing internally using ADRL. This environment is a multi-agent environment called `gym-airsimdroneracinglab` . It works under both Linux and Windows.

`gym-airsimdroneracinglab` encapsulates ADRL in an OpenAI Gym interface for RL applications. It implements a reward function, a `reset` scheme and a `step` scheme. While ADRL currently only allows the user to control a single drone, we slightly modified its python API to control several drones in parallel. In `gym-airsimdroneracinglab` , the RL algorithm controls either one or two quadrotors racing against each other. In particular, this enables

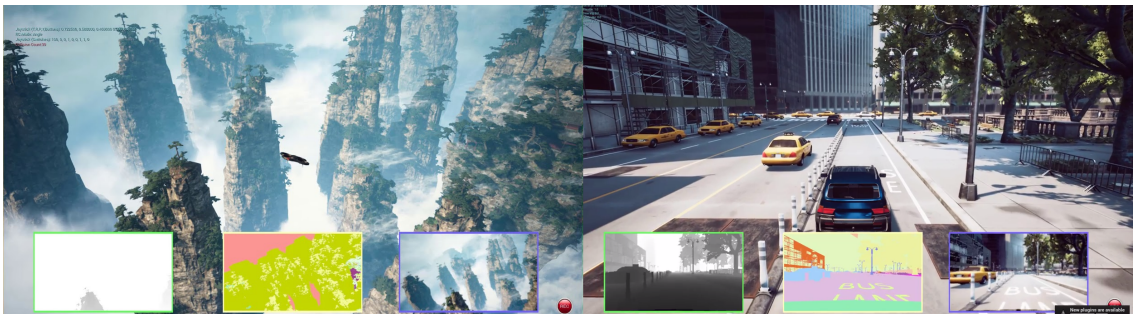

Figure 3.2 AirSim (credit: Microsoft Research)

Figure 3.3 AirSim Drone Racing Lab (credit: Microsoft Research)

learning a racing policy under the self-play paradigm, i.e. training the agent against itself. Note that the OpenAI Gym interface is not natively supporting MARL settings (such as this two-agents setting), and therefore does not provide a unified interface for multiple agents. In order to alleviate this issue, we chose to use an additional dimension in the actions that `step` expects, and in the observations that `reset` and `step` return. The index along this additional dimension identifies the agent. Note that another convention has been adopted by the popular RLlib library [49], which instead uses nested hash tables where each agent is represented by a key. For convenience, we also provide a wrapper that converts our Gym environment into a RLlib Gym environment, so it can be used directly with the multi-agent RL algorithms readily implemented in RLlib.

We also modified ADRL to enable several instances of the simulator in parallel on the same machine. If enough computational resources are available, this enables running multiple instances of `gym-airsimdroneracinglab` in parallel on a single machine, which multiplies the data collection rate by the number of parallel environments. This option is mainly useful when several dedicated GPUs are available on the same machine, because the AirSim simulator is very heavyweight.

In addition, we modified ADRL to increase (or decrease) the speed of the simulation. This again multiplies the data collection rate by the specified speed. However, there is a trade-off here which is inherent to how AirSim works. Indeed, the simulation is computed on-the-fly

and its quality is hardware-dependent in AirSim. In particular, this means that, when the simulation speed is increased, the simulation quality deteriorates. Conversely, when the simulation speed is decreased, the simulation quality improves.

On top of these data-efficiency features, we implemented a feature for generalization tasks. Namely, while most available maps are fixed, we augmented the 'Soccer_Field_Medium' map with a track randomization algorithm. When `reset` is called on this map, a new track will be randomly generated. This feature is particularly useful to train generalizable policies, whereas policies trained on other tracks are likely to perform well on these other tracks only.

For further details, we refer the interested reader to our code.

3.3 Dataset-Collecting Game

We built a two player 'video game' on top of `gym-airsimdroneracinglab`, also working under Linux and Windows. This software records continuous inputs from connected gamepads (or from the keyboard when no gamepads are available), steps the Gym environment with these inputs, retrieves the output of the transition (set by default to contain camera images and data in several coordinate systems for each drone, along with their transition rewards), displays the camera images of both drones and stores each formed transition in a dataset.

A screenshot of this software is displayed in Figure 3.4. By default, the camera images are of dimension 320×240 with RGB channels, the duration of a transition is 0.1s, and the maximum duration of an episode is 100.0s (the episode is also ended when both competitors have completed the track). While this application can be considered a fairly enjoyable racing game, what it really does is record a dataset containing enough information for image-based as well as simpler Offline RL training. This includes one reward signal and one 'done' signal per drone (the reward function can be customized though a python dictionary), one front camera image per drone (it is possible to change this for a depth map or a segmentation map), as well as linear and angular positions, velocities and accelerations in drone-centric, world-centric and NED coordinates, etc. Once the desired dataset has been collected, the user simply selects the information that is relevant for their application and discards the rest.

3.4 Real-Time caveats

As previously pointed out, `gym-airsimdroneracinglab` (and therefore the Dataset-Collecting Game) works in a non-real-time fashion and is paused between transitions.

This is achieved by using the following stepping architecture: When `step` is called, the

Figure 3.4 Two player game collecting offline RL datasets for quadrotor racing

actions passed as argument to this method (one for each drone) are sent to the simulator. The simulator is then unpaused, 0.1s of time is simulated, and the simulator is paused again. Then, observations are captured (one for each drone) and returned by the `step` method:

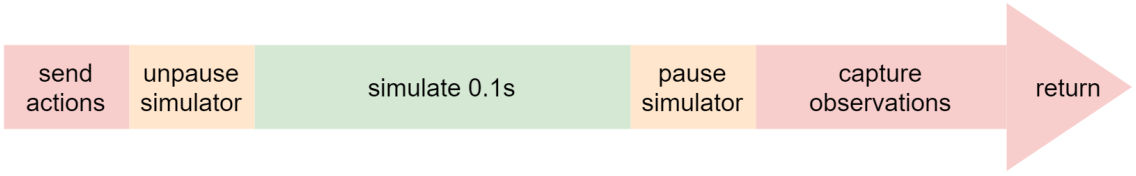

Figure 3.5 Architecture of the step method in gym-airsimdroneracinglab

Figure 3.5 gives an intuition of what is wrong in real-time scenarios with this common stepping scheme. The fact is, if one trains a RL policy in `gym-airsimdroneracinglab` and tests it in the same environment, the policy will perform well. Indeed, the algorithm would then be used to seeing actions computed and applied instantaneously, and observations being retrieved instantaneously (which is the case here because time is paused during these operations). It would also be used to seeing transitions of about 0.1s (in green) plus an additional delay caused by the communication delays between the python program and the simulator (in orange). As long as the setting used to test the learnt policy is the same (which is usually the case in RL benchmarks), there is no issue with this scheme.

However, AirSim is meant to produce realistic simulations, at least from a computer-vision point of view. Arguably its main point is sim-to-real, i.e. enabling the development in

simulation of controllers meant for the real world. In this regard, when targeting high control frequencies, it becomes a major issue to step the environment as we do in Figure 3.5 during training, and to alternate between calls to `step` and action inference. The real world, of course, cannot be paused between transitions. Therefore, the orange delays disappear, and the red delays become part of the duration of the whole time-step. Moreover, between calls to `step`, the model has to compute new actions from observations, which adds another delay. In other words, the duration of each time-step and the dynamics of the world change drastically.

As we show in the next chapters, it is possible to effectively solve most of these issues in single-agent scenarios. As a matter of fact, in `gym-airsimdroneracinglab`, the only difficulty that is not completely solved in this thesis comes from the multi-agent nature of the environment. More specifically, in AirSim, capturing camera images from the drones takes a very long amount of time, and this duration is multiplied when retrieving images for both drones instead of one. This multiplication factor is the issue that is not covered in this thesis. Nevertheless, in the conclusion, we will show how an offline dataset collected with the Dataset-Collecting Game can be used in a real-time scenario, despite being collected in a non-real-time environment.

CHAPTER 4 REINFORCEMENT LEARNING WITH RANDOM DELAYS

4.1 Refresher

This chapter presents our main contribution. We assume the reader is familiar with the mathematical basics of Reinforcement Learning. In particular, a solid understanding of probability densities, MDPs, value functions and off-policy algorithms is strongly advised. We briefly remind the reader with the concepts that are most extensively used in this chapter.

4.1.1 Markov Decision Processes

Mathematically, an MDP can be defined as:

Definition 1. A Markov Decision Process is a tuple $E = (S, A, \mu, p)$ containing:
(1) a state space S such that $s \in S$ for all reachable state s,
(2) an action space A such that $a \in A$ for all possible action a,
(3) an initial state distribution $\mu(s_0)$,
(4) a transition distribution $p(s', r'|s, a)$

where $s_0 \in S$ is an initial state sampled from μ, and s' and r' are a new state and a corresponding reward sampled from p after selecting the action a in the state s [15].

4.1.2 Policies

In RL, a *policy* $\pi(a|s)$ is a probabilistic mapping from the state-space to the action-space. In Deep RL, a policy is usually a deep neural network that maps states to actions, or to action distributions, called an *actor network*.

4.1.3 Value functions

Two value functions are commonly used in the RL literature: the *state-value* function and the *action-value* function. Both are defined under a given policy π.

The state-value function, often simply referred to as the 'value function' in this thesis, maps states to their value under the current policy π. In other words, it maps a state s to the expected sum of γ-discounted rewards that one will achieve when following the policy π from this state. It is defined recursively as follows:

Definition 2. In an MDP E and given a policy π, the state-value function is:

$$v(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[\mathbb{E}_{s',r'\sim p(\cdot|s,a)}[r' + \gamma v(s')]] \tag{4.1}$$

where $0 \leq \gamma \leq 1$ is a discount factor, which must be < 1 in the non-episodic setting (otherwise the value function could be infinite).

The action-value function refers to a similar quantity, but after selecting a certain action:

Definition 3. In an MDP E and given a policy π, the action-value function is:

$$q(s,a) = \mathbb{E}_{s',r'\sim p(\cdot|s,a)}[r' + \gamma v(s')] \tag{4.2}$$

The action-value function is particularly useful in off-policy algorithms, such as Vanilla Q-learning [50] and Soft Actor-Critic (SAC) [17, 18].

4.1.4 On-Policy and Off-Policy algorithms

From a high-level perspective, the spectrum of RL algorithms can be roughly divided in two classes: *on-policy* and *off-policy* algorithms.

On the one hand, on-policy algorithms often directly optimize the state-value function. Their particularity is that they need transitions collected under the current policy to do so.

On the other hand, off-policy algorithms often use the action-value function in order to optimize the state-value. Their particularity is that they do so using samples collected under a policy that is not necessarily the current policy.

4.1.5 Biased statistical estimators

In this thesis, we often refer to the bias-reduction properties of our estimators. An estimator $\hat{\theta}(x)$ estimates an unknown fixed parameter of interest θ, from a sample $x \sim X$. Here, X is a distribution from which we have available samples. Its *bias* is defined as follows:

Definition 4. For a given estimator $\hat{\theta}$ of a parameter θ, the bias is:

$$\text{bias}(\hat{\theta}(X)) = \mathbb{E}_{x\sim X}[\hat{\theta}(x)] - \theta \tag{4.3}$$

4.2 Reinforcement Learning and real-world delays

In the following of this chapter, we deal with a Reinforcement Learning (RL) scenario that is commonly encountered in real-world applications [28, 40, 51]. Oftentimes, actions generated by the agent are not immediately applied in the environment, and observations do not immediately reach the agent. In particular, contrary to the classic environments used in RL literature, reality cannot be paused during computations and information transfers. This implies real-time delays in action inference, action transmission, actuation, observation capture and observation transmission, that the theoretical RL literature usually ignores.

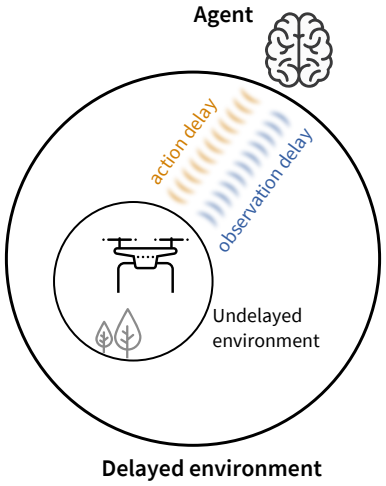

Figure 4.1 Structure of a delayed RL environment

This scenario is depicted in Figure 4.1. Yet common, it is under-explored and has mainly been studied under the unrealistic [40, 52, 53] assumption of constant delays, for which prior work proposed planning algorithms, naively trying to undelay the environment by simulating future observations [41, 42, 44].

We instead propose an off-policy, planning-free approach that enables low-bias and low-variance multi-step value estimation in environments with random delays. First, we study the anatomy of such environments in order to exploit their structure, defining Random-Delay Markov Decision Processes (RDMDP). Then, we show how to transform trajectory fragments collected under one policy into trajectory fragments distributed according to another policy. We demonstrate this principle by deriving a novel off-policy algorithm (DCAC) based on Soft Actor Critic (SAC), and exhibiting greatly improved performance in delayed environments.

We open source our repository, including a Gym wrapper that conveniently transforms any OpenAI Gym environment into a randomly delayed environment.

4.3 Delayed Environments

As illustrated in Figure 4.1, we frame the general setting of real-world Reinforcement Learning in terms of an *agent*, random *observation delays*, random *action delays*, and an *undelayed environment*. At the beginning of each time-step, the agent starts computing a new action from the most recent available delayed observation. Meanwhile, a new observation is sent and the most recent delayed action is applied in the undelayed environment. Real-valued delays are rounded up to the next integer time-step.

For a given delayed observation s_t, the *observation delay* ω_t refers to the number of elapsed time-steps from when s_t finishes being captured to when it starts being used to compute a new action. The *action delay* α_t refers to the number of elapsed time-steps from when the last action influencing s_t starts being computed to one time-step before s_t finishes being captured. We further refer to $\omega_t + \alpha_t$ as the *total delay* of s_t.

As a motivating illustration of real-world delayed setting, we have collected a dataset of communication delays between a decision-making computer and a flying robot over WiFi, summarized in Figure 4.2. In the presence of such delays, the naive approach is to simply use the last received observation. In this case, any delay longer than one time-step violates the Markov assumption, since the last sent action becomes an unobserved part of the current state of the environment. To overcome this issue, we define a Markov Decision Process that takes into account the communication dynamics.

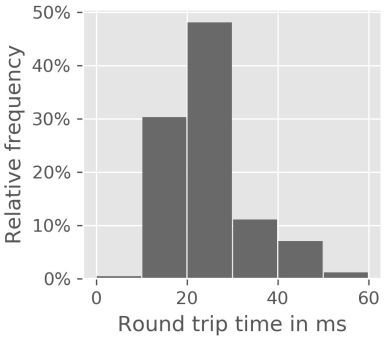

Figure 4.2 Histogram of real world WiFi delays.

4.3.1 Random Delay Markov Decision Processes

To ensure the Markov property in delayed settings, it is necessary to augment the delayed observation with at least the last K sent actions. K is the combined maximum possible observation and action delay (i.e. the maximum possible total delay). This is required as the oldest actions along with the delayed observation describe the current state of the undelayed environment, whereas the most recent actions are yet to be applied.

More specifically, in the worst-case scenario, the total delay is repeatedly K. In other words, the worst-case scenario is equivalent to a constantly delayed setting with a total delay of K. Such scenario can be visualized in Figure 4.3:

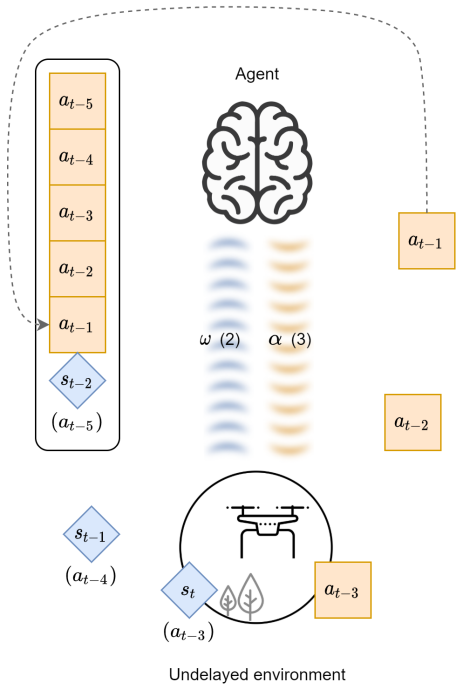

Figure 4.3 Visual example of a Constant Delay MDP

This example features a constant action delay of 3 time-steps and a constant observation delay of 2 time-steps, summing to a constant total delay of $K = 5$ time-steps. Here, for the sake of time-wise visualization, the actions are indexed by the time at which they started being produced, and the observations are indexed by the time at which they finished being captured. At the level of the agent, the observation s_{t-2}, produced 2 time-steps ago, is received and augmented with an action-buffer of the 5 last computed actions. This augmented observation is then used by the agent to compute the action a_t (not represented as it is only starting to be computed at the current time t). Meanwhile, in the undelayed environment, the action a_{t-3} is received and the observation s_t is fully retrieved.

Using this augmentation with an action-buffer is sufficient to ensure the Markov property, not only in constantly delayed environments, but also in randomly delayed environments. This can be visualized in Figure 4.4, featuring a similar example with random action and observation delays. In this example, the action delay is $\alpha \leq 3$ time-steps while the observation delay is $\omega \leq 2$ time-steps, which again sums to a maximum possible total delay of $K = 5$ time-steps. In this situation, actions and observations can be superseded due to the stochastic nature of their delays. When this happens, we only keep the most recently produced actions and observations, and we discard the others (in red). This is because the undelayed environment is an MDP, and thus older observations that originated from it contain only outdated information. This

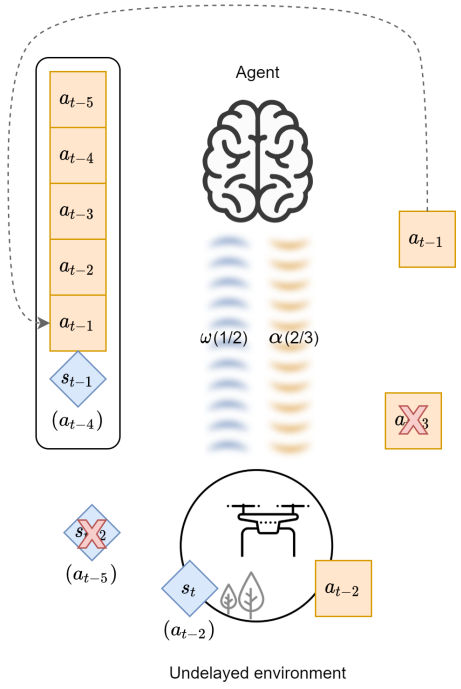

Figure 4.4 Visual example of a Random Delay MDP

is also because we are interested in continuous control, where it is preferable to actuate the most recent available action. However, when discarding older actions is not possible or not suitable, this can be relaxed with no impact on our analysis, as we will see later in this thesis.

The reason why augmenting the state-space with an action-buffer of the last K actions ensures the Markov property is twofold. On the one hand, given an observation s with an observation delay ω and an action delay α, the ω oldest actions of the buffer along with s are a sufficient statistic of the current state of the undelayed environment. On the other hand, the other actions of the buffer are yet to be applied and thus, along with the current state of the undelayed environment, they are a sufficient statistic of the future state at which the currently computed action will be applied in the undelayed environment.

Nevertheless, it is possible to design faster-converging algorithms by taking advantage of the problem's structure when the delays themselves are also part of the state-space (and not only the action buffer). First, this allows us to mathematically model self-correlated delays, e.g. discarding outdated actions and observations. Second, this provides useful information to the model about how old an observation is and what actions have been applied next. Third, knowledge over the total delay allows for efficient credit assignment and off-policy partial trajectory resampling, as we will now show in this thesis.

We propose Random Delay Markov Decision Processes (RDMDPs). RDMDPs are MDPs

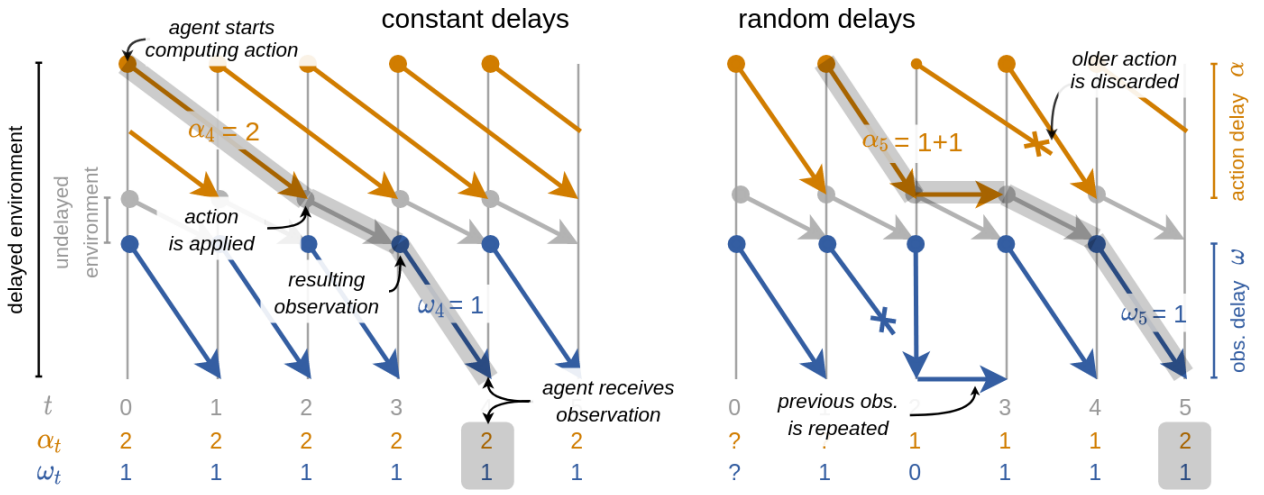

Figure 4.5 Influence of actions on delayed observations in delayed environments.

(therefore they are compatible with the whole RL theory related to MDPs) and they generally describe real-world environments with variable delays. This mathematical object is fairly complex, because α and ω have recursive and interleaved dynamics. A precise understanding of the RDMDP definition is suitable to follow the proofs, but is not required to follow the further development of this thesis. Instead, Figure 4.5 conveys an intuitive understanding of random delays dynamics, and can be substituted to Definition 5. It will also help the interested reader understand this definition:

Definition 5. A Random Delay Markov Decision Process $RDMDP(E, p_\omega, p_\alpha) = (X, A, \tilde{\mu}, \tilde{p})$, augments a Markov Decision Process $E = (S, A, \mu, p)$ with:
(1) state space $X = S \times A^K \times \mathbb{N}^2$, (2) action space A,
(3) initial state distribution $\tilde{\mu}(x_0) = \tilde{\mu}(\,s, u, \omega, \alpha\,) = \mu(s)\ \delta(u - c_u)\ \delta(\omega - c_\omega)\ \delta(\alpha - c_\alpha)$,
(4) transition distribution $\tilde{p}(\,s',u',\omega',\alpha'\,,r'|\,s,u,\omega,\alpha\,,a) = f_{\omega-\omega'}(s',\alpha',r'|\,s,u,\omega,\alpha\,,a)\,p_\omega(\omega'|\omega)\,p_u(u'|u,a)$,

where δ is the Dirac delta distribution[1], $s \in S$ is the unaugmented delayed observation, $u \in A^K$ is a buffer of the last K sent actions, $\omega \in \mathbb{N}$ is the *observation delay*, and $\alpha \in \mathbb{N}$ is the *action delay* as defined above. To avoid conflicting with the subscript notation that we use later in this thesis, we index the action buffers' elements using square brackets. Here, $u_{[1]}$ is the most recent and $u_{[K]}$ is the oldest action in the buffer. We denote slices by $u_{[i:j]} = (u_{[i]}, \ldots, u_{[j]})$ and $u_{[i:-j]} = (u_{[i]}, \ldots, u_{[K-j]})$. We slightly override this notation and additionally define $u_{[0]} = a$. The constants $c_u \in A^K$ and $c_\omega, c_\alpha \in \mathbb{N}$ initialize u, ω, α. The transition distribution $\tilde{p}$ itself is composed of three parts: (1) The observation delay distribution p_ω modelling the evolution of observation delays. Note that this density function must represent a discrete distribution

[1]When marginalized out, the Dirac delta distribution $\delta(y - z)$ sets $y = z$ with probability 1.

(i.e. be a weighted sum of Dirac delta distributions). Furthermore, this process will repeat observations if there are no new ones available. This means that the observation delay can maximally grow by one from one time-step to the next. (2) The transition distribution for the action-buffer $p_u(u'|u,a) = \delta(u' - (a, u_{[1:-1]}))$. (3) The distribution f_Δ describing the evolution of observations, rewards and action delays (Definition 6).

Definition 6. For each change in observation delays ($\Delta{=}\omega{-}\omega'$) we define a variable step update distribution f_Δ as

$$f_\Delta(s',\alpha',r'|\,s,u,\omega,\alpha\,,a){=}\mathbb{E}_{\bar{s},\bar{\alpha},\bar{r}\sim f_{\Delta-1}(\cdot|\,s,u,\omega,\alpha\,,a)}[p(s',r'{-}\bar{r}|\bar{s},u_{[\overbrace{\omega-\Delta}^{\omega'}+\alpha']})\ p_\alpha(\alpha'|\bar{\alpha})]. \tag{4.4}$$

The base case of the recursion is $f_{-1}(s',\alpha',r' \mid \,s,u,\omega,\alpha\,,a) = \delta(s'-s)\ \delta(\alpha'-\alpha)\ \delta(r')$.

Here, p_α is the action delay distribution which, similar to p_ω, must be discrete. The transition distribution of the underlying, undelayed MDP is p. Finally, the $r' - \bar{r}$ term accumulates intermediate rewards in case observations are skipped. Indeed,

$$\mathbb{E}_{\bar{r}\sim f}[p(r'-\bar{r})] = \int p(r'-\bar{r})f(\bar{r})d\bar{r}$$

is a convolution, i.e. a density of the sum of two random variables: r' and $\bar{r}$ [54].

We have implicitly made a choice here. Keep in mind that observations can be dropped (superseded) at the level of the agent. In such case, $\bar{s}$ represents a skipped observation and $\bar{r}$ represents the corresponding reward which could have been skipped as well. Instead, we chose to accumulate the rewards corresponding to the lost transitions. When an observation gets repeated because no new observation is available, the corresponding reward is 0, and when a new observation arrives, the corresponding reward contains the sum of intermediate rewards in lost transitions.

In practice, this is ensured for example by making the assumption that the remote robot (i.e. the undelayed environment) can observe its own instantaneous reward. This allows the robot to compute its cumulative reward and send it to the agent along with the observation. The agent can then compute the difference between the last cumulative reward it received from the remote robot and the new one for each incoming observation (NB: outdated observations are discarded so the agent only sees cumulative rewards with time-increasing timestamps).

Alternatively, when a trick similar to the aforementioned cannot be implemented, the practitioner can choose to repeat the delayed rewards along with the repeated delayed observations at the level of the agent (this is what we do in earlier versions of this work). This has no impact on our analysis. However, the reward signal will inherently have a higher variance.

The observation delay cannot increase by more than 1, because this is what happens when an observation is repeated (otherwise ω either stays the same or decreases). In such situation, $\Delta = \omega - \omega' = -1$, and f_{-1} is used. Note that f_{-1} repeats s in s' (the unaugmented observation is repeated), that it repeats α in α' (the action delay of the repeated observation doesn't change, only the observation delay is incremented), and that it sets $r' = 0$ (we chose not to repeat rewards when observations are repeated: instead the cumulative sum of skipped rewards is received when a more recent observation is received).

To further understand how f_Δ works, let us look at what happens when there is no change in observation delay, i.e. $\Delta = \omega - \omega' = 0$. Then f_0 is used. According to Definition 6:

$$f_0(s', \alpha', r' \mid s, u, \omega, \alpha, a) = p(s', r' | s, u_{[\overbrace{\omega}^{\omega'} + \alpha']}) \; p_\alpha(\alpha' | \alpha)$$

where $\bar{s}, \bar{\alpha}, \bar{r} \sim f_{-1}(\cdot | s, u, \omega, \alpha)$ have been marginalized out (replaced by s, α and 0). Note that the new action delay α' is then sampled in the action delay distribution $p_\alpha(\alpha'|\alpha)$. Note also that s' and r' are then sampled from the transition distribution of the undelayed environment, but where the applied action is the $(\omega' + \alpha')^{\text{th}}$ action in the buffer. Indeed, this action is the action that was applied in the undelayed environment ω' time-steps ago, and that started being produced by the agent $\omega' + \alpha'$ time-steps ago.

Finally, a last insightful example is $\Delta = 1$, i.e. $\omega' = \omega - 1$. In this situation, an observation has been superseded (e.g. see $t = 2$ in Figure 4.5 - right). According to Definition 5:

$$\begin{aligned} f_1(s', \alpha', r' | s, u, \omega, \alpha, a) &= \mathbb{E}_{\bar{s}, \bar{\alpha}, \bar{r} \sim p(\bar{s}, \bar{r} | s, u_{[\omega + \bar{\alpha}]}) \; p_\alpha(\bar{\alpha}|\alpha)} [p(s', r' - \bar{r} | \bar{s}, u_{[\omega - 1 + \alpha']}) \; p_\alpha(\alpha' | \bar{\alpha})] \\ &= \iiint p(s', r' - \bar{r} | \bar{s}, u_{[\omega - 1 + \alpha']}) \; p(\bar{s}, \bar{r} | s, u_{[\omega + \bar{\alpha}]}) \; p_\alpha(\alpha' | \bar{\alpha}) p_\alpha(\bar{\alpha} | \alpha) d\bar{s} d\bar{\alpha} d\bar{r} \end{aligned}$$

Here, $\bar{s}$ is a skipped unaugmented observation, $\bar{\alpha}$ is its corresponding action delay, and $\bar{r}$ is its corresponding reward. They are marginalized out in order to yield the new variables s', α', r' which are part of the transition actually seen by the agent (whereas $\bar{s}, \bar{\alpha}, \bar{r}$ are not observed by the agent). Indeed, at the level of the undelayed environment, two transitions occurred:

$$(s, r) \xrightarrow[u_{[\omega + \bar{\alpha}]}]{\alpha \to \bar{\alpha}} (\bar{s}, \bar{r}) \xrightarrow[u_{[\omega - 1 + \alpha']}]{\bar{\alpha} \to \alpha'} (s', r' - \bar{r}).$$

A simple special case of the RDMDP is the constant observation and action delay case with $p_\omega(\omega'|\omega) = \delta(\omega' - c_\omega)$ and $p_\alpha(\alpha'|\alpha) = \delta(\alpha' - c_\alpha)$. Here, the RDMDP reduces to a Constant Delay Markov Decision Process, described by [41]. In this case, the action and observation delays α, ω can be removed from the state-space as they do not carry information.

4.4 Reinforcement Learning in Delayed Environments

Delayed environments as described in Section 4.3 are specific types of MDP, with an augmented state-space and delayed dynamics. Therefore, using this augmented state-space, traditional algorithms such as Soft Actor Critic (SAC) [17, 18] will always work in randomly delayed settings. However their performance will still deteriorate because of the more difficult credit assignment caused by delayed observations and rewards, on top of the exploration and generalization burdens of delayed environments. We now analyze how to compensate for the credit assignment difficulty by leveraging our knowledge about the delays' dynamics.

One class of solutions is to perform *on-policy* multi-step rollouts on sub-trajectories that are longer than the considered delays. However, on-policy algorithms are known to be sample-inefficient [1] and therefore are not commonly used in real-world applications, where data collection is costly. This motivates the development of *off-policy* algorithms able to reuse old samples, such as SAC.

Intuitively, in delayed environments, one should take advantage of the fact that actions only influence observations and rewards after a number of time-steps relative to the beginning of their computation (the total delay $\omega+\alpha$). Since the delay information is part of the state-space, it can be leveraged to track the action influence through time. However, applying conventional off-policy algorithms in delayed settings leads to the following issue: the trajectories used to perform the aforementioned multi-step backups have been sampled under an outdated policy, and therefore contain outdated action buffers. In this section, we propose a method to tackle this issue by performing *partial trajectory resampling*. We make use of the fact that the delayed dynamics are known to simulate the effect they would have had under the current policy, effectively transforming off-policy sub-trajectories into on-policy sub-trajectories. This enables us to derive a family of efficient off-policy algorithms for randomly delayed settings.

4.4.1 Partial Trajectory Resampling in Delayed Environments

Given a policy μ, we define the distribution of n-step sub-trajectories reachable from x:

Definition 7. The n-step state-reward distribution for a $RDMDP(E, p_\omega, p_\alpha) = (X, A, \tilde{\mu}, \tilde{p})$ and a policy μ is defined as

$$p^\mu_{n+1}(\underbrace{x', r', \tau_n}_{\tau_{n+1}}|x) = \mathbb{E}_{a\sim\mu(\cdot|x)}[p^\mu_n(\tau_n|x')\tilde{p}(x', r'|x, a)] = \int_A p^\mu_n(\tau_n|x')\tilde{p}(x', r'|x, a)\mu(a|x)da \quad (4.5)$$

with the base case $p^\mu_0(x) = 1$ and the first iterate $p^\mu_1(x', r'|x) = \int_A \tilde{p}(x', r'|, a)\mu(a|x)da$.

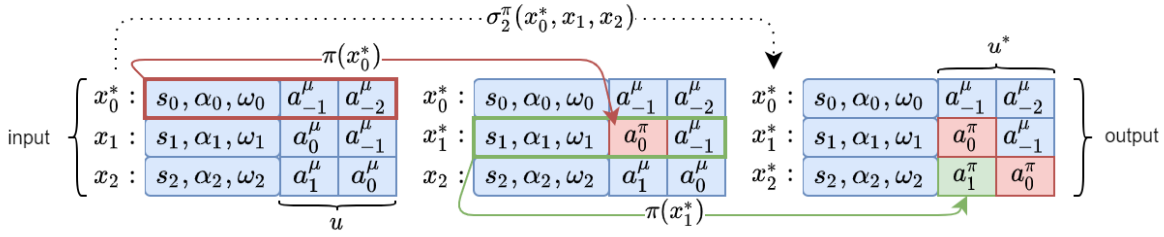

Figure 4.6 Partial resampling of a small sub-trajectory.

As previously seen in Figure 4.5, given the delayed dynamics of RDMDPs, some actions contained in the action-buffer of an off-policy state do not influence the subsequent delayed observations and rewards for a number of time-steps. We show that, if an off-policy sub-trajectory is short enough, it is possible to recursively resample its action buffers with no influence on the return. We propose the following transformation of off-policy sub-trajectories:

Definition 8. The partial trajectory resampling operator recursively updates action buffers as follows

$$\sigma_n^\pi(\underbrace{s_1^*, u_1^*, \omega_1^*, \alpha_1^*}_{x_1^*}, r_1^*, \tau_{n-1}^* | x_0^*; \underbrace{s_1, u_1, \omega_1, \alpha_1}_{x_1}, r_1, \tau_{n-1}) \\ = \delta((s_1^*, \omega_1^*, \alpha_1^*, r_1^*) - (s_1, \omega_1, \alpha_1, r_1)) \mathbb{E}_{a_0 \sim \pi(\cdot|x_0^*)}[\delta(u_1^* - (a_0, u_{0[1:-1]}^*))] \; \sigma_{n-1}^\pi(\tau_{n-1}^* | x_1^*; \tau_{n-1}) \quad (4.6)$$

with trivial base case $\sigma_0(x_0^*) = 1$

This operator recursively resamples the most recent actions of each action-buffer in an input sub-trajectory τ_n, according to a new policy π. Everything else stays unchanged. A visual example is provided in Figure 4.6 with $n = 2$ and an action buffer of 2 actions (since rewards are not modified by σ, they are omitted in this illustration).

When resampled actions are delayed and would not affect the environment, they do not "invalidate" the sub-trajectory. The resampled trajectories can then be considered on-policy:

Theorem 1. The partial trajectory resampling operator σ_n^π (Def. 8) transforms off-policy trajectories into on-policy trajectories

$$\mathbb{E}_{\tau_n \sim p_n^\mu(\cdot|x_0)}[\sigma_n^\pi(\tau_n^* | x_0; \tau_n)] = p_n^\pi(\tau_n^* | x_0) \quad (4.7)$$

on the condition that none of the delayed observations depend on any of the resampled actions, i.e.

$$\omega_t^* + \alpha_t^* \geq t \quad (4.8)$$

where t indexes the trajectory $\tau_n^* = (s_1^*, u_1^*, \omega_1^*, \alpha_1^*, r_1^*, \ldots, s_n^*, u_n^*, \omega_n^*, \alpha_n^*, r_n^*)$ from 1 to n.

The condition of Theorem 1 (Equation 4.8) can be understood visually with the help of Figure 4.5. In the constant delay example it is fulfilled until the third timestep. After that the observations would have been influenced by the resampled actions (starting with a_0).

Theorem 1 is the core of our contribution. It says that, in a randomly delayed environment, if a sub-trajectory τ_n collected under an old policy μ is short enough, the partial resampling operator σ_n^π transforms this sub-trajectory into an equivalent on-policy sub-trajectory τ_n^*. In other words, it exactly transforms this sub-trajectory into what it would have been under the current policy π. The mathematical proof is fairly involved, but once it is proven, Theorem 1 yields a straightforward class of efficient algorithms for randomly delayed settings. We prove Theorem 1 in three intermediate steps:

Lemma on a Dirac delta product distribution

In order to prove Theorem 1, we first prove a few intermediate results. The first one is fairly intuitive:

Lemma 1. Let $p(u, v) = \delta(u - c)q(u, v)$
If $q(c, v) < \infty$, then $p(u, v) = \delta(u - c)q(c, v)$.

Proof. If $u = c$ then $p(u, v) = \delta(u - c)q(c, v)$, otherwise $p(u, v) = 0 = \delta(u - c)q(c, v)$ □

Lemma on f_Δ

The second result we need says two things. First, the output of f_Δ does not depend on the action passed as input (this is obvious from Definition 6: the a term is there so the parallel between f_Δ and a usual transition distribution is more intuitive to the reader), therefore this action can be arbitrarily rewritten. Second, there exists a number of actions in the buffer that can also be arbitrarily rewritten:

Lemma 2. The dynamics described by f_Δ depend neither on the input action nor on a range of actions in the action buffer:

$$f_\Delta(s_1^*, \alpha_1^*, r_1^* | x_0, a_0^\mu) = f_\Delta(s_1^*, \alpha_1^*, r_1^* | x_0^*, a_0^\pi)$$

with $x_0 = s_0, u_0, \omega_0, \alpha_0$ and $x_0^* = s_0^*, u_0^*, \omega_0^*, \alpha_0^*$, given that $s_0, \omega_0, \alpha_0 = s_0^*, \omega_0^*, \alpha_0^*$ and given

$$u_0[\omega_0^* - \delta + \alpha_1^*] = u_0^*[\omega_0^* - \delta + \alpha_1^*] \quad \text{for all} \quad \delta \in \{\Delta, \Delta - 1, \dots, 0\}$$

Proof. We prove this result by induction.

The base case ($\omega_0^* - \omega_1^* = -1$) is trivial since it does not depend on the inputs that differ.

Our induction hypothesis is that Lemma 2 is true at the previous induction step ($\Delta - 1$).

For the induction step, let us consider that the condition of Lemma 2 is checked at the new induction step (Δ).

$$f_\Delta(s_1^*, \alpha_1^*, r_1^* | \, s_0, u_0, \omega_0, \alpha_0 \, , a_0^\mu) = \\ \mathbb{E}_{\bar{s},\bar{\alpha},\bar{r} \sim f_{\Delta-1}(\cdot | \, s_0,u_0,\omega_0,\alpha_0 \, , a_0^\mu)} [p(s_1^*, r_1^* - \bar{r} | \bar{s}, u[\omega_0 - \Delta + \alpha_1^*]) \; p_\alpha(\alpha_1^* | \bar{\alpha})] \quad (4.9)$$

Because of the condition on u_0 and u_0^* and the fact that $\omega_0 = \omega_0^*$ this is equal to:

$$\mathbb{E}_{\bar{s},\bar{\alpha},\bar{r} \sim f_{\Delta-1}(\cdot | \, s_0,u_0,\omega_0,\alpha_0 \, , a_0^\mu)} [p(s_1^*, r_1^* - \bar{r} | \bar{s}, u_0^*[\omega_0^* - \Delta + \alpha_1^*]) \; p_\alpha(\alpha_1^* | \bar{\alpha})]$$

We can now use the induction hypothesis since the conditions on $s_0, u_0, \omega_0, \alpha_0$ are still met when $\Delta \leftarrow \Delta - 1$:

$$f_\Delta(s_1^*, \alpha_1^*, r_1^* | \, s_0, u_0, \omega_0, \alpha_0 \, , a_0^\mu) = \\ \mathbb{E}_{\bar{s},\bar{\alpha},\bar{r} \sim f_{\Delta-1}(\cdot | \, s_0^*,u_0^*,\omega_0^*,\alpha_0^* \, , a_0^\pi)} [p(s_1^*, r_1^* - \bar{r} | \bar{s}, u_0^*[\omega_0^* - \Delta + \alpha_1^*]) \; p_\alpha(\alpha_1^* | \bar{\alpha})] \\ = f_\Delta(s_1^*, \alpha_1^*, r_1^* | \, s_0^*, u_0^*, \omega_0^*, \alpha_0^* \, , a_0^\pi) \quad (4.10)$$

□

Lemma on partial resampling

We now prove our main result, which is slightly more general than Theorem 1. However, in the context of this thesis, Theorem 1 is the result that will ultimately be useful.

The Lemma says that, under specific conditions on the initial state x_0 and the following n-step sub-trajectory collected under μ, the partial resampling operator transforms this trajectory into a sub-trajectory that is on-policy with respect to the current policy π. Compared to Theorem 1, it provides some additional freedom on the action buffers of the resampled trajectory, but we do not make use of this freedom in this thesis.

Lemma 3. Partially resampling trajectories collected under a policy μ according to σ_n^π transforms them into trajectories distributed according to π:

$$\mathbb{E}_{\tau_n \sim p_n^\mu(\cdot|x_0)}[\sigma_n^\pi(\tau_n^*|x_0^*;\tau_n)] = p_n^\pi(\tau_n^*|x_0^*)$$

with $x_0 = s_0, u_0, \omega_0, \alpha_0$ and $x_0^* = s_0^*, u_0^*, \omega_0^*, \alpha_0^*$, on the condition that $s_0, \omega_0, \alpha_0 = s_0^*, \omega_0^*, \alpha_0^*$ and on the condition that the actions in the initial action buffers u_0 and u_0^* that are applied in the following trajectory are the same, i.e.:

$$k > 0 \text{ and } u_0[k:\text{end}] = u_0^*[k:\text{end}] \quad \text{with} \quad k = \min_i(\omega_{i+1}^* + \alpha_{i+1}^* - i) \quad \text{for} \quad i \in \{0, n-1\}$$

$$\text{and for the trajectory } \tau_n^* = (s_1^*, u_1^*, \omega_1^*, \alpha_1^*, \ldots, s_n^*, u_n^*, \omega_n^*, \alpha_n^*).$$

Proof. We start with the induction base for $n = 0$.

The lemma is trivial in this case since we have 0-length trajectories () and we defined $p_0^\mu(()|x_0) = \sigma_0^\pi(()|x_0^*;()) = p_0^\pi(()|x_0^*) = 1$.

For the induction step, we consider that the condition of Lemma 3 is met. We start with the left hand side of the Lemma's main equation:

$$\mathbb{E}_{\tau_n \sim p_n^\mu(\cdot|x_0)}[\sigma_n^\pi(\tau_n^*|x_0^*;\tau_n)]$$
$$= \mathbb{E}_{a_0^\mu \sim \mu(\cdot|x_0)}[\mathbb{E}_{x_1,r_1 \sim \tilde{p}(x_1,r_1|x_0,a_0^\mu)}[\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|x_1)}[\sigma_n^\pi(\tau_n^*|x_0^*;x_1,r_1,\tau_{n-1})]]] \quad (4.11)$$

with (Definition 5):

$$\tilde{p}(s_1, u_1, \omega_1, \alpha_1, r_1 | s_0, u_0, \omega_0, \alpha_0, a_0^\mu) = f_{\omega_0-\omega_1}(s_1, \alpha_1, r_1 | s_0, u_0, \omega_0, \alpha_0, a_0^\mu)\, p_\omega(\omega_1|\omega_0)\, p_u(u_1|u_0, a_0^\mu)$$

Plugging this and solving the integral over u_1 (according to Definition 5, p_u then replaces u_1 by $(a_0^\mu, u_0[1:-1])$) yields:

$$= \mathbb{E}_{a_0^\mu \sim \mu(\cdot|x_0)}[\mathbb{E}_{\omega_1 \sim p_\omega(\cdot|\omega_0)}[\mathbb{E}_{s_1,\alpha_1,r_1 \sim f_{\omega_0-\omega_1}(\cdot|s_0,u_0,\omega_0,\alpha_0,a_0^\mu)}[$$
$$\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|s_1,(a_0^\mu,u_0[1:-1]),\omega_1,\alpha_1)}[\sigma_n^\pi(\tau_n^*|x_0^*; s_1, (a_0^\mu, u_0[1:-1]), \omega_1, \alpha_1, r_1, \tau_{n-1})]]]] \quad (4.12)$$

Rolling out σ_n^π by one step and integrating out $s_1, \omega_1, \alpha_1, r_1$ yields:

$$= \mathbb{E}_{a_0^\mu \sim \mu(\cdot|x_0)}[\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|\, s_1^*, (a_0^\mu, u_0[1:-1]), \omega_1^*, \alpha_1^*)}[\mathbb{E}_{a_0^\pi \sim \pi(\cdot|x_0^*)}[\delta(u_1^* - (a_0^\pi, u_0^*[1:-1]))$$
$$\sigma_{n-1}^\pi(\tau_{n-1}^*|\, s_1^*, u_1^*, \omega_1^*, \alpha_1^*; \tau_{n-1}) f_{\omega_0 - \omega_1^*}(s_1^*, \alpha_1^*, r_1^*|\, s_0, u_0, \omega_0, \alpha_0, a_0^\mu)\ p_\omega(\omega_1^*|\omega_0)]]] \quad (4.13)$$

Reordering terms and substituting $s_0, \omega_0, \alpha_0 = s_0^*, \omega_0^*, \alpha_0^*$ yields:

$$= p_\omega(\omega_1^*|\omega_0^*)\mathbb{E}_{a_0^\pi \sim \pi(\cdot|x_0^*)}[\delta(u_1^* - (a_0^\pi, u_0^*[1:-1]))$$
$$\mathbb{E}_{a_0^\mu \sim \mu(\cdot|x_0)}[f_{\omega_0^* - \omega_1^*}(s_1^*, \alpha_1^*, r_1^*|x_0, a_0^\mu)$$
$$\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|\, s_1^*, (a^\mu, u_0[1:-1]), \omega_1^*, \alpha_1^*)}[\sigma_{n-1}^\pi(\tau_{n-1}^*|x_1^*; \tau_{n-1})]]] \quad (4.14)$$

We can now substitute the f term according to Lemma 2 since the condition between x_0 and x_0^* are met. More precisely, the condition on u_0 and u_0^* is met because $k \leq \omega_0^* - \Delta + \alpha_1^* = \omega_1^* + \alpha_1^*$. After the substitution we have:

$$= p_\omega(\omega_1^*|\omega_0^*)\mathbb{E}_{a_0^\pi \sim \pi(\cdot|x_0^*)}[\delta(u_1^* - (a_0^\pi, u_0^*[1:-1]))\ f_{\omega_0^* - \omega_1^*}(s_1^*, \alpha_1^*, r_1^*|x_0^*, a_0^\pi)$$
$$\mathbb{E}_{a_0^\mu \sim \mu(\cdot|x_0)}[\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|\, s_1^*, (a^\mu, u_0[1:-1]), \omega_1^*, \alpha_1^*)}[\sigma_{n-1}^\pi(\tau_{n-1}^*|x_1^*; \tau_{n-1})]]] \quad (4.15)$$

We can substitute the induction hypothesis in the following form:

$$\mathbb{E}_{\tau_{n-1} \sim p_{n-1}^\mu(\cdot|x_1)}[\sigma_{n-1}^\pi(\tau_{n-1}^*|x_1^*; \tau_{n-1})] = p_{n-1}^\pi(\tau_{n-1}^*|x_1^*)$$

on the condition that:

$$k > 0 \text{ and } u_1[k : \text{end}] = u_1^*[k : \text{end}] \quad \text{with} \quad k = \min_i(\omega_{i+2}^* + \alpha_{i+2}^* - i) \quad \text{for} \quad i \in \{0, n-2\}$$

for the trajectory $\tau_{n-1}^* = (s_2^*, u_2^*, \omega_2^*, \alpha_2^*, \ldots, s_n^*, u_n^*, \omega_n^*, \alpha_n^*)$

(NB: The recursion moves forward in the trajectory when n decrements. With respect to u_1^*, $\omega_{i+2}^* + \alpha_{i+2}^* - i$ is the index in u_1^* of the action applied at step $i+2$).

To check that this condition is met we observe that $u_1 = (a_0^\mu, u_0[1:-1])$ and substitute $u_1^* = (a_0^\pi, u_0^*[1:-1])$ (this is made possible by Lemma 1). Since the condition of Lemma 3 is met for u_0 and u_0^*, it also implies:

$$k'>1 \quad \text{and} \quad u_0[k'-1\text{:end}]=u_0^*[k'-1\text{:end}] \quad \text{with} \quad k'=\min_i(\omega_{i+2}^*+\alpha_{i+2}^*-i) \quad \text{for} \quad i\in\{0,n-2\}$$

And since $u_1[k'>1] = u_0[k'-1]$ (same for u_1^* and u_0^*) the condition on u_1 and u_1^* follows. Substituting the induction hypothesis yields:

$$= p_\omega(\omega_1^*|\omega_0^*)\mathbb{E}_{a_0^\pi\sim\pi(\cdot|x_0^*)}[\delta(u_1^*-(a_0^\pi, u_0[1:-1]))\, f_{\omega_0^*-\omega_1^*}(s_1^*, \alpha_1^*, r_1^*|x_0^*, a_0^\pi)\, p_{n-1}^\pi(\tau_{n-1}^*|x_1^*)] \quad (4.16)$$

which is

$$\mathbb{E}_{a_0^\pi\sim\pi(\cdot|x_0^*)}[p_{n-1}^\pi(\tau_{n-1}^*|x_1^*)\ \tilde{p}(x_1^*, r_1^*|x_0^*, a_0^\pi)] = p_n^\pi(\tau_n^*|x_0^*)$$

□

Proof of Theorem 1

Proof. The Theorem is a special case of Lemma 3 with $x_0 = x_0^*$. This allows us to simplify the condition in the Lemma.

Since $u_0 = u_0^*$ we can allow all $k \geq 1$ which is the minimum allowed index for u. Therefore we must ensure $1 \leq \min_i(\omega_i^* + \alpha_i^* - i)$. Since the min must be larger than 1 then all arguments must be larger than 1 which means this is equivalent to:

$$1 \leq \omega_i^* + \alpha_i^* - i \quad \text{for} \quad i \in \{0, n-1\}.$$

This can be transformed into:

$$\omega_t^* + \alpha_t^* \geq t \quad \text{for} \quad i \in \{1, n\} \quad (4.17)$$

□

4.4.2 Multistep Off-Policy Value Estimation in Delayed Environments

We have shown in Section 4.4.1 how it is possible to transform off-policy sub-trajectories into on-policy sub-trajectories in the presence of random delays. From this point, we can derive a family of efficient off-policy algorithms for the randomly delayed setting. For this matter, we make use of the classic on-policy Monte-Carlo n-step value estimator:

Definition 9. The n-step state-value estimator is defined as

$$\hat{v}_n(x_0;\ \underbrace{x_1^*, r_1^*, \tau_{n-1}^*}_{\tau_n^*}) = r_1^* + \gamma \hat{v}_{n-1}(x_1^*;\ \tau_{n-1}^*) = \sum_{i=1}^{n} \gamma^{i-1} r_i^* + \gamma^n \hat{v}_0(x_n^*). \tag{4.18}$$

where $\hat{v}_0$ is a state-value function approximator (e.g. a neural network).

Indeed, in γ-discounted RL, performing on-policy n-step rollouts to estimate the value function reduces the bias introduced by the function approximator by a factor of γ^n:

Lemma 4. The n-step value estimator has the following bias:

$$\text{bias}(\hat{v}_n(x_0, \cdot)) = \gamma^n \mathbb{E}_{\ldots, x_n^*, r_n^* \sim p_n^\pi(\cdot|x_0)}[\text{bias}(\hat{v}_0(x_n^*))] \tag{4.19}$$

Proof.

$$\begin{aligned}
&\text{bias}(\hat{v}_n(x_0, \cdot)) = \mathbb{E}_{\tau_n^* \sim p_n^\pi(\cdot|x_0)}[\hat{v}_n(x_0, \tau_n^*) - v^\pi(x_0)] \\
&= \mathbb{E}_{\tau_n^* \sim p_n^\pi(\cdot|x_0)}[r_1^* + \gamma \hat{v}_{n-1}(x_1^*; \tau_{n-1}^*)] - \mathbb{E}_{a_0 \sim \pi(\cdot|x_0)}[\mathbb{E}_{r_1^*, x_1^* \sim \tilde{p}(\cdot|x_0, a_0)}[r_1^* + \gamma v^\pi(x_1^*)]] \\
&= \mathbb{E}_{\tau_n^* \sim p_n^\pi(\cdot|x_0)}[r_1^* + \gamma \hat{v}_{n-1}(x_1^*; \tau_{n-1}^*) - r_1^* - \gamma v^\pi(x_1^*)] \\
&= \gamma \mathbb{E}_{\tau_n^* \sim p_n^\pi(\cdot|x_0)}[\hat{v}_{n-1}(x_1^*; \tau_{n-1}^*) - v^\pi(x_1^*)] \\
&= \ldots \\
&= \gamma^n \mathbb{E}_{\tau_n^* \sim p_n^\pi(\cdot|x_0)}[\hat{v}_0(x_n^*) - v^\pi(x_n^*)] \\
&= \gamma^n \mathbb{E}_{\ldots, x_n^*, r_n^* \sim p_n^\pi(\cdot|x_0)}[\text{bias}(\hat{v}_0(x_n^*))]
\end{aligned}$$

□

A simple corollary of Lemma 4 is that the on-policy n-step value estimator is unbiased when the function approximator $\hat{v}_0$ is unbiased. On the other hand, Theorem 1 provides a recipe for transforming sub-trajectories collected under old policies into actual on-policy sub-trajectories. From a given state in an off-policy trajectory, this is done by applying σ_n^π to all the subsequent transitions until we meet a total delay $(\omega_i + \alpha_i)$ that is shorter than the length of the formed

sub-trajectory. Consequently, the transformed sub-trajectory can be fed to the on-policy n-step value estimator, where n is the length of this sub-trajectory. Not only this yields a better value estimate than usual 1-step off-policy estimators according to Lemma 4, but this maximally compensates for the multi-step credit assignment difficulty introduced by random delays. Indeed, the length of the transformed sub-trajectory is then exactly the number of time-steps it took the first action of the sub-trajectory to have an influence on subsequent delayed observations, minus one time-step.

As opposed to other unbiased n-step off-policy methods such as importance sampling [55] and Retrace [56], this method doesn't suffer from variance explosion. This is because the presence of delays allows us to transform off-policy sub-trajectories into on-policy sub-trajectories, so that old samples don't need to be weighted by the policy ratio.

Although we use a multi-step state-value estimator, the same procedure can be applied to action-value estimation as well. In fact, the trajectory transformation described in Definition 8 enables efficient off-policy n-step value estimation in any value-based algorithm that would otherwise perform 1-step action-value backups, such as DQN [57], DDPG [58] or SAC [17,18]. In the next section, we will illustrate this principle by improving SAC in randomly delayed environments.

Figure 4.7 summarizes the whole procedure in a simple 1D-world example. The maximum possible delay is $K = 3$ here, and the agent can only go 'left' or 'right'. The original trajectory has been sampled under the old policy μ: 'always go left', whereas the current policy is π: 'always go right'. An initial augmented state x_0 is sampled from the replay memory, along with the 3 subsequent augmented states and rewards. The condition of Theorem 1 is satisfied for $n \leq 2$. It follows that $\tau_n = \tau_2 = (x_1, x_2)$. This off-policy trajectory fragment is partially resampled, which yields the corresponding on-policy trajectory fragment $\tau_n^* = \tau_2^*$. This on-policy trajectory fragment can then be used to compute an unbiased n-step value estimate of the initial state $x_0 = x_0^*$, that is, $\hat{v}_2(x_0^*, \tau_2^*)$.

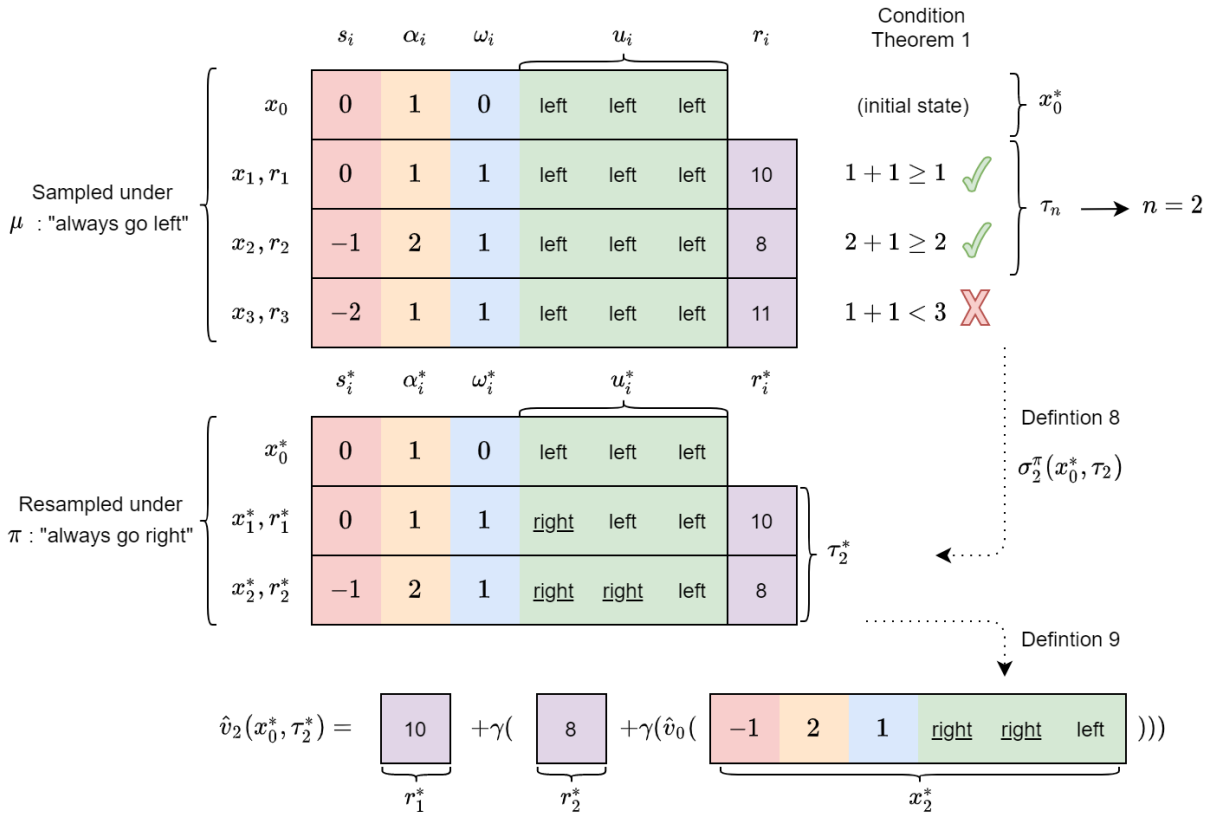

Figure 4.7 Visualization of the procedure in a 1D-world with random delays ($K = 3$).

4.5 Delay-Correcting Actor-Critic

We have seen in Section 4.4 how it is possible, in the delayed setting, to collect *off-policy* trajectories and still use *on-policy* multi-step estimators in an unbiased way, which allows us to compensate for the more difficult credit assignment introduced by the presence of random delays. We now apply this method to derive Delay-Correcting Actor-Critic (DCAC), an improved version of SAC [17, 18] for real-time randomly delayed settings.

4.5.1 Value Approximation

Like SAC, DCAC makes use of the entropy-augmented soft value function [17]:

Definition 10. In a RDMDP (E, p_ω, p_α) the soft value function is:

$$v^{\text{soft}}(x_0^*) = \mathbb{E}_{a\sim\pi(\cdot|x_0^*)}[\mathbb{E}_{x_1^*, r_1^*\sim\tilde{p}(\cdot|x_0^*, a)}[r_1^* + \gamma v^{\text{soft}}(x_1^*)] - \log\pi(a|x_0^*)] \tag{4.20}$$

To estimate this soft value function, we make use of the n-step state-value estimator. Namely, we augment the reward function with the soft entropy reward in Definition 9, which yields:

Definition 11. The delayed on-policy n-step soft state-value estimator, i.e. the n-step state-value estimator with entropy augmented rewards under the current policy π, is

$$\hat{v}_n^{\text{soft}}(x_0^*;\, \tau_n^*) = r_1^* + \gamma \hat{v}_{n-1}^{\text{soft}}(x_1^*; \tau_{n-1}^*) - \mathbb{E}_{a\sim\pi(\cdot|x_0^*)}[\log\pi(a|x_0^*)] \tag{4.21}$$

where $\hat{v}_0^{\text{soft}}$ is a soft state-value function approximator (e.g. a neural network).

Given the off-policy trajectory transformation proposed in Section 4.4, Definition 11 directly gives DCAC's value target. To recap, we sample an initial state x_0 $(= x_0^*)$ and a subsequent trajectory τ_n $(= x_1, r_1, \ldots x_n, r_n)$ from a replay memory. The sampling procedure ensures that n is the largest length such that the sampled trajectory τ_n does not contain any total delay $\omega_i + \beta_i < i$. This trajectory was collected under an *old policy* μ, but we need a trajectory compatible with the *current policy* π to use $\hat{v}_n^{\text{soft}}$ in an unbiased way. Therefore, we feed τ_n to the partial trajectory resampling operator defined in Definition 8. This produces an *equivalent on-policy* sub-trajectory τ_n^* with respect to the current policy π according to Theorem 1, while maximally taking advantage of the bias reduction described by Lemma 4. This partially resampled on-policy sub-trajectory is fed as input to $\hat{v}_n^{\text{soft}}(x_0; \tau_n^*)$, which yields the value target used in DCAC's soft state-value loss:

Definition 12. The DCAC critic loss is

$$L_v^{\text{DCAC}}(v) = \mathbb{E}_{(x_0,\tau_n)\sim\mathcal{D}}\ \mathbb{E}_{\tau_n^*\sim\sigma_n^\pi(\cdot|x_0;\tau_n)}[(v_\theta(x_0) - \hat{v}_n^{\text{soft}}(x_0;\ \tau_n^*))^2] \tag{4.22}$$

where x_0, τ_n are a start state and following trajectory, sampled from the replay memory $\mathcal{D}$, and satisfying the condition of Theorem 1, and θ are the non-target parameters of the model.

Note that, although the condition of Theorem 1 is on the resampled sub-trajectory (x_0^*, τ_n^*), the equivalent on the original sub-trajectory (x_0, τ_n) is trivial. Indeed, the resampling operator does not modify the initial state nor the delays. In other words, $x_0^* = x_0$ and $\alpha_i^*, \omega_i^* = \alpha_i, \omega_i$.

4.5.2 Policy Improvement

In addition to using the on-policy n-step value estimator as target for our parametric value estimator, we can also use it for policy improvement. Similarly to SAC, we use the reparameterization trick [59] to obtain the policy gradient from the value estimator. However, since we use our trajectory transformation and a multi-step value estimator, this involves backpropagation through time in the action-buffer.

Definition 13. The DCAC actor loss is

$$L_\pi^{\text{DCAC}}(\pi) = -\mathbb{E}_{(x_0,\tau_n)\sim\mathcal{D}}\ \mathbb{E}_{\tau_n^*\sim\sigma_n^\pi(\cdot|x_0;\tau_n)}[\hat{v}_n^{\text{soft}}(x_0;\ \tau_n^*)] \tag{4.23}$$

where x_0, τ_n are a start state and following trajectory, sampled from the replay memory $\mathcal{D}$, and satisfying the condition of Theorem 1.

If an unbiased function approximator $\hat{v}_0^{\text{soft}}$ is used, the gradient of the DCAC actor loss is the true policy gradient. In addition, when $\hat{v}_0^{\text{soft}}$ is biased, we can show that the DCAC actor loss is a less biased version of the the SAC actor loss, assuming SAC is using a similarly biased function approximator. By this we mean that SAC is using a soft action-value function approximator $\hat{q}_0^{\text{soft}}$ that has the same bias as the soft state-value function approximator $\hat{v}_0^{\text{soft}}$ used by DCAC. Indeed, whereas DCAC uses the n-step soft state-value estimator described in Definition 11, SAC uses a 1-step soft action-value estimator:

Definition 14. The 1-step soft action-value estimator is defined as

$$\hat{q}_1^{\text{soft}}(x,a;\ x',r') = r' + \gamma\ \mathbb{E}_{a'\sim\pi(\cdot|x')}[\hat{q}_0^{\text{soft}}(x',a') - \log(a'|x')]. \tag{4.24}$$

where $\hat{q}_0^{\text{soft}}$ is a soft action-value function approximator (e.g. a neural network).

As hinted by Lemma 4, this makes SAC more biased than DCAC:

Proposition 1. The DCAC actor loss is a less biased version of the SAC actor loss with

$$\text{bias}(L_\pi^{\text{DCAC}}) = \mathbb{E}_n[\gamma^n] \,\text{bias}(L_\pi^{\text{SAC}}) \tag{4.25}$$

assuming both are using similarly biased parametric value estimators to compute the loss, i.e.

$$\text{bias}(\hat{v}_0^{\text{soft}}(x)) = \mathbb{E}_{a\sim\pi(\cdot|x)}[\text{bias}(\hat{q}_0^{\text{soft}}(x,a))] \tag{4.26}$$

with the simplifying assumption that $\mathcal{D} \sim p_{\text{lim}}^\pi$, the limiting distribution under π.

This assumption on the action buffer $\mathcal{D}$ being sampled from the limiting distribution of the Markov Reward Process is a bit misleading. In particular, it implies that the action buffer is on-policy, although of course in reality it contains off-policy samples. Yet, we needed this assumption to make the proof simple. Another assumption that would have worked would have been that the bias of both $\hat{v}_0$ and $\hat{q}_0$ is constant across the whole input space, but this would have been less realistic.

Lemma on Steady-State Value Estimation Bias

Lemma 5. The expected bias of the n-step value estimator under the steady-state distribution (if it exists) is

$$\mathbb{E}_{x\sim p_{\text{ss}}^\pi}[\text{bias}\,\hat{v}_n(x)] = \gamma^n \mathbb{E}_{x\sim p_{\text{ss}}^\pi}[\text{bias}\,\hat{v}_0(x)] \tag{4.27}$$

Proof. We remind ourselves that the steady state distribution observes

$$p_{\text{ss}}^\pi(x_n) = \mathbb{E}_{x_0\sim p_{\text{ss}}^\pi}[p_n^\pi(..., x_n, r_n|x_0)]. \tag{4.28}$$

According to Lemma 4 we then have

$$\begin{aligned}
\mathbb{E}_{x_0\sim p_{\text{ss}}^\pi} \text{bias}(\hat{v}_n(x_0,\cdot)) =& \gamma^n \mathbb{E}_{...,x_n^*,r_n^*\sim p_n^\pi(\cdot|x_0)}[\text{bias}(\hat{v}_0(x_n^*))] && (4.29)\\
=& \gamma^n \mathbb{E}_{x\sim p_{\text{ss}}^\pi}[\text{bias}\,\hat{v}_0(x)]. && (4.30)
\end{aligned}$$

□

Proof of Proposition 1

For simplicity, we assume that the states in the replay memory are distributed according to the steady-state distribution, i.e. $D \sim p^{\pi}_{\text{ss}}$. This assumption could be avoided by making more complicated assumptions about the biases of the state-value and action-value estimators.

We now start with the bias of the DCAC loss w.r.t to an unbiased SAC loss using the true action-value function,

$$\text{bias}(L^{\text{DCAC}}_{\pi}) = L^{\text{DCAC}}_{\pi} - L^{\text{UB}}_{\pi} \tag{4.31}$$

where

$$\begin{aligned} L^{\text{DCAC}}_{\pi} &= -\,\mathbb{E}_{x_0,\tau_n\sim\mathcal{D}}\;\mathbb{E}_{\tau^*_n\sim\sigma^{\pi}_n(\cdot|x_0;\tau_n)}[\hat{v}^{\text{soft}}_n(x_0;\;\tau^*_n)] && (4.32)\\ &= -\,\mathbb{E}_{x_0\sim D}\mathbb{E}_n\mathbb{E}_{\tau^*_n\sim p^{\pi}_n(\cdot|x_0)}[\hat{v}^{\text{soft}}_n(x_0;\;\tau^*_n)] \quad \mid \text{ Theorem 1} && (4.33)\end{aligned}$$

and

$$\begin{aligned} L^{\text{UB}}_{\pi} &= \mathbb{E}_{x_0\sim\mathcal{D}}[\mathbb{E}_{a\sim\pi(\cdot|x_0)}[\log\pi(a|x_0) - q^{\text{soft}}(x_0,a)]] && (4.34)\\ &= -\,\mathbb{E}_{x_0\sim\mathcal{D}}[v^{\text{soft}}(x_0)]. && (4.35)\end{aligned}$$

Substituting these we have

$$\begin{aligned} \text{bias}(L^{\text{DCAC}}_{\pi}) &= \mathbb{E}_{x_0\sim D}\mathbb{E}_n\mathbb{E}_{\tau^*_n\sim p^{\pi}_n(\cdot|x_0)}[\hat{v}^{\text{soft}}_n(x_0;\;\tau^*_n) - v^{\text{soft}}(x_0)]\\ &= \mathbb{E}_{x_0\sim D}\mathbb{E}_n\mathbb{E}_{\tau^*_n\sim p^{\pi}_n(\cdot|x_0)}[\text{bias}(\hat{v}^{\text{soft}}_n(x_0;\tau^*_n)]\\ &= \mathbb{E}_{x_0\sim D}\mathbb{E}_n[\gamma^n\mathbb{E}_{\ldots,x^*_n,r^*_n\sim p^{\pi}_n(\cdot|x_0)}[\text{bias}(\hat{v}^{\text{soft}}_0(x^*_n))]] \quad \mid \text{ Lemma 4}\\ &= \mathbb{E}_n[\gamma^n]\;\mathbb{E}_{x\sim D}[\text{bias}(\hat{v}^{\text{soft}}_0(x))] \quad \mid \text{ using } D\sim p^{\pi}_{\text{ss}} \text{ and Lemma 5}\\ &= \mathbb{E}_n[\gamma^n]\;\mathbb{E}_{x\sim D}[\mathbb{E}_{a\sim\pi(\cdot|x)}[\text{bias}(\hat{q}^{\text{soft}}_0(x,a))]] \quad \mid \text{ Equation 4.26}\\ &= \mathbb{E}_n[\gamma^n]\;\mathbb{E}_{x\sim D}[\mathbb{E}_{a\sim\pi(\cdot|x)}[\hat{q}^{\text{soft}}_0(x,a) - q^{\text{soft}}(x,a)]]\\ &= \mathbb{E}_n[\gamma^n]\;(L^{\text{SAC}}_{\pi} - L^{\text{UB}}_{\pi})\\ &= \mathbb{E}_n[\gamma^n]\,\text{bias}(L^{\text{SAC}}_{\pi})\end{aligned}$$

□

4.6 Experimental study

To evaluate our approach and make future work in this direction easy for the RL community, we release as open-source, along with our code, a Gym wrapper that introduces custom multi-step delays in any classical turn-based Gym environment. When using this wrapper, the wrapped turn-based environment effectively becomes the undelayed environment described in Figure 4.1, and randomly delayed dynamics are artificially simulated. This is useful because this allows to use common Gym benchmarks to evaluate algorithms in the more realistic randomly delayed setting. In particular, this enables us to introduce random delays to the Gym MuJoCo continuous control suite [10, 60], which is otherwise turn-based (Figure 4.8).

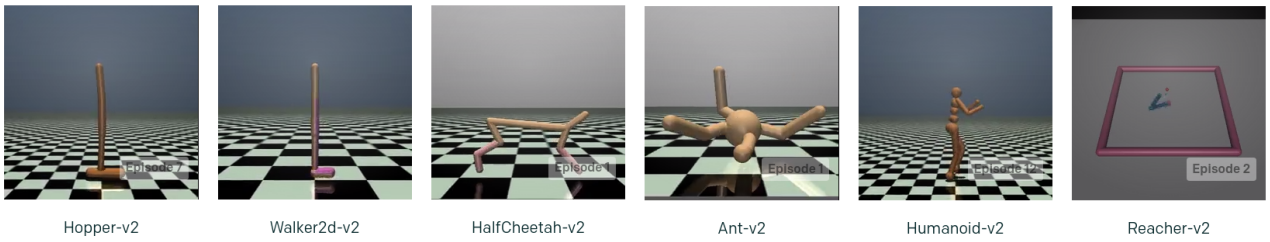

Figure 4.8 Gym MuJoCo tasks augmented with random delays in our experiments

4.6.1 Compared algorithms

As previously pointed out, in delayed scenarios, it would be naive to only use the unaugmented delayed observations, as this would violate the Markov assumption. For illustration, we provide a few experiments where SAC is using only these observations, as it would normally do in the Gym MuJoCo tasks. However, we have artificially introduced a constant single-step delay to these tasks. The naive version of SAC is compared to the same algorithm, only augmenting the observations with an action buffer of 1 action as described in Section 4.3.1:

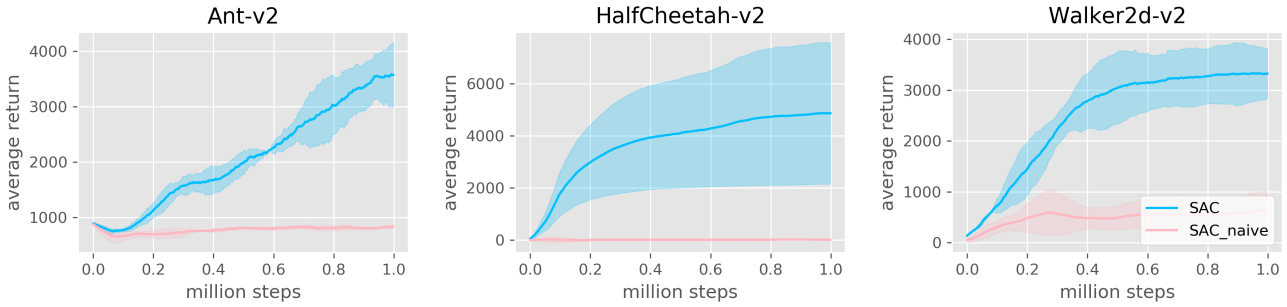

Figure 4.9 Importance of the augmented observation space

In Figure 4.9, the action delay is $\alpha = 1$ (e.g. the time needed for inference in a neural network) whereas the observation delay is $\omega = 0$ (no observation delay). This sums to a constant 1-step total delay, which is our simplest possible delayed task. Yet, this task is already enough to illustrate the importance of the augmented observation space in delayed scenarios. Even with this small 1-step constant delay, the unaugmented observations are not Markov and a naive algorithm using only these observations (here: SAC naive) exhibits near-random results. By comparison, an algorithm using the *RDMDP* augmented observations instead (here: SAC) is able to learn in delayed environments.

In order to make a fair comparison, all our other experiments compare DCAC against SAC in the same RDMDP setting. That is to say, all algorithms use the augmented observation space as defined in Section 4.3.1, as the non-naive version of SAC does in Figure 4.9. Since SAC is the algorithm we chose to improve for delayed scenarios, comparing DCAC against SAC in the same setting provides a like-for-like comparison. We also found interesting to compare DCAC against RTAC [46]. Indeed, DCAC reduces to this algorithm in the special case where observation transmission is instantaneous (ω=0) and action computation and transmission constantly takes 1 time-step (α=1). Whereas DCAC performs variable-length state-value backups with partial trajectory resampling as explained in Section 4.5 , RTAC performs 1-step state-value backups, and SAC performs the usual 1-step action-value backup described in its second version [18].

4.6.2 Implementation details

The model architecture we use is the same in all our experiments and across all tested algorithms. It is composed of two separate multi-layer perceptrons (MLPs): a critic network, and an actor network. Both MLPs are built with the same simple architecture of 2 hidden layers, 256 units each. The critic outputs a single value, whereas the actor outputs an action distribution with the dimension of the action space, from which actions are sampled with the reparameterization trick. This architecture is compatible with the second version of SAC described in [18]. The only difference between models is that in SAC the critic tracks $q(x)$, whereas in DCAC it tracks $v(x)$. Indeed, differently from usual actor-critic algorithms, the output of DCAC's critic approximates the state-value $v(x)$ (instead of the action-value $q(x)$), as it is sufficient to optimize the DCAC actor loss described in Definition 13. We initialize all weights and biases of the MLPs with the default Pytorch initializer. Both the actor and the critic are optimized by gradient descent with the Adam optimizer. For DCAC, the optimized losses are $L^{\text{DCAC}}(\pi)$ (equation 4.23) and $L^{\text{DCAC}}(v)$ (equation 4.22), respectively. Classically, we use twin critic networks [61, 62] with target weight tracking [5] to stabilize training.

Other than their neural network architecture, our implementations of SAC, RTAC and DCAC all share the hyperparameters featured in Table 4.1.

Table 4.1 Hyperparameters

Name	Value
Optimizer	Adam [63]
Learning rate	0.0003
Discount factor (γ)	0.99
Target weights update coefficient (τ)	0.005
Gradient steps / environment steps	1
Reward scale	5.0
Entropy scale	1.0
Replay memory size	1000000
Number of samples before training starts	10000
Number of critics	2

The 'reward scale' is a multiplier of the reward term in Definition 11, and the 'entropy scale' is a multiplier of the logarithm in the same definition. These scales are described in [17], we omitted them in our study for simplicity because they don't play any role in our analysis (but of course they play the same role as in SAC for balancing pure rewards and entropy).

Similarly to [5], the target parameters are updated according to the following running average:

$$\overline{\theta} \leftarrow \tau\theta + (1-\tau)\overline{\theta}$$

where $\overline{\theta}$ are the parameters of the slowly-moving model used to estimate value targets in the critic loss whereas θ are the parameters of the current model.

In the constantly delayed setting, the delays ω and α are fixed, and can be implicitly considered as a constant property of the environment. In this setting, we simply concatenate the members s and u of the augmented observation $x = s, u, \omega, \alpha$, and feed the concatenation to the model.

In the more general case of $RDMDP(E, p_\alpha, p_\omega)$, we use the full information available. As a matter of fact, we even use slightly more information than what we described in Section 4.3.1. Namely, we use a more informative action delay as input to the model, effectively replacing α for the sole purpose of action inference. Remember that the action delay α is needed to define RDMDPs, and is used by DCAC as part of the total delay $\alpha + \omega$. It is also useful as an input to the model, because it identifies the last action that influenced the observation, and therefore gives information about what actions may be applied next. However, in RDMDPs, this information is incomplete. We do even better by appending another kind of action delay

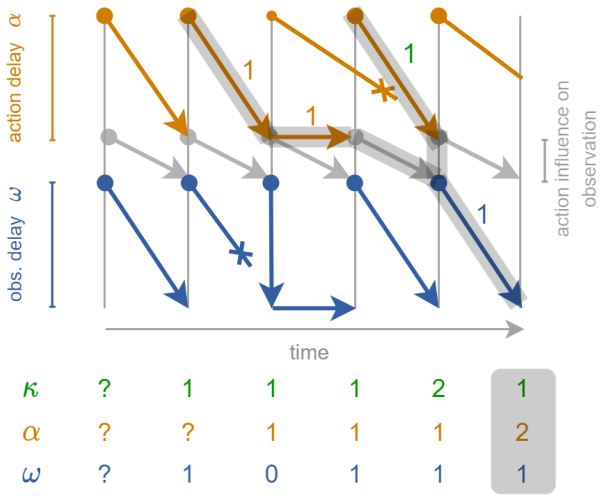

Figure 4.10 A better action delay for inference: κ

to the observations on top of α, which we call κ. Whereas α identifies the action that was applied in the undelayed environment during the time-step prior to the end of observation capture, κ identifies the action that was applied next. This is possible in practice by appending κ to the observation at the level of the remote robot right before sending this observation to the agent. We use κ instead of α as input of the model for all tested algorithms.

DCAC requires that the $\omega_i + \alpha_i$ information is available for each delayed observation s_i. This is achieved by appending an identifier to the action computed by the agent. The identifier of the action that was applied during the time-step prior to the end of observation capture is then sent back to the agent, along with the observation. When the agent receives the identifier back, it looks the identified action up in the action-buffer. This action is found at the $(\omega_i + \alpha_i)^{th}$ position, as it was sent $\omega_i + \alpha_i$ time-steps ago. The action identifier can always be the timestamp at which the agent started computing the action. Making the additional assumption that the remote robot (i.e. the undelayed environment) and the agent have synchronized clocks, the robot can directly append to the observation the local timestamp corresponding to the end of observation capture. This timestamp can then be used by the agent to deduce ω_i, κ_i and α_i separately.

Although this is not required by the training algorithm (which only needs the $\omega_i + \alpha_i$ information), this is theoretically required as input to the model for inference since it is part of the state space. It adds the information 'how old is the delayed observation' (ω_i) to the information 'how old is the action that influenced it' ($\omega_i + \alpha_i$). Thus we feed the concatenation of x_i, ω_i and κ_i as input of our model for all tested algorithms. More exactly, we feed ω_i and κ_i as one-hot encoded inputs, since we believe this maps to positions in the action-buffer (i.e.

to past moments in time) in a more natural way than feeding their raw integer values.

For DCAC to work, the minimal information required is the minimum possible total delay (in this situation, one would always partially resample trajectories for up to this number of time-steps). It is then easy to adapt the aforementioned details related to delays depending on the level of control the practitioner has on the remote robot. For instance, if the only thing that is easy to do is to pass the identifier of the action that was applied prior to the end of observation capture along with the observation, one can easily compute the total delay at the level of the agent. The total delay would be fed to the model instead of the separate delays, since the information on separate delays would not be available. Keep in mind that not feeding the separate delays turns the problem into a Partially Observable MDP, though.

As the reader may have noticed, a difficulty arises when implementing DCAC with minibatch tensors. Namely, with random delays, the length of the performed multi-step backup is different for each individual trajectory sampled from the replay memory. In our implementation, we alleviate this by sampling trajectories of length K. We then compute a mask in a batched fashion, representing for each trajectory the longest sub-trajectory that satisfies the condition of Theorem 1. This mask is then used to resample the action-buffers and perform the multi-step backup in a batched fashion. For more details, we refer the reader to our code.

All the details that have been mentioned in this section are implemented for all tested algorithms (SAC, RTAC and DCAC), as they are related to RDMDP s rather than DCAC in particular.

4.6.3 Empirical results

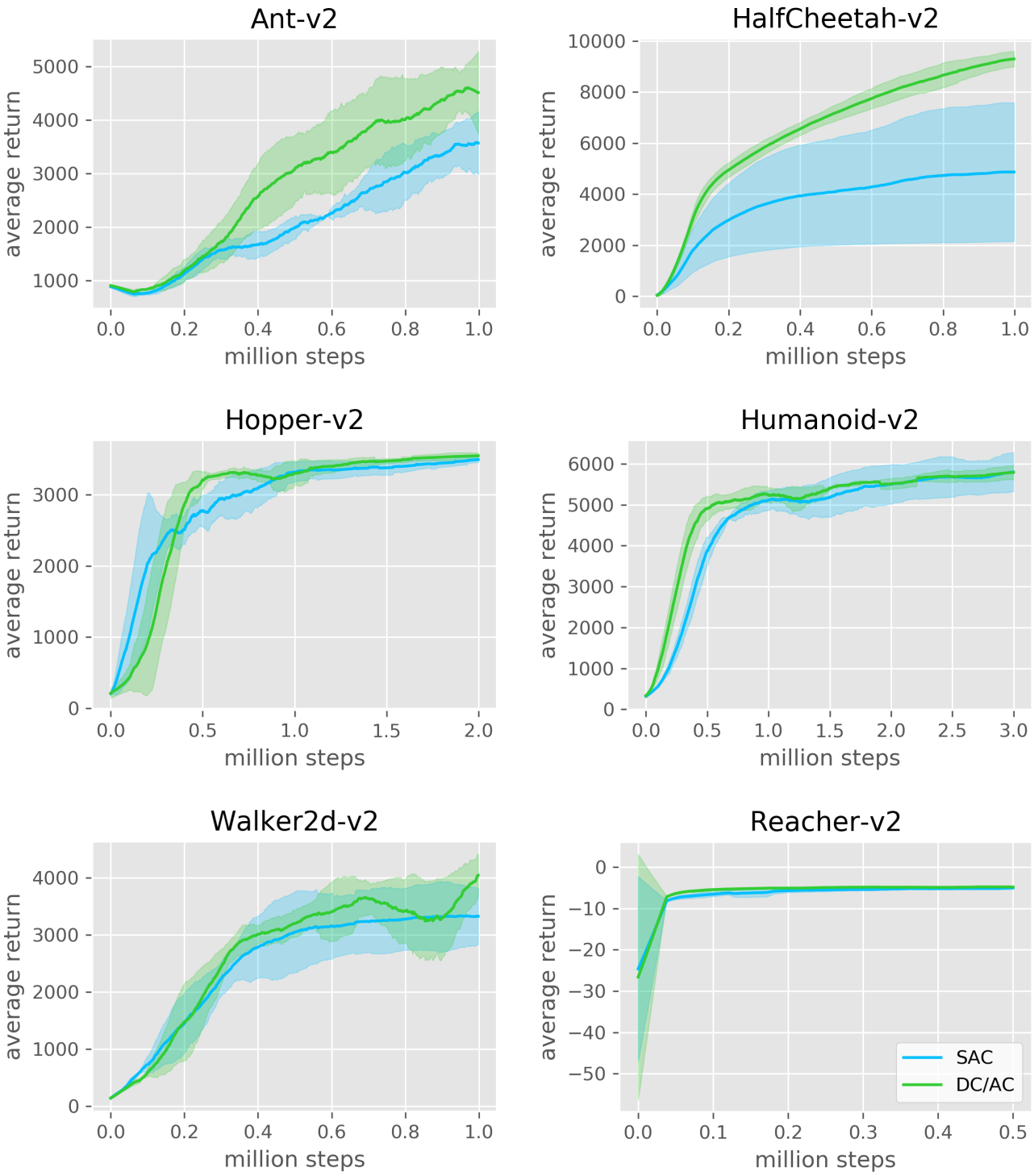

Figure 4.11 Constant 1-step delays

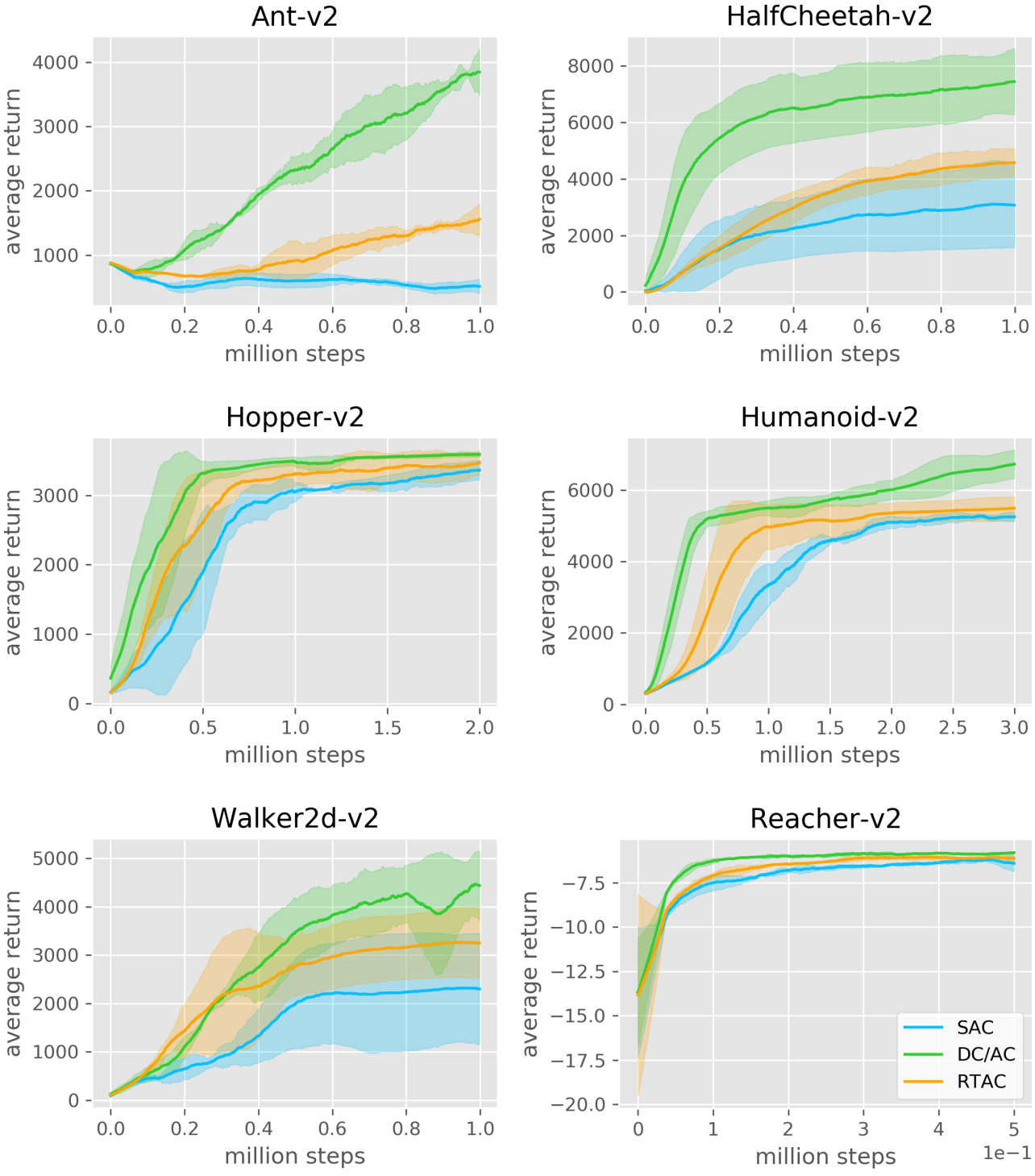

Figure 4.12 Constant 3-step delays

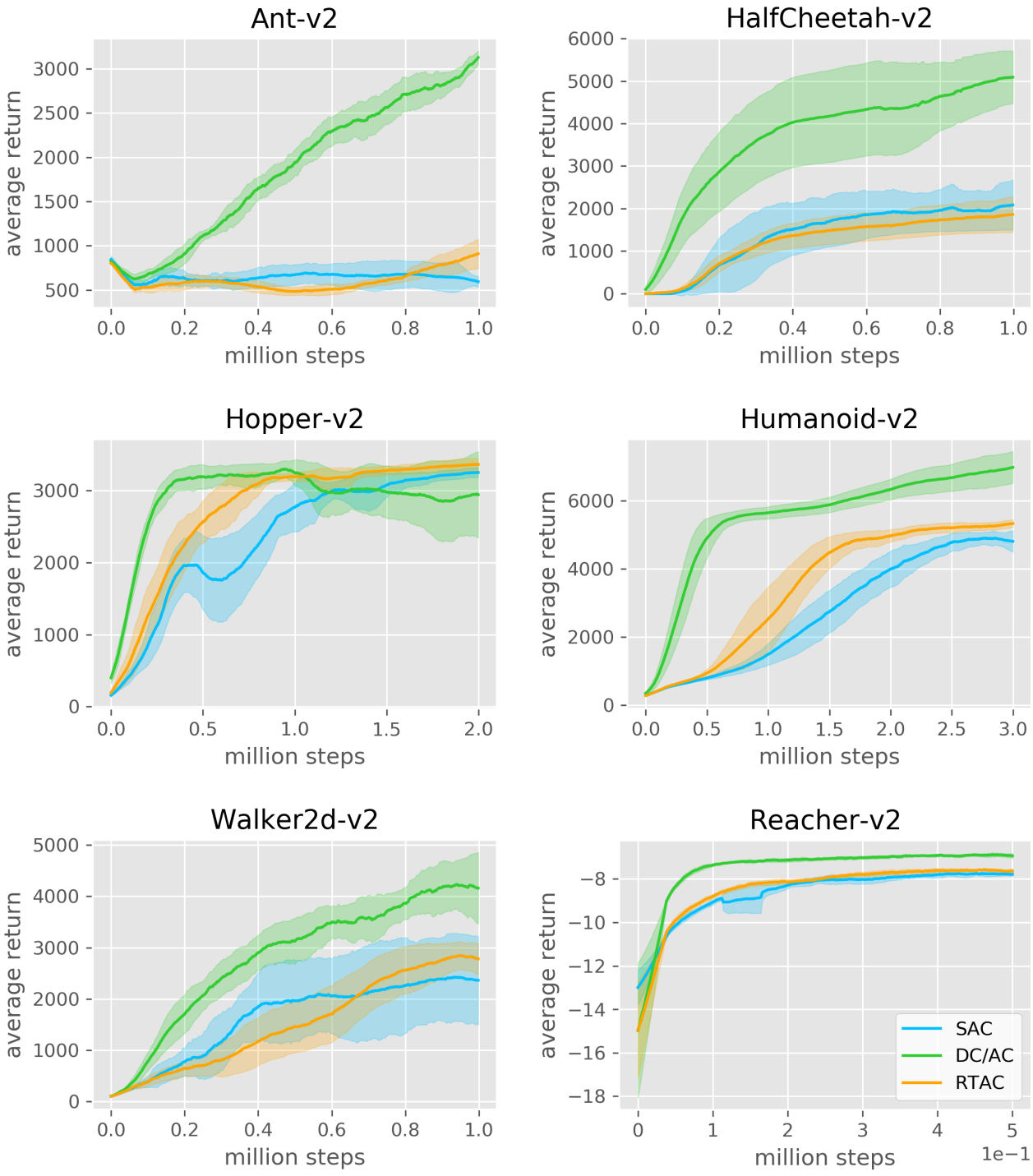

Figure 4.13 Constant 5-step delays

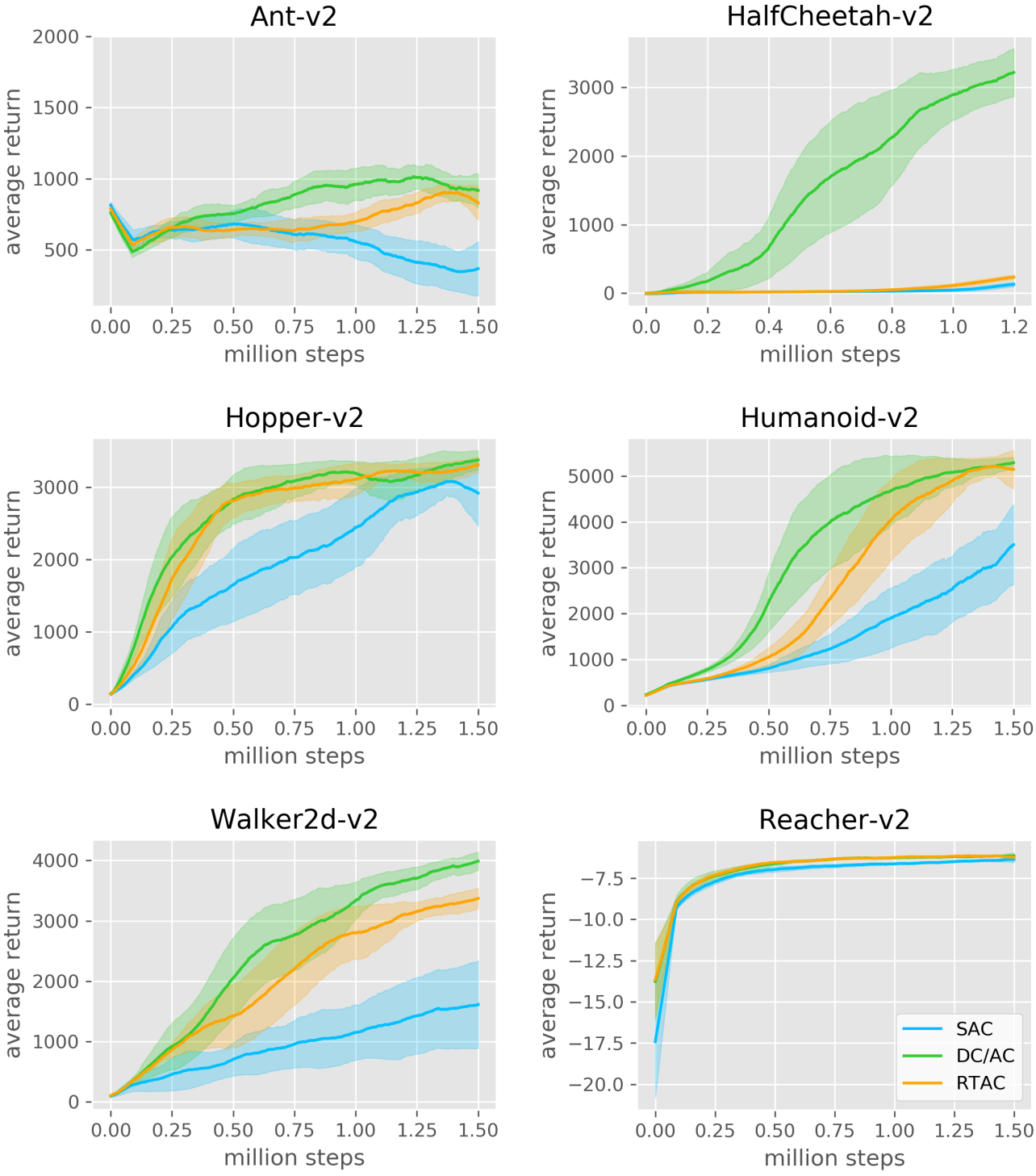

Figure 4.14 Uniformly sampled random delays

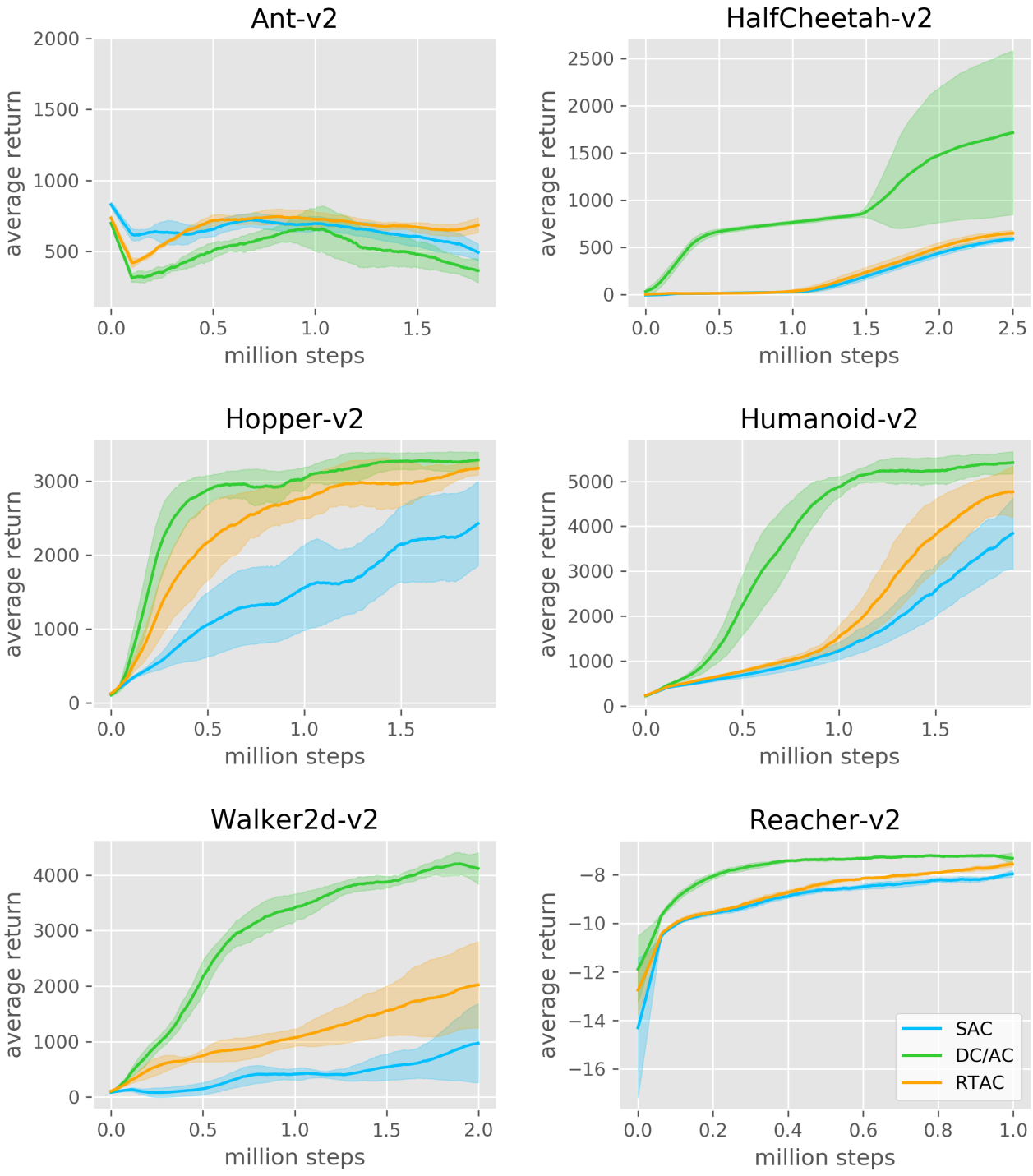

Figure 4.15 Real-world WiFi delays

For each experiment presented in this section, we performed six runs with different seeds. We shade the 90% confidence intervals in our plots.

Constant delays

Our first batch of experiments features simple, constantly delayed scenarios. Figure 4.11 features our simplest experiment (i.e. with the shortest delays), where the action delay is constantly $\alpha = 1$ and observation delay is constantly $\omega = 0$. In this simple setting, DCAC reduces to the vanilla RTAC [46] algorithm (without output normalization and merged networks). DCAC (RTAC) slightly outperforms SAC when the total delay is 1.

The setting featured in Figure 4.12 is slightly more difficult. Here, the action delay is constantly $\alpha = 2$, and the observation delay is constantly $\omega = 1$, summing to a total constant delay of 3 time-steps instead of 1. DCAC starts really showing its potential, clearly outperforming all other approaches as the delay grows larger. Finally, Figure 4.13 features the most difficult of our batch of constantly delayed experiments. Here, the action delay is constantly $\alpha = 3$, and the observation delay is constantly $\omega = 2$, summing to a constant total delay of 5 time-steps. DCAC now exhibits a very strong advantage in performance, as all other approaches struggle in the presence of long constant delays. Since all tested algorithms use the same RDMDP augmented observations, the advantage of DCAC comes only from the unbiased and low-variance n-step backups made possible by the partial resampling procedure. In these simple constantly delayed experiments, the condition of Theorem 4.6.2 is always met for the first K time-steps. Therefore, when delays are constant, the backup length is always $n = K$ (except when a terminal state is met in the trajectory fragment). The advantage of using DCAC is obvious in the presence of long constant delays. Note that DCAC reduces to the RTAC [46] algorithm when $\omega = 0$ and $\alpha = 1$, and behaves as an evolved form of RTAC in the presence of longer constant delays. We found the comparison with RTAC particularly insightful in these experiments, as it allows us to see the importance of the backup length with respect to the length of the delays. We observe that the credit assignment difficulty is efficiently compensated by DCAC over the full length of the resampled trajectory, and not only over the few first steps as would have been the case if RTAC were not greatly outperformed.

Random delays

Our second batch of experiments features random delays of different magnitudes, sampled from different distributions.

In Figure 4.14, the communication delays are sampled from uniform distributions. More

precisely, we consider that it takes a maximum of 1 time-step for the model to infer an action. Then, the communication delay for this action is uniformly sampled from $[0; 2]$, which sums to an action delay sampled from $[1; 3]$ on the one hand. On the other hand, the transmission delay for the observation is also uniformly sampled from $[0; 2]$.

Note that, although the transmission delays are sampled from uniform distribution, the actual α_i and ω_i seen by the agent still depend on recursive and interleaved dynamics as described by Definition 5, which forms more complicated distributions. When action or observation communications supersede previous communications, only the most recently produced information is kept. In other words, when an action is received in the undelayed environment, its age is compared to the action that is currently being applied. Then, the one that the agent most recently started to produce is applied. Similarly, when the agent receives a new observation, it only keeps the one that was most recently captured in the undelayed environment (see the right-hand side of Figure 4.5 for a visual example).

Since an equal probability is given to all possible transmission delays in the specified ranges, the action and observation delays have a strong variance which makes this experiment fairly difficult. However, this also means that the delays are smaller than in e.g. the previously presented constant 5-step delay experiment, although the maximum delay is also 5 here. This is why the advantage of DCAC is not as obvious over RTAC, the total delay being closer to 1. On the other hand, the fact that delays are random clearly makes the tasks much more difficult, despite the mean delay not being very large. Indeed, the agent has no way of predicting when the action it computes is going to be applied, and must therefore take stochastic chains of decisions. In particular, delayed Ant has become too difficult for all tested algorithms. For other tasks, the advantage of DCAC over SAC is very clear.

Finally, the last experiment we present in this chapter is motivated by the fact that our approach is designed for real-world applications.

In the experiment featured in Figure 4.15, we sample the communication delays for actions and observations from our real-world WiFi dataset, previously presented in Figure 4.2.

In our future work, we desire to control a drone at 50Hz. Thus, we discretize the communication delays by using a time-step of 20ms. Importantly, note that Figure 4.2 has been cropped to 60ms, but the actual dataset contains outliers that can go as far as 1s. However, long delays (longer than 80ms in our example) are almost always superseded and discarded. Therefore, when such information is received, we clip the corresponding delay with no visible impact in performance. In practice, the maximum acceptable delays are design choices since the actual maximum delays can be prohibitively long and would require a long action buffer to be handled in the worst-case scenario. Observations reaching the agent with a total delay

that exceeds the chosen K value should simply be discarded, and a procedure implemented to handle the unlikely edge-case where more than K such observations are received in a row. These choices can be guided by existing probabilistic timing methods [64]. Similarly to the previous experiment, we consider that action inference takes a maximum of 1 time-step, and superseded actions or observations are discarded. In a nutshell, the maximum possible delay here is $K = 9$ time-steps, with random actions and observation transmission delays sampled from a real-world WiFi distribution.

Unsurprisingly, Delayed Ant is still too difficult for all approaches in this scenario, as random delays in Figure 4.15 are larger than in Figure 4.14. On other tasks, DCAC clearly dominates the baselines. Interestingly, Delayed HalfCheetah became difficult too and all approaches struggle in local minima, from which only DCAC successfully escapes.

4.6.4 Other practical considerations

The way we chose to define RDMDPs, the agent only keeps the delayed observation that was most recently captured in the undelayed environment. Ideally, it is also ensured by the undelayed environment that the applied action is the action that most recently started being computed by the agent. In practice, this can be ensured by augmenting the actions with time-stamps corresponding to the beginning of their computation, and observations with time-stamps corresponding to the end of their capture. Thus, the undelayed environment and the agent can keep track of the most recent received time-stamp and discard outdated incoming information.

The separation between ω and α is mainly required to allow auto-correlated conditional distributions on both delays, so for example superseded actions and observations can be discarded. In such situations, it is theoretically required to observe ω and α separately. However this may not always be possible in practice. Note that DCAC doesn't make use of the separate delays, and instead simply looks at the total delays $(\omega_i + \alpha_i)$. Therefore, when only the total delays are available (they can easily be measured by the agent), they can still be used by DCAC and observed by the model.

The action buffer (u) is required to define the mathematical framework as a Markov Decision Process. When a small time-step is chosen, the size of the action-buffer grows large. One possibility to practically handle such situation if desired is to compress the action buffer in the hidden state of an RNN, so the model observes it in an approximate manner.

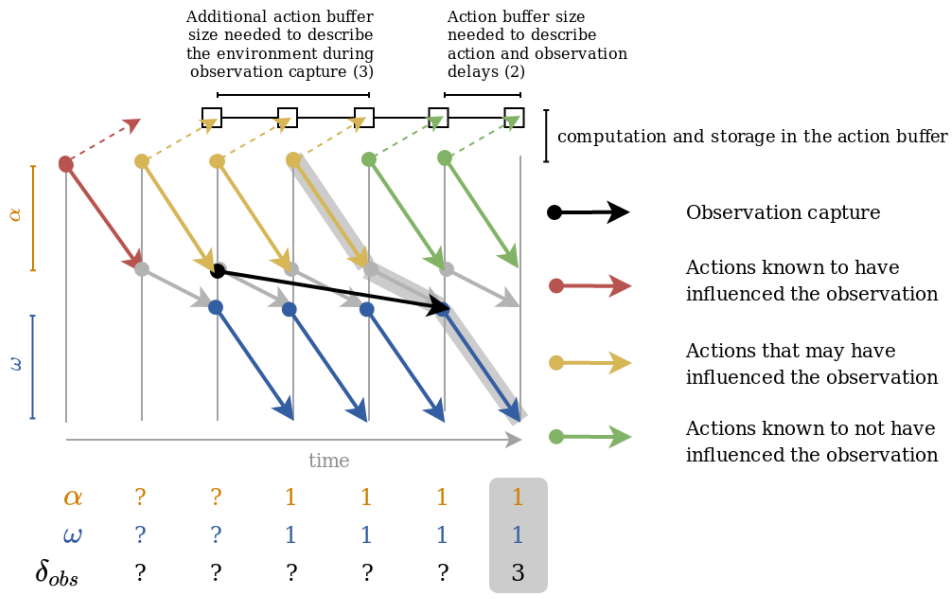

Figure 4.16 Long observation capture

In this chapter, we have implicitly assumed that observation capture is instantaneous whereas processing and transmission can be part of the observation delay. However, in practice, it is often the case that observation capture cannot be considered instantaneous. In such situations, the practitioner should increase the size of the action buffer by a number of time-steps corresponding to the maximum duration of observation capture, and append the observation capture duration δ_{obs} to the observation when it is available and not constant. Indeed, as illustrated in Figure 4.16, when observation capture is not instantaneous, it is not necessarily possible to know which undelayed state(s) the observation describes. The length of the multi-step backup performed by DCAC is not impacted by this change, because it only cares about the first action that is known not to have influenced the delayed observation. The same holds when observations are formed of several combined parts that were captured at different timestamps. In such situation, one can consider the whole span of these timestamps as the observation capture duration. Additionally, these timestamps can be appended to the formed observation when they are available.

4.7 Impact

Being fairly general, DCAC can be used in most RL situations where delays exist. It deals with delays as small as 1 time-step, and is most relevant in the presence of long delays. This includes high-frequency remote control of flying drones, low-frequency control of distant space robots, high-frequency control of real-time video games, high-speed trading, etc.

CHAPTER 5 A REAL-TIME FRAMEWORK FOR REAL-WORLD APPLICATIONS

In the previous chapter, we have provided a study of how RL theory can be adapted in order to perform well in the presence of real-world random delays. This study was strongly motivated by the perspective of real-world applications in various domains where time cannot be ignored. Examples of such domains are continuous controllers for robotics, real-time video games, and more generally all applications where either inference or communications are slow with respect to the properties of the controlled system. However, we only tested the developed approach in an artificial setting for convenience so far. Indeed, the Gym wrapper we used was merely a tool to simulate random delays in classical turn-based tasks. Time was still paused during action inference and communications in practice, although the time these operations would have taken in the real-world was properly simulated.

In this final chapter we extend the aforementioned approach to actual real-world systems were time cannot be paused. In particular, we release an overlay of OpenAI Gym that enables easy implementation of proper RDMDPs in the real-world. We use this overlay to implement real-time Gym environments in applications such as real flying quadrotors and real-time autonomous driving in car racing video games.

5.1 Real-Time Gym

We introduce Real-Time Gym, a simple framework that enables easy implementation of RDMDPs in the real world. Real-Time Gym is published on PyPI under the name `rtgym`, along with a complete tutorial.

5.1.1 Overview

The purpose of `rtgym` is twofold. First, it makes it easy to design RL environments in the real-world by abstracting the handling of real-time delays from the designer. More precisely, `rtgym` automatically enforces elastically constant time-steps in parallel to action inference and observation retrieval. It also appends the action buffer to observations (this is optional, the designer may chose to implement another compatible representation such as an RNN if they want to). Second, the environments produced when using `rtgym` are compatible with the OpenAI Gym interface. Hence, from the user's point of view, an `rtgym` real-world environment works exactly like usual turn-based Gym environments do. In other words, RL

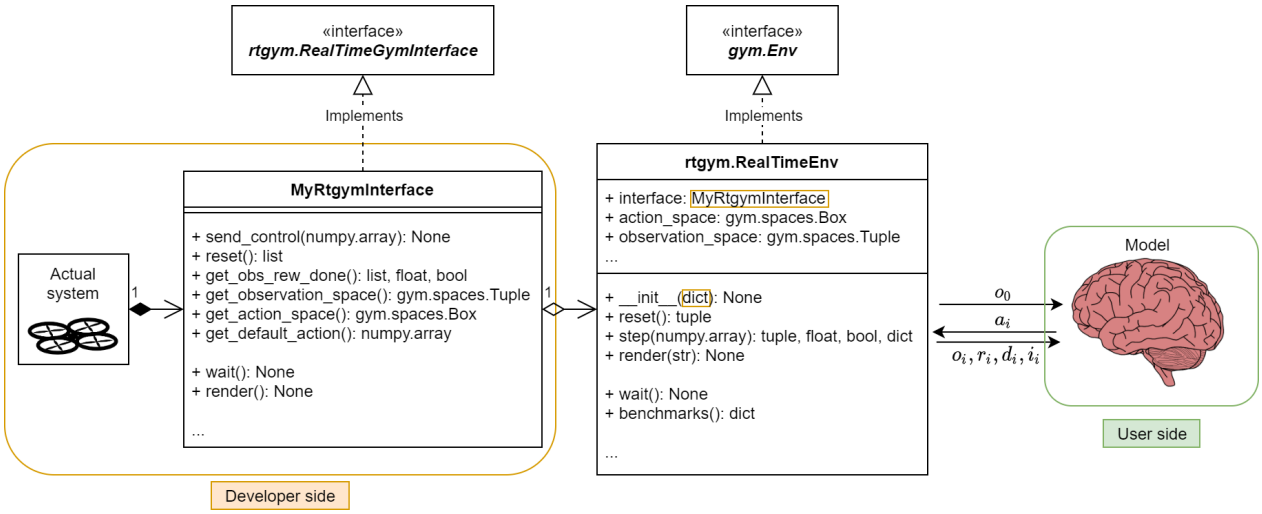

Figure 5.1 High-level architecture or Real-Time Gym

algorithms implemented to work with classical turn-based Gym tasks will work with `rtgym` tasks as well (with one precaution though: the user must call the `env.wait` method if they need to 'stop' the environment).

5.1.2 Principle

From a designer's point of view, Real-Time Gym consists of a minimal abstract python interface and a simple configuration dictionary. An UML diagram describing the software architecture from a high-level standpoint is provided in Figure 5.1.

For a custom task, six abstract methods must be implemented:

- `get_observation_space` defines the shape of the observations that the environment outputs.
- `get_action_space` defines the shape of the actions that the environment takes as input.
- `get_default_action` defines a default action in order to initialize the action buffer.
- `reset` defines what happens when `env.reset` is called and outputs the initial observation.
- `get_obs_rew_done` defines the output of `env.step` .
- `send_control` defines what `env.step` does with its input.

Two additional methods may be overridden if desired:

- `wait` will be called if the user calls `env.wait` .
- `render` will be called if the user calls `env.render` .

From a user's perspective, two optional methods are added to the default OpenAI Gym API: `env.benchmarks` provides useful timing benchmarks, and `env.wait` is used if one wants to 'stop' the `rtgym` environment, which would otherwise have to run continuously.

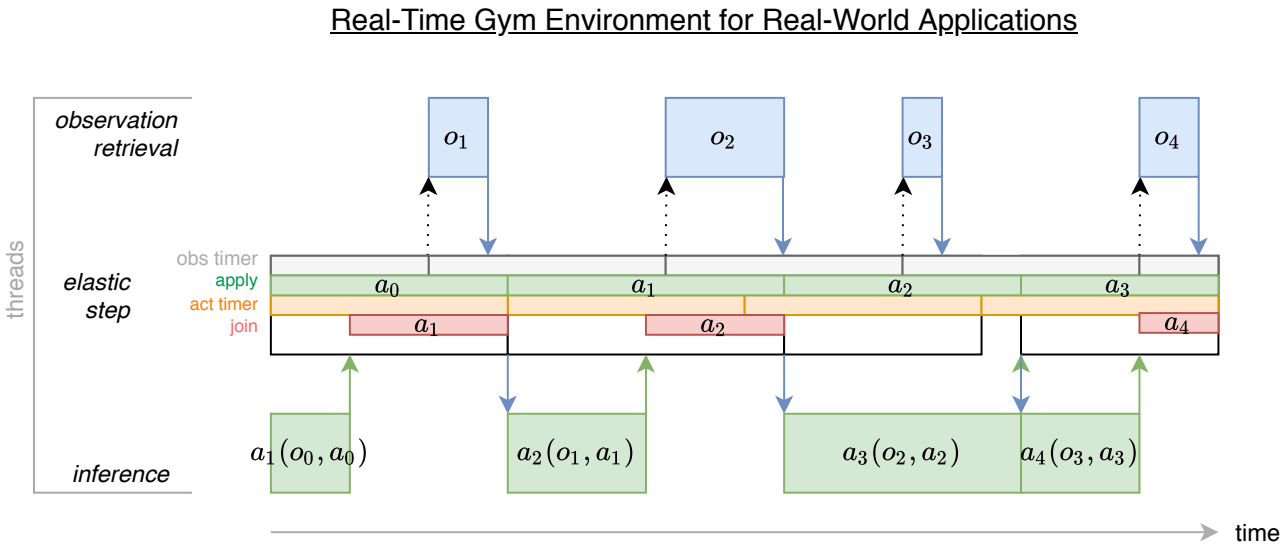

Figure 5.2 Elastic time-steps in Real-Time Gym

Indeed, same as with usual Gym environments, in `rtgym` the user gets an initial observation from `env.reset`, computes an initial action from this initial observation, and passes this action as argument to the `env.step` method which answers with a new observation. Then the user computes a new action from this observation, passes it to `env.step`, and so on. But of course, for this scheme to work in the real world, many things happen in parallel in `env.step` that are hidden from the user. In particular, computing an action from an observation is not instantaneous, and retrieving an observation in the environment is not instantaneous either.

Figure 5.2 illustrates how these issues are dealt with in `rtgym`. The user computes the action a_1 from the previous observation o_0, and an action buffer containing at least the last computed action a_0. Indeed, this action is now either currently being applied in the undelayed environment (if the action delay is 1) or it is travelling toward the undelayed environment (if the action delay is more than 1). Therefore, at least a_0 is required along with o_0 to statistically describe the state at which a_1 is going to be applied. Once a_1 is computed, the user calls `env.step`(a_1). This call is blocking until the end of the current time-step, which is defined by an action timer. By the time the action timer ends, an observation timer clocked the beginning of observation retrieval. If the new observation is available at the end of the action timer (in Figure 5.2 this is the case for o_1), `env.step` returns this observation. Otherwise (e.g. o_2 in Figure 5.2), `env.step` blocks until this observation is available. In its default real-time behavior, `rtgym` only sends the action a_1 at this point, internally using `send_control` to do so. This is because the default mode is built with the underlying assumption that the user

wants to control the device as fast as possible. In this situation, the time-step duration is set to be close to the mean of inference duration, and the time represented in red in Figure 5.2 is small. Conversely, this allows us to enforce healthy (i.e. elastically constant) time-steps, regardless of inference duration.

More precisely, action inference and observation retrieval are performed in parallel and expected to take less than 1 time-step (if observation capture takes more than 1 time-step in average then it must be performed outside of what we call 'observation retrieval' here). Yet, since these durations are random, `rtgym` only requires their mean to be less than 1 time-step. Individually, these durations are allowed to overflow. When they do, `rtgym` allows a certain elasticity that the user defines in terms of number (or portion) of time-steps. As long as the maintained elasticity does not overflow from this number of time-steps, `rtgym` will compensate for the fact that previous time-steps ended after they were expected to by shrinking the next time-steps. If this elasticity stretches more than its allowed range, the user is simply warned and the elasticity is broken. In such situation, the elasticity is directly reset to track the overflow of the current time-step. This is suitable because some occasional operations performed for example in `reset` may break this elasticity (purposefully or not) with no serious impact.

5.1.3 Examples of real-time environments

We demonstrate `rtgym` on a real flying quadrotor, and on several real-time autonomous racing tasks performed from pixels in the untouched (i.e. neither paused nor accelerated) Trackmania [27] video game.

Cognifly in air-balloon mode

We have implemented a simple pilot `rtgym` task on the Cognifly 3D-printed quadrotor [65]. This task, illustrated in Figure 5.3, is conceptually similar to the dummy RC drone task that we provide in our tutorial for `rtgym.` We tether the drone and power it thanks to a computer power supply, so that it can fly with no battery issues. We have modified the drone's controller so the only thing that can be controlled is the velocity on the Z axis. For instance, when the velocity control is 0.0, the drone simply 'floats' as a constant altitude. When pushed on the XY plane, the drone dampens the caused drift on this plane, somehow like an air-balloon. The drone has a laser sensor which reads its altitude. We use a small history of these readings so the model can infer the drone's dynamics. We also use an action buffer similar to what we used in our WiFi experiments in the previous chapter, as the drone is communicating with the decision-making computer by UDP on a local WiFi network. The

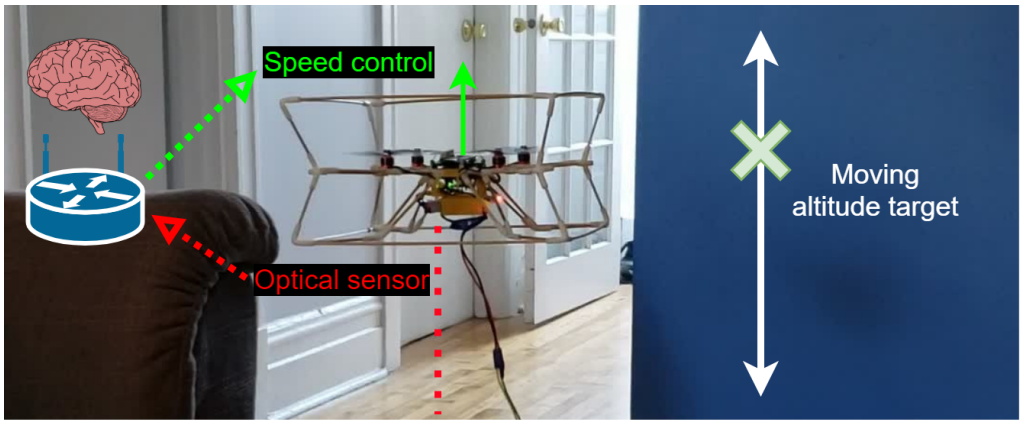

Figure 5.3 Cognifly simple benchmark task

task is to track an altitude target that is moving away from the drone. The only difficulty of this task comes from the random observation and action delays which make it hard to precisely reach the target. Therefore, it can be used as a very simple real-world benchmark for randomly delayed RL.

Real-time autonomous racing from pixels in Trackmania

We have implemented a more advanced series of environments in the Trackmania Nation Forever and Trackmania 2020 video games [27], which we present here to demonstrate `rtgym` in real-time applications.

These tasks consist of autonomous 'racing against the clock' from real-time screenshots. Our environments capture the screen at a frequency of 20Hz, which we define by configuring `rtgym` with a time-step of 0.05s. These images are output by `env.reset` and `env.step`, while actions sent as arguments to `env.step` directly control either the keyboard or a virtual gamepad. Some of these environments preprocess the captured images in order to compute a pixel-based lidar output which can easily be processed by an MLP, and some others directly output the raw images to be processed by a CNN. When using lidars, we arbitrarily fix the number of beams to 19. We did not look into the influence of changing this parameter, which has been done by [28] in a similar setting.

As a side note, while [28] exhibit impressive results in a setting that may seem close to ours, it is actually far from that. In particular, despite the game they use being visually realistic, they do not use any vision processing. Instead, they use an ad-hoc simulator which allows them to collect ground-truth samples for parameters such as acceleration or road curvature from 80 agents in parallel. We use screenshots from the real video game and collect samples from one

single agent (although the framework we present later allows us to collect samples from several agents running in parallel if desired). Besides, it is unclear but likely that the simulator in [28] enforces a constant inference delay of 1 time-step (or no delay at all). Conversely, `rtgym` enforces this type of healthy time-steps in the real world, as well as in real-time video games over which we have no insider control (of course, a reward signal is still needed for training).

The first `rtgym` environment of our series was implemented in the Trackmania Nations Forever video game. This environment is illustrated in Figure 5.4. It uses a single lidar measurement per observation, computed from pixels, along with a speed computed with the 1-nearest neighbor algorithm from the relevant area of the screen. The reward function for this task is the raw speed. This task works only on tracks with black borders, due to how we compute the lidar measurements. Both lidar and velocity measurements are noisy, due to changing colors within the game.

Our second environment is very similar to the first one, with one substantial difference. To implement this environment, we moved on to the more recent Trackmania 2020 video game, which allows us to implement a more advanced and conceptually more interesting reward function. For this matter, we use the AngelScript [66] API provided by the OpenPlanet website [67]. This API enables access to some ground truth parameters of the game. From these parameters, we only allow the model to access the norm of the car's linear speed, so the observation space is the same as in our previous environment. However, internally, we use the ground truth position of the car to compute a reward function from a single demonstration trajectory, as illustrated in Figure 5.5. For a given track, this demonstration is recorded once. It does not have to be a good demonstration, but only to follow the track. Once the demonstration trajectory is recorded, it is automatically divided into equally spaced points. The time-step reward is then the number of these points that the car has passed since the previous time-step. In a nutshell, whereas the reward function of the previous environment

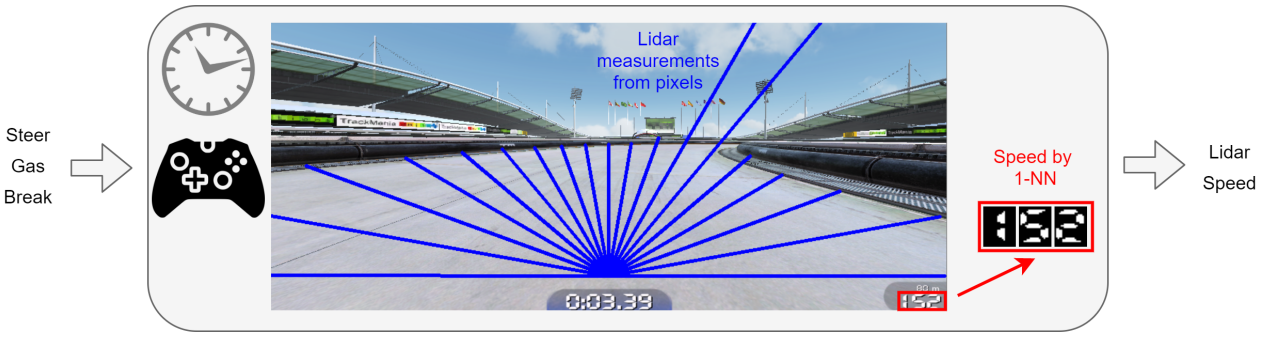

Figure 5.4 Autonomous racing task from preprocessed pixels

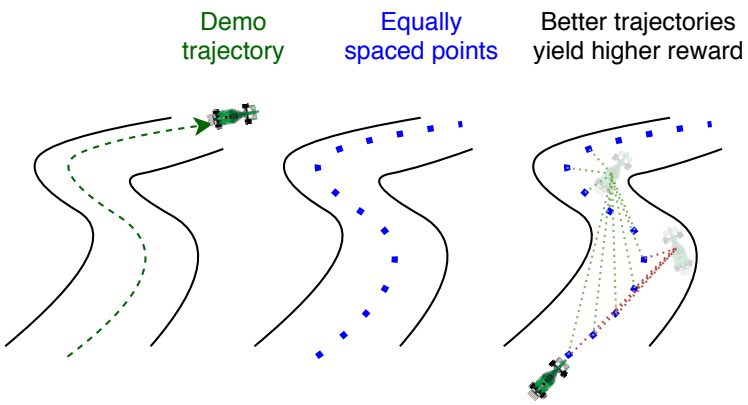

Figure 5.5 Reward function based on a demonstration trajectory

was measuring how fast the car was, this new reward function measures how good it is at covering a big portion of the track in a given amount of time.

Although in practice we find that it is sufficient to reach an acceptable performance, the observation space of these two tasks is naive. To highlight this point, one can imagine themselves in a similar situation: given a single image and a single number, what is the best option? The information provided by a single lidar is not enough to infer the dynamics of the world, and therefore the state space in these two environments is far from Markov. We alleviate this issue in a second version of the aforementioned environment, this time providing an history of the 4 last lidar measurements. This is made easy and meaningful by the fact that `rtgym` elastically constrains the time at which observations are retrieved. Therefore, this history of lidar contains enough information to statistically describe the first moments of the car's dynamics, such as the car's linear and angular speeds, its linear and angular accelerations, etc.

In the aforementioned task, `rtgym` naturally allowed us to use an history of lidar mea-

Figure 5.6 Autonomous racing tasks from raw images

surements computed from images whose retrieval was equally spaced in time. There is no reason why it shouldn't allow us to do exactly the same thing with the unprocessed images themselves, which is what we do in the last `rtgym` autonomous racing environment that we present in this thesis. Whereas all the previous environments were constrained to specific tracks featuring black borders that were making it easy to compute a lidar from pixels, this new task is conceptually compatible with any track, featuring all sorts of complex visuals as illustrated in Figure 5.6, and eventually with similar vision-based tasks in the real world. The observation space of this last task consist in an history of the 4 last screenshots taken by our environment, with no further processing. It is therefore much higher dimensional than the observation spaces of the previous tasks, which practically means that the model used with this environment must be a CNN, whereas the output of the previous environments could be processed by a simple MLP. In the next section, we present results in the lidar environments only, and we leave training CNNs in this last presented environment for future work.

5.2 Remote training framework

In this final section, we present a framework that we developed in order to train RL agents in real-time environments, such as the environments presented in the previous section. We call this framework our 'remote training framework'. Its purpose is to allow data collection to happen on one or several machines, while training is performed in parallel on another remote machine such as a GPU cluster. The architecture of this remote training framework is largely inspired from the popular RLlib library [49]. We use our framework instead of RLlib because it allows us to easily implement custom modular pipelines which are compatible with Real-Time Reinforcement Learning, and task-wise efficient.

5.2.1 Clients-server architecture

The remote training framework is made out of 3 main entities, which communicate over the Internet via the TCP protocol. This architecture is described by Figure 5.7.

In the middle of this network architecture, a central *Server* is responsible for gathering and broadcasting information. Namely, the Server gathers samples from all connected RolloutWorkers and forwards them to the TrainerInterface, while it receives updated weights from the TrainerInterface and broadcasts them to all connected RolloutWorkers. The Server must live on a machine that is reachable from other machines. In practice, there are two possibilities for achieving this: either the Server lives on a machine that is reachable from the Internet through port forwarding, either all machines live on the same local network.

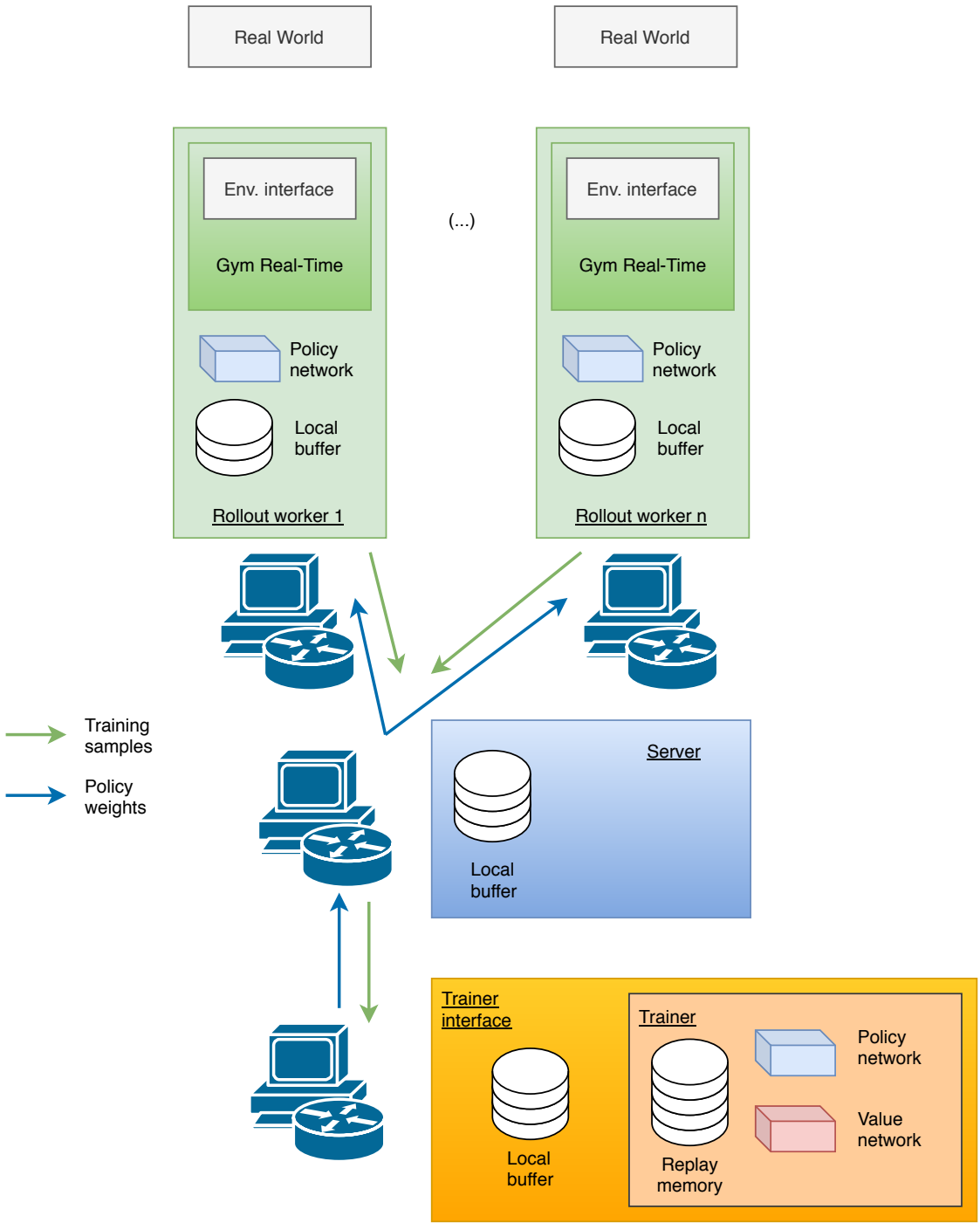

Figure 5.7 Architecture of the remote training framework

A *RolloutWorker* can be spawned at any time, either on the same or on a different machine, and as many RolloutWokers as desired can be spawned in parallel. When a RolloutWorker is spawned, it starts continuously collecting training samples from an embedded Gym environment, which in our case is a `rtgym` environment. At the same time, it repeatedly tries to connect to the central Server until the connection is successfully established. Once a RolloutWorker is connected to the central Server, it periodically sends buffers of collected samples, and the Server periodically sends updated policy weights. These weights are then used by the RolloutWorker to update its internal inference model.

These weights come from a third entity called the *TrainerInterface*. Similarly to the RolloutWorkers, the TrainerInterface can be spawned at any time and will automatically connect to the server, but only one single TrainerInterface can live in the network. Once the TrainerInterface is connected to the Server, it periodically receives the buffered samples collected by the Server from the RolloutWorkers. These samples are then used by the Trainer object contained within the TrainerInterface to train a model with an actor-critic algorithm. At the moment of writing this thesis, the only algorithm that is supported is SAC, and we are currently working on implementing DCAC in this framework for real-world robotic demonstrations of DCAC in our future work. The TrainerInterface then periodically sends the new trained weights of its policy network (i.e. the actor) to the Server, to be broadcast to the RolloutWorkers so they update their inference model. The value weights (i.e. the critic) are only relevant to the training algorithm, and therefore are not sent to the RollourWorkers.

5.2.2 Modular interfaces

To keep our code maintainable and useful for custom tasks, the remote training framework relies on abstract python interfaces. These interfaces are for the user to implement depending on their own application. The principal point of doing this is to implement custom pipelines that are more efficient brandwidth-wise and memory-wise than e.g. RLlib, and with more freedom regarding sample processing. In particular, our remote training framework enables the user to implement a procedure that compresses samples out of the Gym envrionment before storing them in the RolloutWorker's local buffer, a procedure that stores these samples in the replay memory at the level of the TrainerInterface, and a procedure that reconstructs the original sample from the replay memory. This is particularly useful in settings where an observation is made out of an history of images.

For example, when using the last computer-vision environment that we presented in the previous section, a raw observation contains 4 numpy arrays that represent an history of the 4 last captured images. The custom pipeline that we have implemented for this task

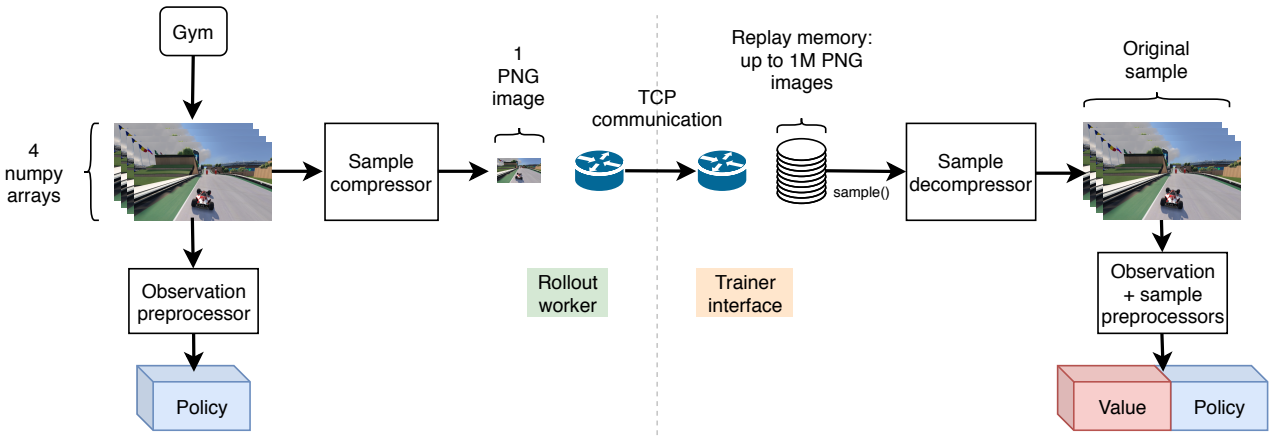

Figure 5.8 Example of custom pipeline in the remote training framework

is illustrated in Figure 5.8. The remote training framework's interface allows us to discard the 3 oldest images at the level of the RolloutWorker, and to convert the most recent image into PNG format, which considerably reduces its byte-wise size in a lossless way. Then our interface allows us to rebuild a numpy array out of this PNG image at the level of the replay memory when the sampling procedure is called, along with the 3 previous images contained in the memory so that we rebuild the full observation. Since such custom remote pipelines are bug-prone, we have also implemented a debugging procedure checking that the rebuilt samples are identical to the original samples (this procedure runs only at debug time, because it cancels the point of compressing samples in its current version). Other useful things that our framework enables are observation preprocessing (e.g. to format observations with respect to our custom models) and sample preprocessing (e.g. for data augmentation).

5.2.3 Trained real-time agents

To demonstrate our framework, we have trained a few RL agents in the real-time environments presented earlier in this chapter. At the moment of writing this thesis, this study is ongoing and our results in this setting are only preliminary. Nevertheless, we already conducted some qualitative experiments which we find worth mentioning. This will conclude this chapter and introduce our future work.

In all presented experiments, our model is an MLP similar to what we have used in the experimental section of Chapter 4. We train our `rtgym` tasks with the remote training framework and the SAC algorithm. Indeed, we have mainly experimented on the autonomous racing tasks so far, which have a constant 1-step delay for inference. In this specific setting, DCAC (RTAC) can only be expected to perform marginally better than SAC. We train using a single

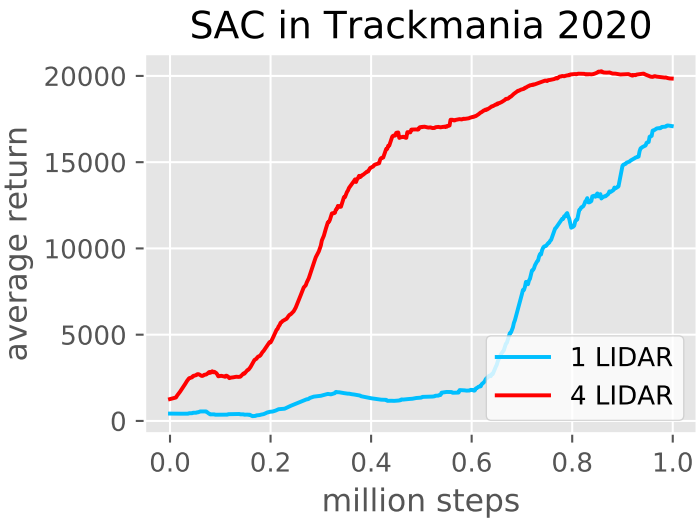

Figure 5.9 Training with 1 and 4 lidar measurements under the same reward function

RolloutWorker, which means using a single learning agent. As explained previously, training either on a real-world robot of in a real-time video game such as Trackmania is conceptually the same thing in this setting. Indeed, we do not use a time-controlled simulator, and therefore we do not pause nor accelerate the simulation during non-instantaneous operations such as inference or observation capture and processing.

Our first attempt was made in the Trackmania Nations Forever video game with only one single lidar measurement along with a float describing the instantaneous speed of the car. Both were computed in real-time from screenshots, and we used the raw speed as reward. Despite both the observation space and the reward function being quite naive, this setting was sufficient to reach a beginner level performance in our experiments. However, we quickly noticed that the AI was trying to take trajectories which were sub-optimal with respect to the lap-time, as it was taking large turns to go faster on average.

We believed the culprit was our naive reward function, and moved on to the equivalent task in Trackmania 2020. This new simulation allowed us to use the more advanced, trajectory-based reward described in Section 5.1.3. The training curve for this task is displayed in blue in Figure 5.9. The trajectories learned in this setting with this new reward function were slightly more sensible, but still sub-optimal. In particular, we kept oberving large turns in technical situations where we believe the optimal strategy was to break abruptly and take a sharp turn.

We then moved on to our third environment, this time feeding a concatenated history of the 4 last lidar measurements (captured at an elastically constant control rate of 20Hz) to our model,

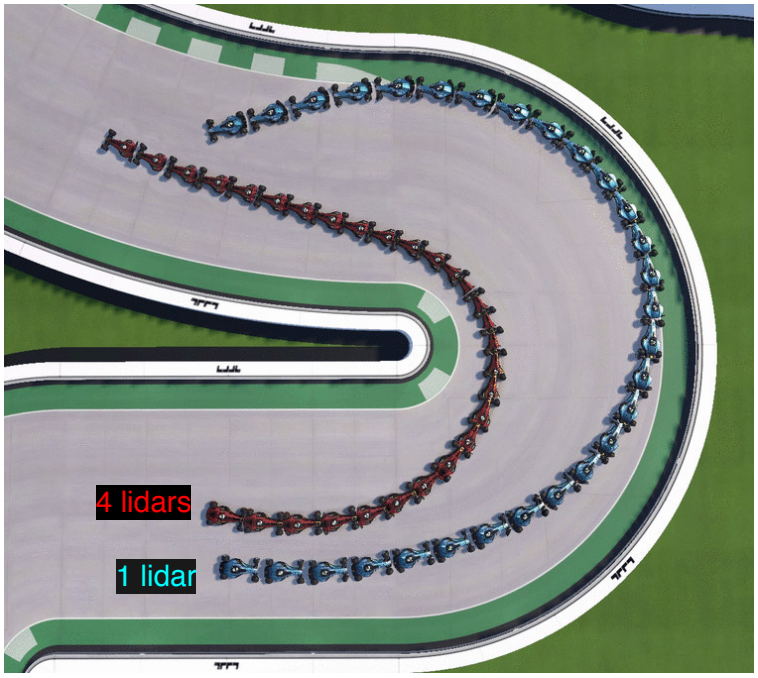

Figure 5.10 Trajectory comparison of the 1 and 4 lidar policies

along with the speed norm. We expected training to be slower due to the larger observation space, but to converge to a better policy eventually since the car was now able to infer more complex dynamics in a more Markovian setting. As seen in Figure 5.9, not only the training converged to a better policy, but converged faster. The AI learned more complex strategies, and was able to find better trajectories in technical situations. Figure 5.10 illustrates this point. In this Figure, an external camera is used in order to compare a trajectory learned with 1 single lidar + speed norm (blue car) to a trajectory learn with an history of 4 lidars + speed norm (red car) under the same reward function. For each car, 29 positions have been photographed with 10ms intervals. The red car now being able to estimate its acceleration and angular dynamics from this history of lidar measurements, we observe that it learned to break and turn sharply. On the other hand, the 1-lidar car's strategy is to maintain a roughly constant speed. We can observe that the 4-lidar car's strategy is indeed closer to optimal here. We estimate the current performance of the 4-lidar policy to be comparable to a slightly over-average human player.

Both policies were learned in about 12 hours of remote training. Training was performed on a P100 GPU and 6 Intel E5-2650 v4 Broadwell CPU cores provided by the Compute Canada HPC system [68], while samples were collected locally on a laptop running the game on an old GeForce 780m GPU and performing inference on an even older i7-2720QM CPU. The batch size used for minibatch gradient descent was 128 with the Adam optimizer. Training

episodes were 50s each, with 3s being lost in useless samples as the car waited for the starting signal at the beginning of each episode. We did not discard these samples. When new model weights were available at the end of an episode, the model was updated, otherwise a new episode started.

5.3 Discussion and future work

The results obtained thus far are encouraging, especially given the poor hardware setup. As a matter of fact, the slow control frequency of 20Hz is about the fastest we could achieve without `rtgym` timing-out as a result of slow observation retrieval. Indeed, we implemented observation capture as part of observation retrieval and used a buffer of 2 actions instead of 1 for all Trackmania our environments, c.f. Figure 4.16. This is convenient as this enforces a constant 1-step delay, but this makes observation capture our bottleneck by a large margin in lidar-based environments. Our end goal is to use a CNN and raw images instead of an MLP and pixel-based lidar measurements. In such situation, observation capture and action inference take about the same amount of time in our setup. Nevertheless, an interesting avenue for future work is to keep working with lidar-based environments and perform observation capture independently from observation retrieval, in a parallel infinite loop. This would enable higher control frequencies, at the price of introducing random observation delays.

Our near future work will consist in searching for hyperparameters that achieve better performance (our hyperparameter search has been really coarse so far). Currently, we are also working on training similar policies with raw images instead of lidar measurements. In addition, we are planning to perform similar experiments without the 'help' of the speed norm as an input, being interested in a pure vision-based approach. Eventually, we plan to demonstrate our approach on similar racing robots directly trained in the real world, as the only (yet non negligible) difficulty ahead is to design a safe `rtgym` environment that is similar to our Trackmania environments while protecting the robot from crashes.

Additionally, we are working on making the Cognifly benchmark environment more relevant for RL with random delays. In our preliminary experiments, we noticed that the laser rangefinder was too coarse and the flight controller was not aggressive enough, which made the task very easy even for SAC.

CHAPTER 6 CONCLUSIONS

In Chapter 3, we introduced a 2-player game that acts as an offline dataset collector for drone-racing tasks. Although we trained all the policies presented in this thesis from scratch (i.e. with replay memories initialized empty and models initialized randomly), our remote framework enables initializing the replay memory directly from an available similar dataset. As previously pointed out, the dataset-collector game presented in Chapter 3 was working in a time-pausing-and-stepping fashion as usual RL simulators do, and a policy learned directly from such simulator is not trivially compatible with the real-time setting. Nevertheless, as shown in Chapter 4, a real-world randomly delayed environment is simply a classical turn-based MDP augmented with random delay dynamics. Unfortunately, we believe that augmenting a policy learned under the undelayed MDP with the delays from the RDMDP is intractable. However, it is very straightforward to turn an offline dataset of samples collected under the undelayed MDP (e.g. our dataset collector game) into an RDMDP dataset. Indeed, one simply has to artificially modify the dataset with the desired delay dynamics. For instance, in Chapter 5, we use `rtgym` in actual real-time application with a simple 1-step delay that is sufficient to cover action inference and observation capture in parallel. To train a policy that is compatible with this setting from a dataset collected under the undelayed MDP, one simply needs to shift all actions by one time-step, and augment each state with the corresponding action buffer. A similar trick can be implemented with any RDMDP .

In Chapter 4, we proposed a deep off-policy and planning-free approach that explicitly tackles the credit assignment difficulty introduced by real-world random delays. This is done by taking advantage of delay measurements in order to generate actual on-policy sub-trajectories from off-policy samples. In addition, we provided a theoretical analysis that can easily be reused to derive a wide family of algorithms such as DCAC, while previous work in this area mostly dealt with finding approximate ways of modeling the state-space in constantly delayed environments. The action-buffer is fundamentally required to define a Markovian state-space for RDMDPs, but it is of course possible to observe this action-buffer approximately, e.g. by compressing it in the hidden state of an RNN, which is complementary to our work. We have designed our approach with real-world applications in mind, and it is scalable to a wide variety of scenarios such as those presented in Chapter 5. To the best of our knowledge, DCAC is the first deep actor-critic approach to exhibit such strong performance on both randomly and constantly delayed settings, as it makes use of the partially known dynamics of the environment to compensate for difficult credit assignment. We believe that our model can be further improved by making use of the fact that our critic estimates the state-value

instead of the action-value function. Indeed, in this setting, [46] showed that it is possible to simplify the model by merging the actor and the critic networks using the PopArt output normalization [69], which we did not try yet and leave for future work. Our approach handles and adapts to arbitrary choices of time-step duration, although in practice time-steps smaller than the upper bound of the inference time will require a few tricks. We believe that this approach is close to time-step agnostic RL, and will investigate this direction in future work.

Finally, in Chapter 5, we introduced a small framework, Real-Time Gym (`rtgym`), that abstracts real-time considerations from both the developer and the end user. Real-Time Gym makes the practitioner's life easy when implementing Gym environments in real-time applications, as we demonstrated on real-time simulations and real-world robots. In particular, we demonstrated this framework on a real-time racing video game. Of course, this game is not the real world in the sense that it is not controlling real robots, nevertheless, it is a real-world application in the sense of real time since we have no control over the underlying simulation and simply capture screenshots and send commands with real-time delays, as humans do. RL agents were trained on these tasks using a framework which we also introduced in Chapter 5, and which enables training on distant machines with optimal, task-specific pipelines. We collected samples from a single agent in real time, and achieved very encouraging performance where the agent demonstrated learning near-optimal trajectories. In our future work, we intend to train even more impressive policies in similar single-worker real-time setups, and port this approach to real racing robots.

We spent a fair amount of attention to making this work both theoretically strong and practically relevant for real-world applications. Whereas in practice it is sometimes difficult to fully implement an RDMDP with a robot since the agent must explicitly observe delays and discard outdated observations, it is straightforward to relax our analysis when either of those things are not possible, as seen in Section 4.6.4. For instance, when individual delays are not measurable, one can still use DCAC with an estimation of the minimum delay. The frameworks presented in Chapter 5 can be used in any of these situations. Delays are inherent to many real-world control situations, and are most relevant when the practitioner is concerned with acting at high speed or, equivalently, in the presence of long latencies. Therefore, a broad range of applications such as autonomous cars, high-speed trading and remote control of space robots could benefit from our work.

Our code is available at the following URLs:
Chapter 3: https://github.com/yannbouteiller/gym-airsimdroneracinglab
Chapter 4: https://github.com/rmst/rlrd
Chapter 5: https://github.com/yannbouteiller/rtgym

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] I. Goodfellow *et al.*, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[3] D. Silver *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] A. Ecoffet *et al.*, "First return then explore," *arXiv preprint arXiv:2004.12919*, 2020.

[5] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[7] K. Doya, "Reinforcement learning in continuous time and space," *Neural computation*, vol. 12, no. 1, pp. 219–245, 2000.

[8] G. Brockman *et al.*, "Openai gym," 2016.

[9] K. V. Katsikopoulos and S. E. Engelbrecht, "Markov decision processes with delays and asynchronous cost collection," *IEEE transactions on automatic control*, vol. 48, no. 4, pp. 568–574, 2003.

[10] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control. in 2012 ieee," in *RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033.

[11] N. Cruz, K. Lobos-Tsunekawa, and J. Ruiz-del Solar, "Using convolutional neural networks in robots with limited computational resources: detecting nao robots while playing soccer," in *Robot World Cup*. Springer, 2017, pp. 19–30.

[12] D. Kalashnikov *et al.*, "Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation," *arXiv preprint arXiv:1806.10293*, 2018.

[13] Y. Bouteiller *et al.*, "Reinforcement learning with random delays," in *International Conference on Learning Representations*, 2021.

[14] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[15] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[16] H. Seijen and R. Sutton, "True online td (lambda)," in *International Conference on Machine Learning*, 2014, pp. 692–700.

[17] T. Haarnoja *et al.*, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv preprint arXiv:1801.01290*, 2018.

[18] ——, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018.

[19] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[20] T. Haarnoja *et al.*, "Learning to walk via deep reinforcement learning," *arXiv preprint arXiv:1812.11103*, 2018.

[21] W. Fedus *et al.*, "Revisiting fundamentals of experience replay," in *International Conference on Machine Learning*. PMLR, 2020, pp. 3061–3071.

[22] I. Akkaya *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[23] W. Koch, R. Mancuso, and A. Bestavros, "Neuroflight: Next generation flight control firmware," *arXiv preprint arXiv:1901.06553*, 2019.

[24] S. Mysore *et al.*, "Regularizing action policies for smooth control with reinforcement learning," in *IEEE International Confernece on Robotics and Automation*, 2021.

[25] R. Madaan *et al.*, "Airsim drone racing lab," in *NeurIPS 2019 Competition and Demonstration Track*. PMLR, 2020, pp. 177–191.

[26] A. Dosovitskiy *et al.*, "Carla: An open urban driving simulator," in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[27] "Trackmania." [Online]. Available: https://trackmania.com/

[28] F. Fuchs *et al.*, "Super-human performance in gran turismo sport using deep reinforcement learning," 2020.

[29] J. Schulman *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[30] R. Lowe *et al.*, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arXiv preprint arXiv:1706.02275*, 2017.

[31] J. N. Foerster, "Deep multi-agent reinforcement learning," Ph.D. dissertation, University of Oxford, 2018.

[32] F. Torabi, G. Warnell, and P. Stone, "Recent advances in imitation learning from observation," *arXiv preprint arXiv:1905.13566*, 2019.

[33] X. Zhang *et al.*, "f-gail: Learning f-divergence for generative adversarial imitation learning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[34] V. Tangkaratt *et al.*, "Variational imitation learning with diverse-quality demonstrations," in *International Conference on Machine Learning*, 2020.

[35] F. Yang *et al.*, "Bayesian multi-type mean field multi-agent imitation learning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[36] Y. Tang, "Self-imitation learning via generalized lower bound q-learning," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[37] Y. Pan *et al.*, "Imitation learning for agile autonomous driving," *The International Journal of Robotics Research*, vol. 39, no. 2-3, pp. 286–302, 2020.

[38] S. Fujimoto, D. Meger, and D. Precup, "Off-policy deep reinforcement learning without exploration," in *International Conference on Machine Learning*, 2019, pp. 2052–2062.

[39] Y. Wu, G. Tucker, and O. Nachum, "Behavior regularized offline reinforcement learning," *arXiv preprint arXiv:1911.11361*, 2019.

[40] A. R. Mahmood *et al.*, "Setting up a reinforcement learning task with a real-world robot," 2018.

[41] T. J. Walsh *et al.*, "Learning and planning in environments with delayed feedback," *Autonomous Agents and Multi-Agent Systems*, vol. 18, pp. 83–105, 2008.

[42] E. Schuitema *et al.*, "Control delay in reinforcement learning for real-time dynamic systems: A memoryless approach," in *International Conference on Intelligent Robots and Systems*, 2010.

[43] T. Hester and P. Stone, "Texplore: real-time sample-efficient reinforcement learning for robots," *Machine learning*, vol. 90, no. 3, pp. 385–429, 2013.

[44] V. Firoiu, T. Ju, and J. B. Tenenbaum, "At human speed: Deep reinforcement learning with action delay," *CoRR*, vol. abs/1810.07286, 2018.

[45] E. Derman, G. Dalal, and S. Mannor, "Acting in delayed environments with non-stationary markov policies," in *International Conference on Learning Representations*, 2021.

[46] S. Ramstedt and C. Pal, "Real-time reinforcement learning," in *NeurIPS*, 2019.

[47] T. Xiao *et al.*, "Thinking while moving: Deep reinforcement learning with concurrent control," in *International Conference on Learning Representations*, 2020.

[48] S. Shah *et al.*, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*. Springer, 2018, pp. 621–635.

[49] E. Liang *et al.*, "Rllib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning*, 2018, pp. 3053–3062.

[50] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[51] J. Hwangbo *et al.*, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[52] J. Nilsson, B. Bernhardsson, and B. Wittenmark, "Stochastic analysis and control of real-time systems with random time delays," *Automatica*, vol. 34, no. 1, pp. 57–64, 1998.

[53] Y. Ge *et al.*, "Modeling of random delays in networked control systems," *Journal of Control Science and Engineering*, vol. 2013, 2013.

[54] C. M. Grinstead and J. L. Snell, *Introduction to probability*. American Mathematical Soc., 2012.

[55] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016, vol. 10.

[56] R. Munos *et al.*, "Safe and efficient off-policy reinforcement learning," in *Advances in Neural Information Processing Systems 29*, D. D. Lee *et al.*, Eds. Curran Associates, Inc., 2016, pp. 1054–1062.

[57] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[58] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[59] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[60] G. Brockman *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[61] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *arXiv preprint arXiv:1509.06461*, 2015.

[62] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[63] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[64] L. Santinelli, F. Guet, and J. Morio, "Revising measurement-based probabilistic timing analysis," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 199–208.

[65] R. de Azambuja, "thecognifly/YAMSPy: Yet Another Multiwii Serial Protocol Python Interface," Dec. 2020, This work was possible thanks to the financial support from IVADO.ca (postdoctoral scholarship 2019/2020). [Online]. Available: https://doi.org/10.5281/zenodo.4306818

[66] angelscript. [Online]. Available: https://www.angelcode.com/angelscript/

[67] openplanet. [Online]. Available: https://openplanet.nl/

[68] S. Baldwin, "Compute canada: advancing computational research," in *Journal of Physics: Conference Series*, vol. 341, no. 1. IOP Publishing, 2012, p. 012001.

[69] H. P. van Hasselt *et al.*, "Learning values across many orders of magnitude," in *Advances in Neural Information Processing Systems*, 2016, pp. 4287–4295.