POLITECNICO DI MILANO
School of Industrial and Information Engineering

Master of Science in Computer Science and Engineering

Dynamic Control Frequency in Reinforcement Learning through Action Persistence

Supervisor: Prof. Marcello Restelli

Co-supervisor: Dott. Lorenzo Bisi
Dott. Alberto Maria Metelli
Dott. Luca Sabbioni

Candidate:
Al Daire Luca, matricola 10687921

Academic Year 2020-2021

dedica

Desidero ringraziare il Prof. Marcello Restelli,
il Dott. Alberto Maria Metelli, il Dott. Luca Sabbioni
e il Dott. Lorenzo Bisi per il loro impegno e supporto
durante lo sviluppo e la stesura della tesi.
Ringrazio i miei genitori e mio fratello
per essermi stati accanto in questo periodo
e per il grosso sostegno ricevuto.

Contents

Contents **iii**

List of Figures **vii**

List of Algorithms **viii**

Sommario **ix**

Abstract **xi**

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Chapter Organization . . . 4

2 Background **5**
- 2.1 Reinforcement Learning . . . 5
- 2.2 Markov Decision Processes . . . 6
  - 2.2.1 Policy and Value Function . . . 7
  - 2.2.2 Bellman Operators . . . 8
  - 2.2.3 Dynamic programming . . . 10
- 2.3 Model-free prediction . . . 12
  - 2.3.1 Monte Carlo . . . 12
  - 2.3.2 Temporal Difference . . . 13
- 2.4 Model-free control . . . 14
  - 2.4.1 SARSA . . . 14
  - 2.4.2 Q-learning . . . 15
- 2.5 Function approximation . . . 16
- 2.6 Neural Networks . . . 17
  - 2.6.1 Deep Learning . . . 19

2.6.2 Deep Q-Learning 20

3 State of the Art **23**
3.1 Static Methods 23
3.2 Dynamic Methods 25
3.2.1 Tabular Methods 25
3.2.2 Neural Networks 26

4 Problem Formulation **29**
4.1 Action Persistence 29
4.1.1 Persistence as Options 30
4.1.2 All-Persistence Bellman Operator 31
4.2 Persistent Q-learning 35

5 Algorithm Implementation **38**
5.1 Language and Library 38
5.2 Persistent Tabular Q-learning 39
5.3 Deep Persistent Q-learning 43
5.3.1 Baselines 43
5.3.2 DeepQ Baselines 44
5.3.3 Deep Persistent Q-learning Implementation 45

6 Empirical Advantages of Persistence **52**
6.1 Exploration 52
6.2 Kemeny's Constant 56
6.3 Sample Complexity 59

7 Experimental Evaluation **63**
7.1 Methodologies 63
7.2 Details on Experimental Evaluation 68
7.2.1 Tabular Environments 68
7.2.2 Mountain Car 71
7.2.3 Atari 72
7.3 Results 74
7.3.1 Tabular 74
7.3.2 Mountain Car 78
7.3.3 Atari 79

8 Conclusions and Future Work **84**
8.1 Conclusion 84
8.2 Future work 85

References **86**

List of Figures

2.1 Agent interaction with environment in Reinforcement Learning settings . . . 5
2.2 MDPs representation example. Each circle is a state and each arrow represent a transition . . . 7
2.3 Policy Improvement process. . . . 10
2.4 Neuron representation. . . . 17
2.5 Fully connected network structure example. . . . 18
2.6 Example of convolution operation used by CNNs . . . 19
2.7 Deep Q-network presented by Mnih, Kavukcuoglu, Silver, et al., 2015 [21] . . . 20
2.8 A popular single stream Q-network (top) and the dueling Q-network (bottom) as presented in Wang et al., 2016 [38]. . . . 22

3.1 Agent-environment interaction without (top) and with (bottom) action persistence, highlighting duality. The transition generated by the k-persistent MDP $\mathcal{M}_k$ is the cyan dashed arrow, while the actions played by the k-persistent policy are inside the cyan rectangle. . . . 24
3.2 Network used in Dynamic action repetition for deep reinforcement learning . . . 26
3.3 Schematic representations of considered architectures for learning when to make decisions, where a_t is the action coming from a separate behaviour policy . . . 27
3.4 Architecture with shared feature representation for joint learning of when to make a decision and what action to take. . . . 27
3.5 TAAC's two-stage policy during inference. In the first stage, an action policy π_ϕ samples a candidate action $\hat{a}$. In the second stage, a binary switching policy β chooses between this candidate and the previous action a^- . . . 28

4.1 An example of the application of Algorithm 4.8 with $\overline{\kappa}_t = 2$ and $K_{\max} = 3$. Rows from top to bottom represent the updates in order. Updates involving the application of $\widehat{T}_t^{\star}$ and $\widehat{T}_t^{\overline{\kappa}_t}$ are denoted, respectively, by magenta and blue nodes, while dashed arrows represent the bootstrap persistence. 36

5.1 Train process with sub-persistence transitions. Red is the initial state and blue is the last state of the transition. 41
5.2 Example of bootstrap update for persistence 5 starting from a sub-persistent transition of 2. Red is the initial state and blue is the last state of the transition. The blue dotted lines represent the additional steps of the bootstrap. 42
5.3 Environment and agent interaction with persistence. 43
5.4 MLP example network with shared configuration 47
5.5 Tensorflow's Graph created by *build_act* function. Yellow nodes are input variables. 49
5.6 Tensorflow's Graph created by *build_train* function. Yellow nodes are input variables. 50

6.1 A full random policy $\psi \in \Psi$ over persistence options $\mathcal{O}$ in Mountain Car environment. The value represents in a logarithmic scale the counter of visited states. The x and y axes are respectively the position and the velocity of the car. We have used different K_{max} values. 10.000 episodes 54
6.2 A full random policy with fixed persistence K in Mountain Car environment. The value represents in a logarithmic scale the counter of visited states. The x and y axes are respectively the position and the velocity of the car. We have used different K values. 10.000 episodes 55
6.3 Normalized Kemeny's constant and entropy in Tabular environments as function of the maximum persistence and horizon H. . . . 58
6.4 Normalized Kemeny's constant in tabular environments as function of the maximum persistence $K_{\max}$. Bullets represent the minimum value of the constant. 59
6.5 Grid environment. Red cell denote holes, green cells the goal. . . . 60
6.6 k-value function representation for different persistence values k ($K_{\max} = 6$). Red cells denote holes, green cell the goal. 60

6.7 L_∞ error on 6x6 grid-world between synchronous Q-learning and Per$Q$-learning (left) and for different persistence options $k \in \{1, ..., 6\}$ of Per$Q$-learning (right). (100 runs, avg ± 95 % c.i.) . . . 61

7.1 Main page for the local server of Tensorboard. 64
7.2 Text parameter saved by Tensorboard. 64
7.3 Margin chart with mean and standard deviation. 65
7.4 65
7.5 Q function for for the environment Freeway with persistence 3. . . 66
7.6 Exploration percentage and mean persistence collected by Tensorboard 66
7.7 Persistence frequency collected by Tensorboard 67
7.8 Mean reward of last 100 episodes. 67
7.9 Example plot with mean and standard deviation through seeds of the same run (MountainCar environment). 68
7.10 Tabular Gridworlds. Red cells denote the starting state and blue cells the goal state. 70
7.11 Generated MyFroznLake example. Red cells denote the starting state and blue cells the goal state. 70
7.12 Mountain Car OpenAI Gym implementation. 71
7.13 Comparison of Persistent Q-learning with different persistences in all tabular environments. In the legend, parenthesis denote the selected $K_{\max}$. 50 runs (avg± 95% c.i.). 74
7.14 Comparison of Q-learning, Persistent Q-learning and TempoRL. Per$Q$-learning and TempoRL with persistence 8. 50 runs (avg± 95% c.i.). 75
7.15 Results of Q learning, TempoRL, PerQ with and without bootstrap in different tabular environments and maximum persistences. On each row, a different maximum persistence is selected for both algorithms. 50 runs (avg± 95% c.i.). 76
7.16 Results of Q learning, TempoRL, PerQ with and without bootstrap in different tabular environments and maximum persistences. On each row, a different maximum persistence is selected for both algorithms. 50 runs (avg± 95% c.i.). 77
7.17 MountainCar results for DQN, PerDQN and TempoRL. Parenthesis in the legend denote $K_{\max}$. 20 runs (avg± 95% c.i.). 78

7.18 PerDQN additional results on MountainCar. Return comparison of PerDQN with and without bootstrap and TempoRL. 20 runs (avg $\pm$ 95%c.i.) 79
7.19 Atari games results for DQN and PerDQN, with $K_{\max} = 4$ and 8. 5 runs (avg$\pm$ 95% c.i.). 80
7.20 Freeway games results for DQN and PerDQN, comparing the version with bootstrapping disabled, with $K_{\max} = 4$. 5 runs (avg$\pm$ 95% c.i.). 81
7.21 Atari games results for DQN and PerDQN, comparing the version with bootstrapping disabled, with $K_{\max} = 4$. 5 runs (avg$\pm$ 95% c.i.). 81
7.22 Freeway games results for DQN, PerDQN and TempoRL, with $K_{\max} = 4$ and $K_{\max} = 8$. 5 runs (avg$\pm$ 95% c.i.). 82
7.23 Atari games results for DQN, PerDQN and TempoRL, with $K_{\max} = 4$. 5 runs (avg$\pm$ 95% c.i.). 83

List of Algorithms

2.1 Iterative Policy Evaluation for estimating $V \approx V^{\pi}$ 11
2.2 Policy Iteration for estimating $\pi \approx \pi^{*}$ 11
2.3 Value Iteration for estimating $\pi \approx \pi^{*}$ 12
2.4 First-visit MC prediction for estimating $V \approx V^{\pi}$ 13
2.5 Tabular TD for estimating $V \approx V^{\pi}$ 14
2.6 Sarsa (on-policy TD control) for estimating $Q \approx Q*$ 15
2.7 Q-learning (off-policy TD control) for estimating $\pi \approx \pi*$ 16
4.8 All Persistence Bellman Update 37
4.9 Persistent Q-learning (Per$Q$-learning) 37
5.10 Function *get_action(state)* 40
5.11 Multiple Replay Buffer Storing 46

Sommario

Nel campo del Reinforcement Learning, la possibilità di controllare la frequenza di esecuzione delle scelte, usata dall'agente per interagire con l'ambiente circostante, può avere significative ripercussioni sul processo di apprendimento della policy ottima.

L'obiettivo di questo progetto consiste nell'implementazione di un metodo adeguato per poter ottenere una frequenza di controllo dinamica. In particolare, abbiamo sfruttato la persistenza delle azioni, un meccanismo che vincola l'agente a ripetere la stessa azione più volte. Con questa tecnica all'agente è permesso operare a diverse frequenze, che sono multiple della frequenza di controllo base.

Inoltre, abbiamo cercato un sistema per sfruttare le informazioni collezionate a varie persistenze per poter avere una sample complexity inferiore rispetto agli algoritmi classici. Per questa ragione abbiamo sviluppato un nuovo operatore di Bellman composto da due parti. La prima utilizza i sample collezionati per stimare la policy ottima, mentre la seconda stima la policy per persistenze più alte utilizzando le informazioni parziali precedentemente raccolte.

Dopo aver implementato il sistema ipotizzato tramite l'impiego delle reti neurali, abbiamo eseguito vari esperimenti su diversi videogiochi di vecchia generazione (Atari 2600). I risultati ottenuti hanno evidenziato come la persistenza dinamica delle azioni permetta non solo di poter esplorare un insieme più ampio di stati del nostro ambiente, ma anche di ridurre la sample complexity con cui l'agente riesce ad imparare la rappresentazione del modello richiesto.

Parole Chiave: Apprenimento tramite Rinforzo, Deep Learning, Persistenza delle Azioni, Frequenza di Controllo, DQN, Deep Q-learning, Atari

Abstract

In Reinforcement Learning, adopting a good control frequency, i.e. the frequency used by the agent to choose the action during the interaction process with the environment, may significantly affect the process of learning the optimal policy.

The aim of this project is to propose the implementation of a new method to obtain a dynamic control frequency. Especially, we exploited the action persistence mechanism, that is to say binding the agent to repeat the same action more than once. With this technique the agent is allowed to operate at different frequencies that are multiple of the base control frequency.

Moreover, we sought a manner to exploit the information collected with different persistences in order to obtain a lower sample complexity compared to the classical methods. Indeed, we developed a new Bellman operator consisting of two parts. The first part uses collected samples to estimate directly the optimal policy, meanwhile, the second one estimates the policy with higher persistence starting from the partial information previously collected.

After a neural network implementation of the proposed method, we have tested our algorithms with different environments and old-gen video-games (Atari 2600). The results obtained point out how the dynamic persistence of the actions allows not only to be able to explore a wider set of states of our environment, but also to reduce the sample complexity, used by the agent to learn the representation of the requested model.

Keywords: Reinforcement Learning, Deep Learning, Action Persistence, Control Frequency, DQN, Deep Q-learning, Atari

Chapter 1

Introduction

In recent years, the Reinforcement Learning field (RL, Richard S Sutton et al., 2018 [34]) has gone through various improvements and has eventually achieved many successes, especially since the advent of Deep Q-Learning, developed by DeepMind Laboratory in 2015, has been introduced. Many features and better algorithms have been implemented, but one of the aspects that barely changed is how the agent collects the samples from the environment, a process usually called *exploration*. Depending on the exploration method chosen, we can modify the set of collected samples and, subsequently, the model built by our agent. The model pursues a given task, an abstract objective concerning our environment (e.g. finding the exit in a maze). Effectively, from our task we can naturally devise a set of rewards for our environment (numerical values gained by our agent in response to its actions). From the rewards collected, our agent can improve its knowledge of the surrounding environment. Based on the model built, we are able to complete the task in a more efficient way or not.

One of the perks of the most common exploration algorithms is generality. Indeed, they do not specifically rely on the underlying environment and, therefore, they are easy to implement and adapt in a wide range of tasks. Anyway, if we used the environment features, then we could achieve better performances. When we have to deal with a sparse reward signal, these methods don't have an informative reward, therefore they rely on uniformly sampling actions from the action space, independently of the history of the agent. It is clear that with a sparse reward it becomes unlikely for our agent to choose the correct path, especially when the environment has a very high horizon length. Imagine a squared maze, with the initial state in the center cell and our goal is to reach any of the borders of the maze. We can define a reward greater than zero in every border cell and a negative reward elsewhere. With classic exploration methods, there are two ways, for our agent, to discover the goal: it can sample a sequence of actions that lead the agent to the goal, still this is extremely unlikely when the maze is really huge; otherwise, we

should wait the agent to progressively learn that the concentric circles, with the origin in the center of the maze, are bad positions. An even trickier scenario is when we have zero reward signal in the middle states, in that case our agent will iterate randomly over the trajectory space, hoping to reach at least one goal state, but it will be probably stuck in a limited region of the environment. Once we have reached the goal, the information is only contained in the local area near the goal. We need to propagate this information to the neighbour states up to the initial state. This process requires the goal state to be reached many times. And this would result in a further problem. In this thesis we face all of the latter problems concerning exploration and sample complexity reduction.

The aim of this project is the develop of a way to introduce in an online Reinforcement Learning context the possibility to control the frequency of our agent. When the agent starts to interact with the environment, it has to take decision and, in general, it choose an action with a certain frequency f, or, equivalently, at each time step $\delta = \frac{1}{f}$. The choice of the control frequency f of a system has a relevant impact on the ability of reinforcement learning algorithms to learn a highly performing policy. It would be great to have the highest possible frequency, because it allows for greater control opportunities, but they have significant drawbacks. The most relevant one is related to the toned down effect of the selected actions. In the limit for time $\delta \to 0$, the advantage of each action collapses to zero, preventing the agent from finding the best action (Baird, 1994 [3]; Tallec et al., 2019 [36]). Even policy gradient methods are shown to fail in this (near-)continuous-time setting, and the reason is related to the divergent variance of the gradient (Park et al., 2021 [23]). The consequences of each action might be detected if the dynamics of the environment has the time to evolve, hence an agent acting with higher frequencies leads to higher sample complexity.

Another consequence of the use of high frequencies is related to the difficulty of exploration. A random uniform policy played at high frequency may not be adequate, as in some classes of environments, including the majority of real-world control problems, tends to visit only a local neighborhood of the initial state (Amin et al., 2020 [1]; Park et al., 2021 [23]; Yu et al., 2021 [40]). This is problematic, especially in goal-based or sparse rewards environments, where the most informative states may never be visited. On the other hand, higher frequency benefit from a higher probability of reaching far states, but they also deeply modify the transition process, hence a possibly large subspace of states may not be reachable.

1.1 Contributions

One of the solutions to achieve the advantages related to exploration and sample complexity, while keeping the control opportunity loss bounded, consists in *action persistence* Schoknecht et al., 2003 [30]; Braylan et al., 2015 [6]; Lakshminarayanan et al., 2017 [14];

Metelli et al., 2020 [19], as the possibility of our agent to repeat an action more than one starting from a state. When the dynamics of an environment are very rapid, action repetition is equivalent to acting at lower frequencies. Thus, the agent can achieve, in some environments, a more effective exploration, better capture the consequences of each action, and, as a final consequence, learn the optimal policy faster.

In particular, the present thesis is based upon the results of a previous work (Metelli et al., 2020 [19]), of which we can consider to be a continuation. They proposed a formalism to describe a fixed action persistence introducing a persistent Markovian Decision Process and, consequentially, a persistent policy that let the agent to choose the same action for a fixed amount of steps. They developed also the Persistent Fitted Q-Iteration, an extension of Fitted Q-Iteration, to analyze empirically what happens with different persistence settings, and also a heuristic method to select the best possible persistence. The results obtained show that an higher persistence is often able to improve both the exploration and exploitation phase.

In this work, we instead propose a value-based approach in which the agent does not only choose the *action*, but also its *persistence*, with the goal of making the most effective use of samples collected at different persistences. The main contribution of this paper is a general approach in which information collected from the interaction with the environment at *one* persistence is used to improve the action value function estimates of *all* the considered possible persistences. On one hand, the transitions sampled with a persistence $\overline{\kappa}$ can be decomposed in many sub-transitions of reduced length and used to update lower persistence $k \leq \overline{\kappa}$ value functions. On the other hand, they represent partial information for the estimation of the effects of higher persistence $k > \overline{\kappa}$ actions. Indeed, they can be employed to update the estimates by using a suitable *bootstrapping* procedure of the missing information. This means that, after the interaction with the environment, according to the action-persistence pair selected by the agent, all value function estimates are updated simultaneously for each of the available persistences $k \in \mathcal{K}$.

We formalize this procedure by introducing the *All-persistence Bellman Operator*. We prove that such an operator enjoys a contraction property analogous to that of the traditional optimal Bellman operator. Consequently, we embed the All-persistence Bellman operator into the classic Q-learning algorithm, obtaining *Persistent Q-learning* (Per$Q$-learning). This novel algorithm, allowing for an effective use of the transitions sampled at different persistences, displays two main advantages. First, since each individual transition is employed to update the value function estimates at different persistences, we experience a faster convergence. Second, the execution of persistent actions, given the nature of a large class of environments, fosters exploration of the state space, with a direct effect on the learning speed.

1.2 Chapter Organization

In this section we briefly describe the structure of this thesis.

Chapter 2 discusses basic Reinforcement Learning concepts used throughout our work, from the classic methods to the most recent tools. We start from a general definition of Reinforcement Learning and the description of a Markov Decision Process, a model used to describe a Reinforcement Learning problem, and we end with the description of function approximation and Deep Q-Learning, one of the first algorithms developed to combine Neural Networks and Reinforcement Learning.

Chapter 3 describes different works that analyze the effect of persistence in Reinforcement Learning and some of the solutions proposed by other authors.

Chapter 4 presents the solution proposed, with the definition of a new set of options to include a dynamic persistence in a MDP model and a new defined pair of Bellman Operators, used to build our agent. We also provide different properties and theorems about the convergence of the new operators.

Chapter 5 shows the implementation of the proposed approach. We start from the tabular version, a simple extension of the standard Q-learning algorithm. A natural evolution of this approach consists in the adoption of Deep Neural Networks, described at the end of the chapter.

Chapter 6 discusses some advantages induced by persistence. When the persistence is introduced the distribution of visited states changes and our agent can cover better the set of states of our environment and reach in a faster way the goal.

Chapter 7 analyses the obtained results of our experiments. We have used our implementation of the tabular version in some simple grid environments and compared both with the standard Q-learning and with the TempoRL method, another method that propose a different solution to the same problem. After that we have analyzed the results of our evaluation with more complex settings. We have used the classic Mountain Car environment and different Atari 2600 games to evaluate our Deep Neural Network's implementation, and we have compared our solution with DQN and with TempoRL.

Chapter 8 closes with the summary of the work done and what we have discovered during our experiments. We also talk about possible future works and ideas come out from this project.

Chapter 2

Background

In this chapter we will introduce the main concepts about Reinforcement Learning. We start with the pure definition of Reinforcement Learning and what is a Markov Decision Process. Then we proceed with the the introduction of the Bellman operator which will be useful in a dynamic programming context. We will briefly introduce some basic algorithms to guide the agent in an environment without knowledge of the model. Lastly we will briefly discuss about how to use neural networks to approximate the state-action value function introduced in model free approaches.

2.1 Reinforcement Learning

Reinforcement Learning is a branch of Machine Learning that studies the problem of learning what to do in a context in which an agent is capable of interacting with an environment. The following figure 2.2 explains the interaction.

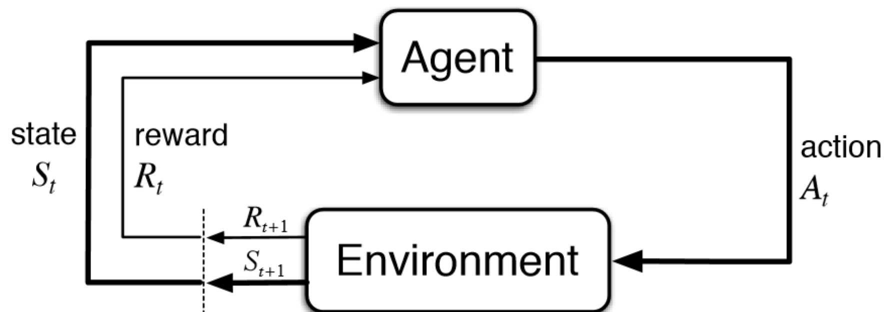


Figure 2.1: Agent interaction with environment in Reinforcement Learning settings

At each time step the agent executes an action from a specific set, the environment evolves and the first receives a reward from the environment. For example we can image

a robot trapped in a maze with the purpose to find the exit. The robot is the agent, the maze is the environment and the possible movements of the the robot (go straight, turn left, etc.) are the actions. Every time the robot performs an action the environment or the agent can change (the position of the robot), after that the agent receives a reward: a signal designed by the problem that represents the results of our action. Let's define a possible design for the rewards for this example. Every step gives a reward of -1 and the step allowing to exit the maze gives a reward of 100. This last step is called goal and leads to a terminal state. The purpose of the reward signal is to lead our agent to the correct path.

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) are a framework widely adopted to describe the interaction between an agent and an environment. We consider discrete time MDPs in which at each time step the agent performs an action and observes how the environment evolves. It is extremely useful when we have to model a Reinforcement Learning problem. We can define an MDP (Puterman, 2014 [27]) as a tuple $\mathcal{M} := \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, where:

- $\mathcal{S}$ is the set containing the possible states of the environment;
- $\mathcal{A}$ the set of possible actions;
- $P : \mathcal{S} \times \mathcal{A} \to \mathscr{P}(\mathcal{S})$ is the Markovian transition kernel, a function that provides the probability of each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ to transit to the next state $s' \in \mathcal{S}$;
- $R : \mathcal{S} \times \mathcal{A} \to \mathscr{P}(\mathbb{R})$ is the reward distribution, a function that provides the probability of retrieving a certain reward after performing an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$;
- $\gamma \in [0, 1)$ is the discount factor. Used to modulate the importance of feature rewards;

We can restrict the reward space and define $\mathcal{R} \in \mathbb{R}$. $R(\cdot|s, a)$ is a distribution over state-action pair (s, a) with an expected value:

$$R_{s,a} = \mathbb{E}\big[R_{t+1} \mid s_t = s,\ a_t = a\big] = \int_{\mathcal{R}} rR(\mathrm{d}r \mid s, a),$$

where $R_{s,a}$ is bounded by $R_{max} < +\infty$ for each $s \in \mathcal{S}$ and $a \in \mathcal{A}$. We can also extend $R(\cdot|s, a)$ to include the next state reached by the agent $R(\cdot|s, a, s')$ and recover the first by marginalization. Now, we shortly describe how the process evolves.

Given an initial distribution of the states μ over the set $\mathcal{S}$, the agent starts in an initial

state $S_0 \sim \mu$. At each time step t the agent chooses an action $a \in \mathcal{A}$ to perform, extracted from a policy π, a distribution that selects an action based on the state of the environment. After that the state of the environment evolves to $s_{t+1} \sim P(\cdot|s_t, a_t)$ and receives a reward $r_t \sim R(\cdot|s_t, a_t)$.
Following this procedure (showed in figure 2.2) we can collect a trajectory $\mathcal{H}_t = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$. This trajectory will be used by Reinforcement Learning algorithms to model our problem.

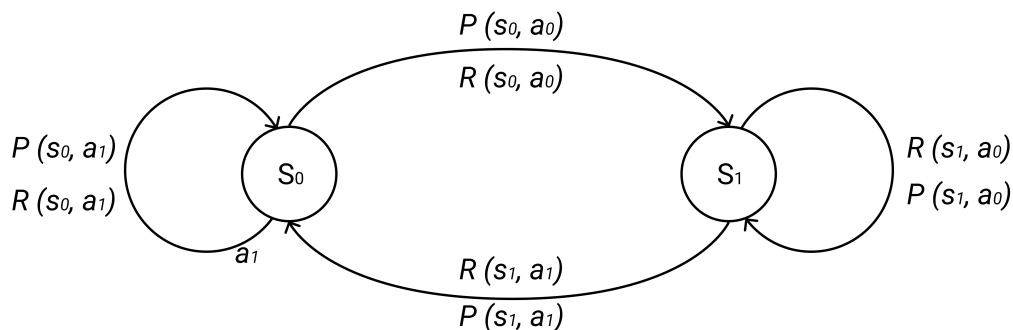


Figure 2.2: MDPs representation example. Each circle is a state and each arrow represent a transition

2.2.1 Policy and Value Function

As mentioned before the agent follows a policy π, which is described in this section. A policy π_t with $t \in \mathbb{N}$ defined at each time step is a map $\pi : \mathcal{H}_t \rightarrow \mathscr{P}(\mathcal{A})$, where $\mathcal{H}_t = (s_0, a_0, \dots, s_{t-1}, a_{t-1}, r_{t-1})$, used to chose an action to perform knowing the full history of the environment evolution. A policy can be deterministic when for each $\mathcal{H}_t$ the policy π_t chooses a fixed action or stochastic when π is in general a distribution over $\mathcal{A}$. If π_t depends only on the last state $s_t \in \mathcal{H}$ then it is also called Markovian. Moreover, if π^t does not depend on t it is stationary, in this case we remove the subscript t. We can also define Π as the set of all possible Markovian stationary policies over an MDP.

The goal of a large part of Reinforcement Learning algorithms is to estimate the value function for a state or a state-pair action. The value function estimates how good is for the agent to be in a state or to perform a specific action in a state and in general a policy induces a value function over the MDP. A first measure helpful to estimate the value function is the cumulative reward. We can start from a state s_t, collect all the feature rewards R_k with $k > t$ and write down:

$$G_t = \sum_{k=t+1}^{T} R_k, \tag{2.1}$$

where T can be ∞ considering $R_t = 0$ when t is greater than the end of the episode or in infinite-horizon environments. In general it is better to introduce discounted cumulative

reward to deal with continuing tasks, where the cumulative rewards can diverge for $T \to \infty$, as follow:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.2}$$

where γ is the parameter defined before. Thanks to the γ and the fact that $R_{s,a}$ is bounded we can prevent G_t to diverge. γ is used to modulate how much future information to incorporate into cumulative reward, $\gamma = 0$ means that we care only to the immediate reward and if we increase γ we can extend our attention to the future states.

Now we can define the value function of a state $s \in \mathcal{S}$ as follows:

$$V^\pi(s) = \mathbb{E}_\pi\big[G_t|s_t = s\big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\big|s_t = s\Big] \;\; \forall s \in \mathcal{S}\ , \tag{2.3}$$

and it represents the return that we can expect starting in a specific state and the following the policy π. We can also define an action-value function, called Q-function, as the expected return starting from a state s, taking an action a and then following the policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi\big[G_t|s_t = s, a_t = a\big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\big|s_t = s, a_t = a\Big] \;\; \forall s \in \mathcal{S}\ , \tag{2.4}$$

In a Reinforcement Learning task we usually want to find the best policy among all the possible ones. We can define a partial ordering over policies induced by the Value function. A policy π is better than or equal to π' if and only if $V_\pi(s) \geq V_{\pi'}(s)$ for every $s \in \mathcal{S}$. In the policy space we can find a policy where V^π is greater or at least equal than all others policies. We call that optimal policy, denoted π^*. For the optimal policy we can now define an optimal value function:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s), \tag{2.5}$$

and an optimal Q-function:

$$Q^*(s,a) = \max_{\pi \in \Pi} Q^\pi(s,a). \tag{2.6}$$

2.2.2 Bellman Operators

A fundamental property of value functions, exploited in most of Reinforcement Learning algorithms, is that they satisfy recursive relationships. For any policy π and any state s the consistency condition 2.7 holds between the value function evaluated on s and the value function of its possible successor states. The relation is used in a recursive way to

evaluate V^π, the value function of a generic policy π, to find the optimal policy π^* at a later time. The relation is the following:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\big[G_t|s_t = s\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} + \gamma G_{t+1}|s_t = s\big] \\ &= \sum_a \pi(a|s) \sum_{s',r} P(s'|s,a)R(r|s,a)[r + \gamma V^\pi(s')] \;\; \forall s \in \mathcal{S} \;, \end{aligned} \tag{2.7}$$

where $\pi(a|s)$ is the probability of taking an action a in state s under policy π. The property 2.7 is valid only with discrete states space $\mathcal{S}$, actions space $\mathcal{A}$ and rewards space $\mathcal{R}$. Thanks to this property we can build a system with $|S|$ equations and $|S|$ unknowns and solve it to find $V^\pi(s)$ (we refer to this operation as policy evaluation). Unfortunately, in order to do this, we first need to know the environment's dynamics, but this is impossible or extremely tedious in most cases. It is better to solve the policy evaluation in an iterative way. We can introduce an operator called Bellman Expectation Operator $T^\pi : \mathbb{R}^{|\mathcal{S}|} \rightarrow \mathbb{R}^{|\mathcal{S}|}$ that maps a value function to a value function as follow:

$$T^\pi V^\pi(s) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s'), \tag{2.8}$$

where $r \sim R(\cdot|s,a)$, $s' \sim P(\cdot|s,a)$ and $a \sim \pi(\cdot|s)$. We can also define the corresponding operator for the optimal value function, the Bellman Optimal Operator $T^* : \mathbb{R}^{|\mathcal{S}|} \rightarrow \mathbb{R}^{|\mathcal{S}|}$:

$$T^* V^*(s) = \max_{a \in \mathcal{A}} \big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s')\big]. \tag{2.9}$$

In a similar way we can define the Bellman Expectation Operator and the Bellman Optimal Operator for the Q function:

$$T^\pi Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s',a'), \tag{2.10}$$

$$T^* Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a'). \tag{2.11}$$

Both expectation and optimal operators are γ contraction with a specific fixed point. For the expectation operator the value function (and the state-action value function) of the policy considered is the fixed point. Instead, the optimal value function (and the optima state-action value function) is the fixed point of the optimal operator. This means that, if we iterate these operators for an appropriate amount of time t (ideally for $t \rightarrow \infty$), starting from a certain value function, we can converge to the value function of policy π, with the Bellman Expectation Operator, or to the optimal value function π^*, with the Bellman Optimal Operator.

2.2.3 Dynamic programming

Dynamic programming (Bellman, 1958 [4]) is a general technique for solving problems breaking them into simple sub-problems and then, for each one of them, computing and storing the solution to recombine them to solve the original one. Classical Dynamic Programming (DP) algorithms are limited in reinforcement learning because they need to know the exact model of our environment, but they provide the foundations for the rest of our speech.

We can distinguish two techniques that exploit dynamic programming:

- Policy Iteration
- Value Iteration

Policy iteration is an on-policy algorithm that takes advantage of the Policy improvement theorem (Richard S Sutton et al., 2018 [34]), where, given a policy π and the corresponding value function $V^{\pi}(s)$, the greedy policy π' derived from $V^{\pi}(s)$ is always better or equal to the original one. The greedy policy derived from a value function is defined as follow:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \sum_{s',r} P(s'|s,a)R(r|s,a)[r + \gamma V(s')]. \tag{2.12}$$

We can exploit this by iterating the procedure as showed in the figure 2.3.

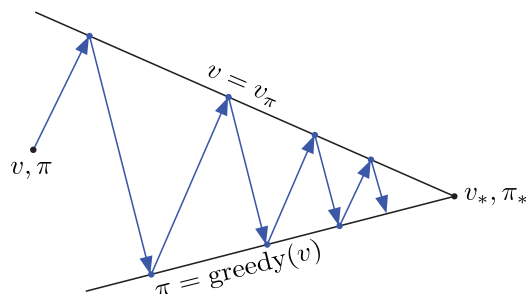


Figure 2.3: Policy Improvement process.

The first step is the policy evaluation where we can approximate the value function of a policy. We can call this part of the problem as policy evaluation. To approximate the value function we can follow the next procedure:

Algorithm 2.1 Iterative Policy Evaluation for estimating $V \approx V^\pi$

Input: Policy π. A small threshold $\theta > 0$ determining accuracy of estimation.
Output: V^π

1: **Initialize:** random $V(s)$, for all $s \in \mathcal{S}$ except $V(terminal) = 0$
2: **repeat**
3: $\quad\Delta \leftarrow 0$
4: $\quad$**for all** $s \in \mathcal{S}$ **do**
5: $\quad\quad v \leftarrow V(s)$
6: $\quad\quad V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} P(s'|s,a) R(r|s,a)[r + \gamma V(s')]$
7: $\quad\quad \Delta \leftarrow max(\Delta, |v - V(s)|)$
8: $\quad$**end for**
9: **until** $\Delta < \theta$

We can now alternate the policy evaluation and the policy improvement processes as in the following procedure:

Algorithm 2.2 Policy Iteration for estimating $\pi \approx \pi^*$

Output: A deterministic policy $\pi \approx \pi^*$. A small threshold $\theta > 0$ determining accuracy of estimation.

1: **Initialize:** random $V(s)$, for all $s \in \mathcal{S}$ except $V(terminal) = 0$
2: $V \leftarrow evaluate(\pi)$ $\quad\triangleright$ From Algorithm 2.1
3: **repeat**
4: $\quad V' \leftarrow V$
5: $\quad \pi \leftarrow greedy(V)$ $\quad\triangleright$ Build the greedy policy from value function V
6: $\quad V \leftarrow evaluate(\pi)$
7: $\quad \Delta \leftarrow max(\Delta, |V' - V|)$
8: **until** $\Delta < \theta$

Opposed to the previous algorithm we can find in the classical background the value iteration procedure that exploits the Bellman optimal operator to directly approximate the value function of the best policy.

Algorithm 2.3 Value Iteration for estimating $\pi \approx \pi^*$

Input: A small threshold $\theta > 0$ determining accuracy of estimation.
Output: A deterministic policy $\pi \approx \pi^*$

1: **Initialize:** random $V(s)$, for all $s \in \mathcal{S}$ except $V(terminal) = 0$
2: **repeat**
3: $\quad \Delta \leftarrow 0$
4: $\quad$ **for all** $s \in \mathcal{S}$ **do**
5: $\quad\quad v \leftarrow V(s)$
6: $\quad\quad V(s) \leftarrow \max_a \sum_{s',r} P(s'|s,a)R(r|s,a)[r + \gamma V(s')]$
7: $\quad\quad \Delta \leftarrow max(\Delta, |v - V(s)|)$
8: $\quad$ **end for**
9: **until** $\Delta < \theta$

2.3 Model-free prediction

The dynamics of the environment are very often unknown, in other cases we have to deal with a large-dimensional spaces, these conditions make dynamic programming infeasible. In these cases it is useful to introduce model-free algorithms. In the next sections we will talk about two basic and simple methods for Value function approximation that do not require a full MDP model.

2.3.1 Monte Carlo

Monte Carlo (MC) methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results, so they could be useful in Reinforcement Learning where we can learn directly from experience. MC algorithms exploit the simple idea that the mean return is a consistent estimator for the Value function. To estimate the value function of a state s we can start from the state itself and follow the selected policy π and use the empirical mean return instead of the expected return. We can develop two type of Monte Carlo's update: first-visit and every-visit. In Monte Carlo first-visit we average returns only for the first visit of state s. In Monte Carlo every-visit we average returns for every visit of state s.

Algorithm 2.4 First-visit MC prediction for estimating $V \approx V^\pi$

Input: A policy π to be evaluated

Output: $V \approx V^\pi$

1: **Initialize:** random $V(s)$, for all $s \in \mathcal{S}$
2: $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
3: **for all** episode **do**
4: Collect the entire history $\mathcal{H}_T = (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, r_T)$ of the episode
5: $G \leftarrow 0$
6: **for all** $t = T-1, T-2, \ldots, 0$ **do**
7: $G \leftarrow \gamma G + r_{t+1}$
8: **if** $s_t \notin (s_0, s_1, \ldots, s_{t-1})$ **then**
9: Append G to $Returns(s_t)$
10: $V(s_t) \leftarrow average(Returns(s_t))$
11: **end if**
12: **end for**
13: **end for**

2.3.2 Temporal Difference

One of the drawbacks of Monte Carlo methods for Reinforcement Learning is that we need to wait the end of an episode to estimate the Value function. If the episode length is very long or we have to face a non episodic environment we cannot use Monte Carlo Algorithms. One possible solution is based on Temporal Difference (TD). Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update the estimate of Value function using both raw samples and the estimated Value function itself computed on another state (this is usually called *bootstrapping*). Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience generated by a policy, both methods update their estimate V of V^π for the non terminal states s_t occurring in that experience. The update rule is the following:

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\Big], \tag{2.13}$$

where α is a parameter between 0 and 1 that tells us how we want to trust new values for value function estimation. In a typical scenario we want to decrease α with the increase of time step, for example we usually have $\alpha = \frac{1}{N(s_t)}$, where $N(s_t)$ is the number of times that state s_t has been visited.

Algorithm 2.5 Tabular TD for estimating $V \approx V^\pi$

Input: A policy π to be evaluated. Learning rate α

Output: $V \approx V^\pi$

1: **Initialize:** random $V(s)$, for all $s \in \mathcal{S}$ except $V(terminal) = 0$
2: **for all** episode **do**
3: **repeat**
4: $a \leftarrow$ action given by π for state s
5: Take action a, observe r, s'
6: $V(s) \leftarrow V(s) + \alpha\Big[r + \gamma V(s') - V(s)\Big]$
7: $s \leftarrow s'$
8: **until** s is terminal
9: **end for**

2.4 Model-free control

We can subdivide the main approaches to learn optimal policy in two categories. On one hand, Value based approaches attempt to learn π^* through a representation of Q, trying to improve the policy at each iteration, similarly to what we have seen previously. On the other hand, policy based methods do not pose their focus on learning Q, but instead policy π is directly parametrized and can be learnt by gradient descent with respect to the expected return. In this work we will focus on Value based approaches.

The part of Reinforcement Learning with the aim of learning an optimal policy is called *control* and we can identify two types of algorithms:

- On-policy algorithms: they evaluate or improve a policy π using the policy itself to choose the action and generate the data.
- Off-policy algorithms: they evaluate or improve a policy π using a different policy π' to choose the action and generate the data.

SARSA and Q-Learning are two examples of respectively on-policy and off-policy RL algorithms. Both of them are state-action value based algorithms that estimate target values using Temporal Difference. We are going to introduce them in the next sections.

2.4.1 SARSA

SARSA (State-Action-Reward-State-Action) 2.6 is an on-policy implementation of TD learning where the target value for the update using the sum of the actual reward and

the Q value, where the last is evaluated in the next state visited and in the next action selected from the current policy π. The update rule for this algorithm is the following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma Q(s',a') - Q(s,a)\Big]. \tag{2.14}$$

The full procedure is the following:

Algorithm 2.6 Sarsa (on-policy TD control) for estimating $Q \approx Q*$

Input: step size $\alpha \in (0,1]$, small $\epsilon > 0$
Output: $Q \approx Q*$
1: **Initialize:** random $Q(s,a)$, for all $s \in \mathcal{S}$ except $Q(terminal, \cdot) = 0$
2: **for all** episode **do**
3: Choose a from s using policy derived from Q
4: **repeat**
5: Take action a, observe r, s'
6: Choose a' from s' using policy derived from Q
7: $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma Q(s',a') - Q(s,a)\Big]$
8: $s \leftarrow s'$
9: $a \leftarrow a'$
10: **until** s is terminal
11: **end for**

2.4.2 Q-learning

Q-learning is an off-policy algorithm, where the update rule uses a policy π' derived from the one we want to improve. In particular we use an *ε-greedy policy* defined as follow:

$$\pi(a|s) = \begin{cases} \dfrac{\varepsilon}{|\mathcal{A}|} + 1 - \varepsilon & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q(s,a') \\ \dfrac{\varepsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{2.15}$$

Moreover, it is common to initialize ε to an high value in the range $[0,1]$ and then decrease it at each episodes. It is important to point out that the expression 2.15 is valid only with a discrete set of actions.

Now, we can define the update rule used with Q-learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max Q(s',a) - Q(s,a)\Big]. \tag{2.16}$$

The algorithm of Q-learning is the following:

Algorithm 2.7 Q-learning (off-policy TD control) for estimating $\pi \approx \pi*$

The pseudo code of the algorithm is the following:

Input: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Output: $Q \approx Q^*$

1: **Initialize:** random $Q(s, a)$, for all $s \in \mathcal{S}$ except $Q(terminal, \cdot) = 0$
2: **for all** episode **do**
3: **repeat**
4: Choose a from s using policy derived from Q
5: Take action a, observe r, s'
6: $Q(s, a) \leftarrow Q(s, a) + \alpha\Big[r + \gamma \max Q(s', a) - Q(s, a)\Big]$
7: $s \leftarrow s'$
8: **until** s is terminal
9: **end for**

In theory, both methods Q-learning and SARSA converge to an exact estimate of $Q^*(s, a)$ when every pair state-action is visited an infinite number of times. However, in practice, the learners reach optimal performance despite not meeting these requirements.

2.5 Function approximation

In the previous sections we presented some of the basic algorithms for reinforcement learning, anyway those algorithms do not suit real applications. In general they cannot be used when the state-action space is continuous, for obvious reasons, or when it is too large to manage, due to memory and time limitations. Instead it is useful to introduce function approximation to represent the Value function or the policy itself. We have to introduce the model of functions $Q(s, a; \boldsymbol{\theta})$ depending on vector parameter $\boldsymbol{\theta}$ and inputs s and a. Moreover, Q needs to be differentiable with respect θ. Varying the values of $\boldsymbol{\theta}$ we can obtain a good approximation of the optimal Q-function. To find the optimal parameters we have first to introduce a loss function:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{s,a\sim\pi}\big[(\widetilde{Q}(s, a) - Q(s, a; \boldsymbol{\theta}))^2\big]. \tag{2.17}$$

Where $L(\boldsymbol{\theta})$ is the *mean squared error* (MSE) used commonly in a lot of Machine Learning problems. $\widetilde{Q}(s, a)$ is the target value, calculated from the samples collected directly from the policy π. The basic online approach to minimize $L(\boldsymbol{\theta})$ is to periodically update the parameters using *Stochastic Gradient Descent* (SGD) with the following equation:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha(\widetilde{Q}(s, a) - Q(s, a; \boldsymbol{\theta}^t))\nabla Q(s, a; \boldsymbol{\theta}^t)\big]. \tag{2.18}$$

When approximation is used, convergence properties are not guaranteed, especially with off-policy methods. Function approximation is extremely useful when we combine it with Deep Learning (LeCun et al., 2015 [16]), a learning paradigm which is characterized by the use of neural networks to represent Value function.

2.6 Neural Networks

Neural networks, also known as artificial neural networks (ANNs), are a subset of machine learning area and the main tools used in deep learning algorithms (Lippmann, 1987 [17]). It is a mathematical model where the simplest computational unit is called *neuron.* A neuron is characterized by n input x_i with $i = 0, 1, \cdots, n-1$ and a weight w_i, corresponding to each input (Figure 2.4). Each neuron calculates the sum of each value multiplied by its respective weight and adds a bias value b. After that, it applies an activation function a and produces an output y, that constitutes the output of the neuron.

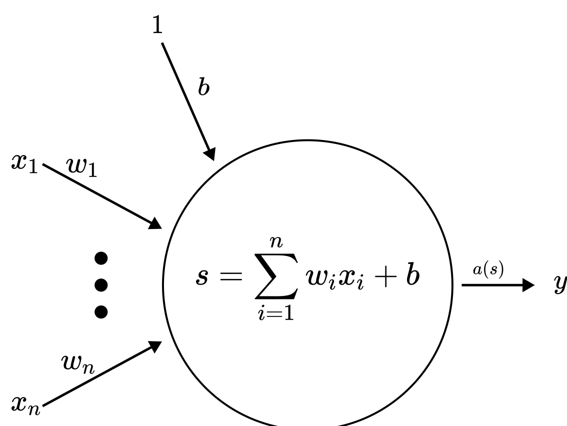


Figure 2.4: Neuron representation.

A single neuron has a limited expression power, so multiple neurons are usually organized in layers and we can build complex networks connecting layers in series. This structure is called *Multi Layer Perceptron* (MLP). One of the main properties of the MLP is that it is an universal function approximator. This means that, even using a simple three-layers structure, a MLP is able to approximate any possible function, given that the hidden layer features a sufficient number of nodes. While this is a theoretical guarantee, finding the right set of weights using only one layer of abstraction might require indefinitely long time. For this reason, it is common to use MLPs with a higher number of hidden layers, which guarantees a progressively deeper level of abstraction. For example, one of the main structures to build a MLP is composed by *Fully Connected*

Layer. In this type of structure, each neuron receives as input all the outputs produced by the previous layer. We can stack Fully Connected Layers with different sizes together to build a more complex networks (example in Figure 2.5). Neural networks like this are commonly used as function approximators in a wide range of problems, for example with reinforcement learning. As mentioned before, after the output of a neuron, a function approximation a is applied. We need to remark that a needs to be a non linear function. With a linear activation function in the network, no matter how many layers it had, the neural network would behave just like a single-layer perceptron. An example of activation function is the ReLU (Rectified Linear Unit), defined as:

$$ReLU(x) = \max(0, x) \tag{2.19}$$

The SGD technique is performed through *back-propagation* (Rumelhart et al., 1986 [28]). The network is trained over a set of data points of the function they are supposed to approximate. These data points contain the function input, used to compute the network output, and the known function output. The two outputs, the real one and the one computed by the network, are compared; then the back-propagation algorithm computes the gradient of the network, starting from the output of the last layer and going back to the input layer, based on the magnitude of the error between the prediction, output of the network, and the real value. Among the main problems of this technique we have *vanishing gradient*. At each step of the back-propagation, the gradient may became smaller and smaller as it is propagated back to the start of the network, in this case only the final part of the network is trained.

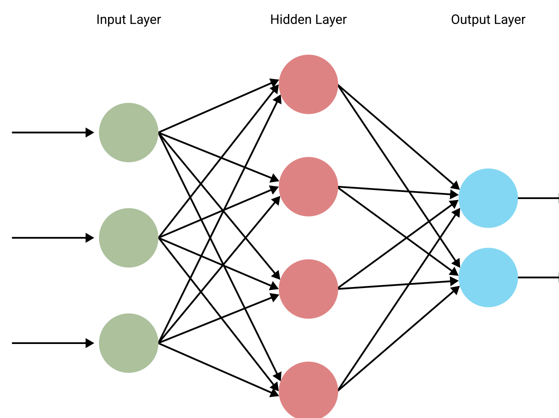


Figure 2.5: Fully connected network structure example.

We also have *Convolutional Neural Networks* (CNNs), which are particularly suitable for dealing with images. They use another type of layer called *convolutional layer*, typically used to recognize spatial patterns present in the input. The main operation of this type of network is the mathematical operation called convolution. This operation

is performed between two matrices (see Figure 2.6), an input matrix, representing for example a two dimensional image, and a kernel matrix, usually smaller than the input one, representing the filter. Formally, the convolution operation can be expressed as follows:

$$Y_{i,j} = [X \otimes W](i,j) + b = \sum_{\alpha=1}^{f} \sum_{\beta=1}^{f} X(s \times i + \alpha, s \times j + \beta) W(\alpha, \beta) + b,$$

$$with\ (i,j) \in \{0, 1, \cdots, L\},\ and\ L = \frac{L_0 + 2p - f}{s} + 1$$

where X is the input matrix , W is the convolutional parameters matrix of the kernel filter, b is the bias parameter, f corresponds to convolutional kernel size, Y is the output of the neuron, L is the dimension of the output, L_0 is the dimension of the input. Instead, s and p are two parameters called stride and padding. The stride correspond to the number of cells by which the center of the filter is moved from one convolution to the next one. The padding is used to add blank or empty cells to the frame of the input matrix for a minimized reduction of size in the output layer. In a convolutional layer the elements of the kernel matrix W and the bias parameters b are the weights that our algorithm needs to optimize.

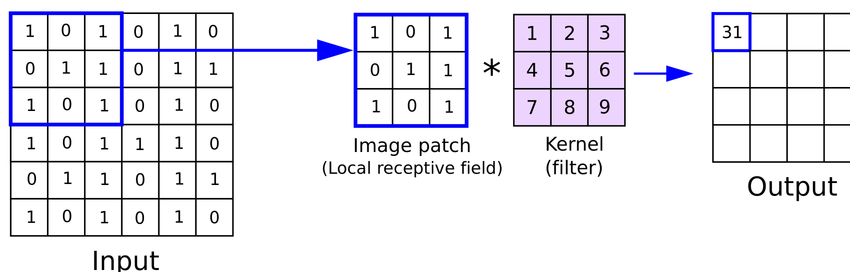


Figure 2.6: Example of convolution operation used by CNNs

2.6.1 Deep Learning

Deep Learning is a branch of machine learning that exploits the power of neural networks to reduce high-dimensional data in a compact and abstract representation. This is extremely useful when we need to recognize patterns and extract features from different kind of data. For example it is used in different fields as image analysis, automatic speech recognition and natural language processing. One of the first attempts to mix Deep Learning with Reinforcement learning goes back to 2013 when DeepMind developed the Deep Q-Learning (DQN) algorithm (Mnih, Kavukcuoglu, Graves, et al., 2013 [20]), using the convolutional neural networks to overcome human scores in different Atari games.

2.6.2 Deep Q-Learning

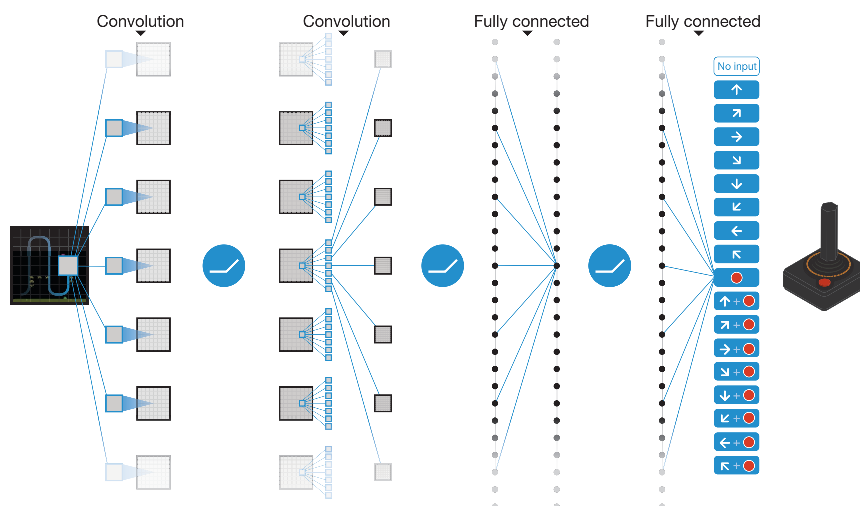


Figure 2.7: Deep Q-network presented by Mnih, Kavukcuoglu, Silver, et al., 2015 [21]

Deep Q-Learning is an extension of the online Q-learning algorithm with the introduction of state-value function approximation through neural network. It exploits a *replay buffer* to store the samples as tuples $\langle s_t, a_t, r_t, s_{t+1} \rangle$ collected during the exploration. When we have to train the network we can randomly extract the tuples from the buffer and use them to construct the target value of our Q-function. We can write the target value as follows:

$$\widetilde{Q}(s,a) = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}^t). \tag{2.20}$$

Where $\boldsymbol{\theta}^t$ represents the parameters (the weight of the neural network) that we need to train. Replay buffer is really useful to reduce correlation between samples in the same batch, since SGD assumes the samples in a mini-batch to be statistically independent. After the original paper of DQN, different features were added to improve performance:

- Prioritized Replay Buffer
- Target network
- Double Deep Q-network (DDQN)
- Dueling networks

With *prioritized replay buffer* (Schaul et al., 2015 [29]) the samples have different weights based on the last Temporal Difference Error (TDE) computed in the previous update:

$$\delta_i = r_i + \gamma \max_{a'} Q(s_i', a'; \boldsymbol{\theta}) - Q(s_i, a_i; \boldsymbol{\theta}). \tag{2.21}$$

The key idea is to give more importance to samples with highest errors computed during the last step of SGD.
In tabular Q-Learning algorithm, the target value of an update is an estimate computed through bootstrapping. When we add function approximation we can have instability caused by the oscillations of the target values, so it is helpful to use a *target network*, a simple copy $\widehat{Q}$ of the online network, to compute the current target value. The new rule is the following:

$$\widetilde{Q}(s,a) = r + \gamma \max_a \widehat{Q}(s', a; \boldsymbol{\theta}^-). \tag{2.22}$$

A new parameter t_{copy} is introduced to copy the updated Q-function to the target Q-function every time step t_{copy}.
In Q-learning and DQN, the *max* opeartor uses the same values to both select and evaluate an action. This can therefore lead to overoptimistic value estimates. To mitigate this problem, *Double Deep Q-network* is introduced (Van Hasselt et al., 2016 [37]) as a double estimator. We update the target value as follows:

$$\widetilde{Q}(s,a) = r + \gamma \widehat{Q}(s', \arg\max_{a'} Q(s', a'; \boldsymbol{\theta}^t); \boldsymbol{\theta}^-), \tag{2.23}$$

where $\boldsymbol{\theta}^-$ represents the parameters of the target network and the Q used with the $\arg\max$ operation is the double one. Another important features is the *dueling network* (Wang et al., 2016 [38]). This feature exploits the advantage function $A^\pi(s,a)$ defined as follows:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{2.24}$$

This new value represents the advantage of performing an action a in a state s compared to the expected return by directly following π from s. The Advantage function and the Value function are estimated separately and combined to estimate Q^π. With this method we are able to capture more specific features. The new network structure is presented in the bottom part of figure 2.8. The first part of the network is composed by shared layers. The output of the shared part is then used as input for the group of layers that evaluates A and the group that represents V. The operation computed by the network is the following:

$$Q^\pi(s,a;\boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{\beta}) = V^\pi(s;\boldsymbol{\theta},\boldsymbol{\beta}) + \left(A(s,a;\boldsymbol{\theta},\boldsymbol{\alpha}) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\boldsymbol{\theta},\boldsymbol{\alpha}) \right), \tag{2.25}$$

where $\boldsymbol{\theta}$ represents the parameters of the shared layers, $\boldsymbol{\alpha}$ the parameters of the layers used to calculate A and $\boldsymbol{\beta}$ the parameters for the V layers.

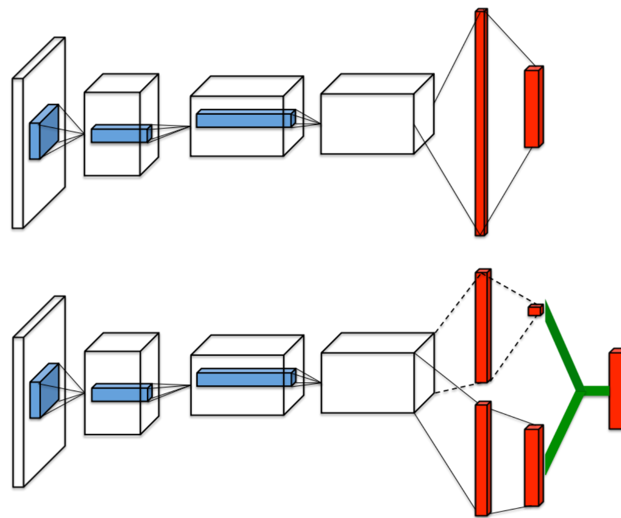


Figure 2.8: A popular single stream Q-network (top) and the dueling Q-network (bottom) as presented in Wang et al., 2016 [38].

Chapter 3

State of the Art

We recall that the aim of this project is the development of a dynamic action repetition system with the purpose to unlock a dynamic control frequency. In this chapter we introduce different works that discuss about action repetition. In Section 3.1 we talk about methods that deal with a fixed action repetition settings. Going forward, we introduce, in Section 3.2, different proposed solution to implement a dynamic action repetition.

3.1 Static Methods

Action Repetition arises naturally in real life as well as simulated environments. The time scale of executing an action enables agents (both humans and AI) to decide the granularity of control during task execution. Classical methods used in most framework use a static action repetition paradigm, wherein the action decided by the agent is repeated for a fixed number of time steps regardless of the contextual state or action while executing the task.

For example (Braylan et al., 2015 [6]) showed the performance of different frame skip in Atari games through the Arcade Learning Environment (ALE). In ALE framework games run at sixty frames per second, and agents can submit an action at every frame, but it is also possible to introduce a frame skip parameter as the number of frames an action is repeated before a new action is selected. As mentioned by the paper, different implementations of common algorithm, such as HNEAT, DQN or SARSA, usually used a different frame skip. If we increase this parameter, we can significantly decrease the time it takes to simulate an episode, at the cost of missing opportunities that only exist at a finer resolution. The authors implemented a variant of Enforced Sub-populations (ESP) (Gomez et al., 2005 [11]) in which they can change the frame skip through the ALE platform. In some games higher frame skip values lead to a very better performance.

Another work about the action repetition as an hyper-parameter useful to improve performance in a reinforcement learning context is Metelli et al., 2020 [19]. Here authors have introduced the notion of action persistence, defined by the parameter k, as the repetition of an action for a fixed amount of steps to obtain a different control frequency. At first they described theoretically the implication of action persistence. The execution of a Markovian stationary policy π at persistence $k > 1$ produces a behavior that, in general, cannot be represented by executing any Markovian stationary policy at persistence 1, because, at each time step, the described policy needs to remember the previous $t-1$ steps and decide if it has to decide a new value or to choose the previous one (so it is non-Markovian and non-stationary). This leads to a definition of a new k-persistent policy induced by a classical Markovian stationary policy π as follow: $\pi_{t,k}$ is equal to π if $t \bmod k = 0$ or it is equal to a Dirac distribution centered in the previous action. This means that this policy has the same behavior of the original one when the time step is a multiple of the persistence and it chooses the previous selected action otherwise. From another point of view we can define a modified MDP that incorporates the concept of persistence transition creating a new transition probability kernel. As shown in figure 3.1 there is a connection between the original MDP solved by a k-persistent policy and the new k-persistent MDP solved by the original policy.

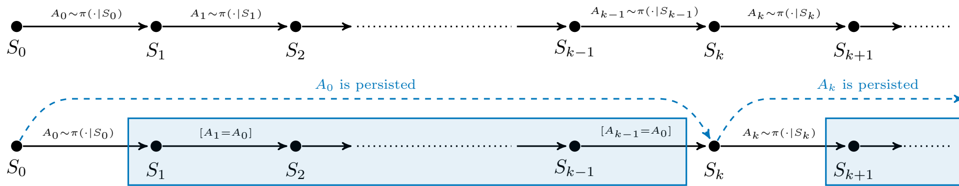


Figure 3.1: Agent-environment interaction without (top) and with (bottom) action persistence, highlighting duality. The transition generated by the k-persistent MDP $\mathcal{M}_k$ is the cyan dashed arrow, while the actions played by the k-persistent policy are inside the cyan rectangle.

They introduced a new Persistent Bellman Operators, constructed from the original Bellman operator, and implemented the Persistent Fitted Q-Iteration, an extension of Fitted Q-Iteration, able to approximate the value function at a given persistence. The experimental results obtained in the paper shown that the introduction of persistence could be a good choice since reducing the control frequency can lead to an improvement when dealing with a limited number of samples. We will use deeply the notion of persistence described by Metelli et al., 2020 [19] to introduce our work in chapter 4. We also need to remark, as stated by authors, that persisting an action, for a fixed amount of time, is a particular instance of a semi-Markov *option*, always lasting k steps. A semi-Markov Decision Process is an extension of MDP formalism that deals with temporally extended actions and/or continuous time. An option is a generalization of primitive actions to include temporally extended courses of action. According to the flat option representation, an option needs three components: a policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$, a

termination condition $\beta : \mathcal{S} \to [0, 1]$, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$. Given a state s, an option $\{\mathcal{I}, \pi, \beta\}$ is available if and only if $s \in \mathcal{I}$. Once we select an option, we follow policy π and check at each step if the option is stochastically terminated according to β. If the option is terminated, the agent has the possibility to select a new option.

One of the main benefits of action persistence is the exploration. The exploration is one of the most important open problems in reinforcement learning. The most commonly used exploration strategies are simple methods like $\epsilon-$greedy, Boltzmann exploration and entropy regularization. These methods are general because they do not rely on strong assumptions about the underlying domain, so they don't require too much implementation effort or per-domain fine tuning. Among these works, Dabney et al., 2020 [8] introduced an ϵz-greedy exploration, with a random exploratory variable deciding the duration of each action. they have replaced actions with temporally-extended sequence of actions, described as options. They ended with a new exploration method called temporally-extended $\epsilon z-$greedy exploration. This strategy depends on choosing an exploration probability ϵ, a set of options Ω, and a sampling distribution p with support Ω. Then, on each step the agent follows the current policy π for one step with probability $1-\epsilon$, or with probability ϵ samples an option $w \sim p(\Omega)$ and follows it until termination. The authors showed, both in tabular and deep reinforcement learning, that the new introduced method can improve exploration and performance in sparse-reward environments.

3.2 Dynamic Methods

In this section we talk about different solutions to obtain a dynamic action repetition, i.e. the ability of the agent to select different repetition rate in a single run.

3.2.1 Tabular Methods

There are few techniques that try to achieve a dynamic time scale in an MDP framework, some of these use the option approach (Richard S. Sutton et al., 1999 [35]; Mankowitz et al., 2014 [18]), some others use hierarchy of abstract machines (HAM) (Parr et al., 1998 [24]) or the MAXQ approach (Dietterich, 1998 [10]). All these methods are based on decomposition of actions into sub-tasks, so they need to know in advance, totally or partially, the model and the effects of each action. One of the first attempts to introduce action persistence, without the creation of options and sub-tasks, goes back to 2003 (Schoknecht et al., 2003 [30]). Here the authors tried to integrate the concept of temporal abstraction in the reinforcement learning framework to improve scalability. One of the concerns of the authors was that standard reinforcement learning algorithms,

like base Q-learning, scale very badly with the increase of problem size. One intuitive reason for this is that according to the problem size, the number of decisions from the start state to the goal state increases. If our agent is able to reduce the number of decision that are necessary to reach the goal, learning could be accelerated. The solution proposed, called MSA (multi-step action), extends classical RL algorithms proposing the concept of explicitly selecting time scale by just choosing the repetition of the action while choosing the action itself.

3.2.2 Neural Networks

More recent works tried to exploit neural networks to extend the state-action value function, or common policy gradient methods, to include also the concept of action repetition, with the goal of making the most effective use of samples collected at different persistences. Among these works, Lakshminarayanan et al., 2017 [14] introduced the idea of enlarging the action space, duplicating actions and paring them with a specific repetition value. Authors implemented this framework with two well-known Deep RL algorithms, DQN and A3C, and used them mainly on Atari games. The authors stated that the key motivation behind the Dynamic Action Repetition paradigm is the observation that when humans execute tasks (such as playing games), the actions tend to be temporally correlated and almost always elongated. They implemented the Augmented DQN (shown in figure 3.2) duplicating the last layer of the network with the output representing the Q-values for actions at two different repetition rates. The main drawback is that repetition rates are hyper-parameters, hence there is no automatic adaptation of frequency.

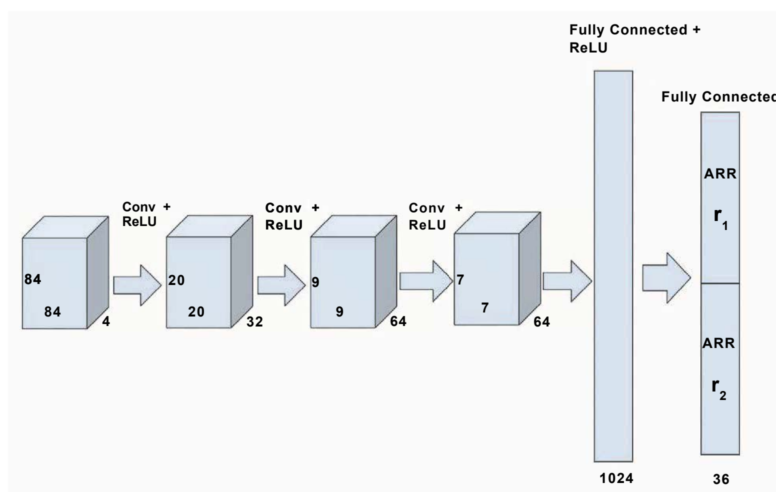
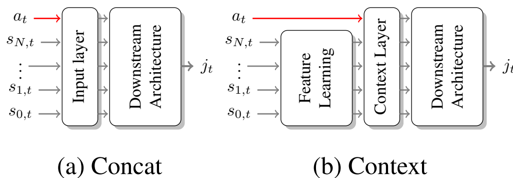


Figure 3.2: Network used in Dynamic action repetition for deep reinforcement learning

The new parameter introduced, called *action repetition rate* (ARR), denotes the number of times an action selected by the agent is repeated. If the ARR is low, the decision making frequency of the agent is high. This leads to policies which evolve rapidly in space and time. On the other hand, a high ARR causes infrequent decisions, which reduce the time to train at the cost of losing fine grained control, but on the other hand a reduction in decision making frequency gives the agent the advantage of not having to learn the control policy in the intermediate states. Such skills would be useful for the agent in games where good policies require some level of temporal abstraction. An example of this situation is Seaquest (an Atari 2600 game) where the agent has to continually shoot at multiple enemies arriving together and at the same depth, one behind another. In the case of Space Invaders (Atari 2600 game), the lasers are not visible in every fourth frame and a higher action repetition could help the agent avoid the confusion of having to decide an action in such peculiar intermediate states, where the lasers are not visible. Another approach (Sharma et al., 2017 [31]) introduces the concept of *skip network*, a second network used to choose the action repetition jointly with the original network (authors proposed mainly policy gradient methods such A3C, TRPO or DDPG). The main drawback of this implementation is that the second network is not action dependent, meaning that the action repetition chosen in a specific state is an average among all actions. This leads the agent to perform worst in states where one action is impulsive and the other one has a really big action repetition value.

One way to differentiate actions is introduced in a similar fashion with TempoRL (Biedenkapp et al., 2021 [5]). Here, two different networks are employed: while the base one is a normal DQN, the skip network depends on both state and action and approximates Q-values for different possible frequencies. The process is the following: the first network is used to choose the action to perform in a state, the state-action pair is then fed to the second network to choose skip value i.e. the repetition rate. To update the Q function at each skip value they use the cumulative discounted reward. Figure 3.3 and 3.4 show the proposed architecture.



(a) Concat (b) Context

Figure 3.3: Schematic representations of considered architectures for learning when to make decisions, where a_t is the action coming from a separate behaviour policy

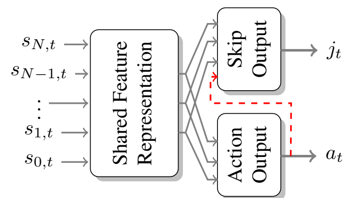


Figure 3.4: Architecture with shared feature representation for joint learning of when to make a decision and what action to take.

Another approach is used by Yu et al., 2021 [40], where authors have introduced a secondary binary policy, with the main purpose of choosing whether to repeat the previous action or to change it according to the principal agent. They have called this method Temporally Abstract Actor-Critic (TAAC). The main network (they have extended a policy gradient method) is used to choose a new action at each state, but this is not taken until a second network has decided if it is more convenient to repeat the previous action or to choose a new one (see figure 3.5 for reference).

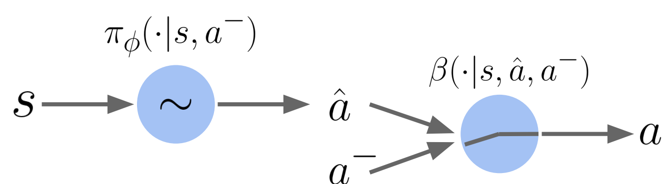


Figure 3.5: TAAC's two-stage policy during inference. In the first stage, an action policy π_ϕ samples a candidate action $\hat{a}$. In the second stage, a binary switching policy β chooses between this candidate and the previous action a^-

Chapter 4

Problem Formulation

In this chapter we explore what we mean with action persistence, how it is possible to compare with existing approaches and our solution to exploit all the samples collected during training of our agent through a new defined operator extended from the optimal Bellman operator.

4.1 Action Persistence

As mentioned in previous chapters, by action persistence we mean the repetition of an action $a \in \mathcal{A}$ starting from a specified space $s \in \mathcal{S}$ for a specified amount of time step $k \in \mathbb{N}$. The static repetition of the same action can be modeled as a k-persistent MDP (extensively discussed in the paper Metelli et al., 2020 [19] introduced in chapter 3), starting from the original MDP. This newly proposed MDP $\mathcal{M}_k = \left(\mathcal{S}, \mathcal{A}, P_k, R_k, \gamma^k\right)$ is characterized by the k-persistent transition model P_k and reward function R_k, defined for any measurable sets $\mathcal{B} \subseteq \mathcal{S}$, $\mathcal{C} \subseteq \mathbb{R}$ and state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ as follows:

$$P_k(\mathcal{B}|s,a) = \left((P^{\delta})^{k-1}P\right)(\mathcal{B}|s,a), \tag{4.1}$$

$$R_k(\mathcal{C}|s,a) = \sum_{i=0}^{k-1} \gamma^i \left((P^{\delta})^{i}R\right)(\mathcal{C}|s,a), \tag{4.2}$$

where P^{δ} is the state-action persistent transition probability kernel of the original MDP $P^{\delta} : \mathcal{S} \times \mathcal{A} \to \mathscr{P}(\mathcal{S} \times \mathcal{A})$ defined as:

$$(P^{\delta})(\mathcal{B}|s,a) = \int_{\mathcal{S}} P(\,\mathrm{d}s'|s,a)\delta_{(s',a)}(\mathcal{B}), \tag{4.3}$$

where $\delta_{(s',a)}(\mathcal{B})$ is the Dirac measure centered in (s', a). We can also define an expected reward $r_k(s, a)$ bounded by $R_{\max}\frac{1-\gamma^k}{1-\gamma}$:

$$r_k(s,a) = \int_{\mathbb{R}} x R_k(\,\mathrm{d}x|s,a) = \sum_{i=0}^{k-1} \gamma^i \left((P^\delta)^i r\right)(s,a). \tag{4.4}$$

If from one side this model is easy to use and analyze, from the other side is constructed with a fixed predefined persistence k. For this purpose we need to model the persistence as an option, this leads us to the creation of the all-persistence Bellman operator, a new operator used to achieve a dynamic persistence.

4.1.1 Persistence as Options

We formalize the decision process in which the agent chooses a primitive action a together with its persistence k. To this purpose, we introduce the persistence option.

Definition 4.1.1. *Let $\mathcal{A}$ be the space of* primitive *actions of an MDP $\mathcal{M}$ and $\mathcal{K} := \{1, \dots, K_{\max}\}$, where $K_{\max} \geq 1$, be the set of persistences. A* persistence option *$o := (a, k)$ is the decision of playing primitive action $a \in \mathcal{A}$ with persistence $k \in \mathcal{K}$. We denote with $\mathcal{O}^{(k)} := \{(a, k) : a \in \mathcal{A}\}$ the set of options with fixed persistence $k \in \mathcal{K}$ and $\mathcal{O} := \bigcup_{k\in\mathcal{K}} \mathcal{O}^{(k)} = \mathcal{A} \times \mathcal{K}$.*

The decision process works at follows. At time $t = 0$, the agent observes $s_0 \in \mathcal{S}$, selects a persistence option $o_0 = (a_0, k_0) \in \mathcal{O}$, observes the sequence of states $(s_1, \dots, s_{k_0})$ generated by repeating primitive action a_0 for k_0 times, i.e., $s_{i+1} \sim P(\cdot|s_i, a_0)$ for $i \in \{0, \dots, k_0 - 1\}$, and the sequence of rewards $(r_1, \dots, r_{k_0})$ with $r_{i+1} = r(s_i, a_0)$ for $i \in \{0, \dots, k_0 - 1\}$. Then, in state s_{k_0} the agent selects another option $o_1 = (a_1, k_1) \in \mathcal{O}$ and the process repeats. During the execution of the persistence option, the agent is not allowed to change the primitive action.

Remark 4.1.1. (**Persistence and Options**) *The persistence option (Definition 4.1.1) is in all regards a* semi-Markov option *Precup, 2001 [26], where the initiation set is the set of all states $\mathcal{S}$, the termination condition depends on time only, and the intra-option policy is a constant policy. Indeed, the described process generates a* semi-Markov decision process *(Puterman, 2014 [27]), which is fully determined by the behavior of $\mathcal{M}$, as shown in Richard S. Sutton et al., 1999 [35].*

Remark 4.1.2. (**Persistence Options vs Augmented Action Space**) *There is an important difference between using the persistence options $\mathcal{O}$ in the original MDP $\mathcal{M}$ and defining an augmented MDP $\mathcal{M}_{\mathcal{K}}$ with new action space $\mathcal{A} \times \mathcal{K}$ and properly redefined transition model and reward function Lakshminarayanan et al., 2017 [14]. Indeed, when*

executing a persistence option $o_t = (a_t, k_t) \in \mathcal{O}$ from time t, we observe the full *sequence of states $(s_{t+1}, \dots, s_{t+k_t})$ and rewards $(r_{t+1}, \dots, r_{t+k_t})$. Instead, in the augmented MDP $\mathcal{M}_{\mathcal{K}}$ we* only *observe the last state s_{k_t} and the cumulative reward $r^k_{t+1} = \sum_{i=0}^{k-1} \gamma^i r_{t+i+1}$. We will heavily exploit the particular option structure, re-using fragments of experience in order to perform intra-option learning.*

We now extend the policy and state-action value function definitions to consider this particular form of options. A *Markovian stationary policy over persistence options* $\psi : \mathcal{S} \to \mathscr{P}(\mathcal{O})$ is a mapping between states and probability measures over persistence options. We denote with Ψ the set of the policies of this nature. The state-option value function $Q^\psi : \mathcal{S} \times \mathcal{O} \to \mathbb{R}$ following a policy over options $\psi \in \Psi$ is defined as $Q^\psi(s, a, k) := \mathbb{E}_\psi\big[\sum_{t=0}^{+\infty} \gamma^t r_{t+1} | s_0 = s, a_0 = a, k_0 = k\big]$. In this context, the optimal action-value function is defined as: $Q^\star_{\mathcal{K}}(s, a, k) = \sup_{\psi \in \Psi} Q^\psi(s, a, k)$.

4.1.2 All-Persistence Bellman Operator

We start by defining a $\overline{\kappa}$-persistence transition as $(s, s', a, r_1, r_2, \dots, r_{\overline{\kappa}})$ where $s \in \mathcal{S}$ is the state where we choose action $a \in \mathcal{A}$, $s' \in \mathcal{S}$ is visited after our agent repeated action a for $\overline{\kappa}$ time steps and $(r_1, r_2, \dots, r_{\overline{\kappa}})$ are all the collected rewards. When we collect a $\overline{\kappa}$-persistence transition it is important to underline that we are trying to use that $\overline{\kappa}$-persistence transition to learn $Q^\star_{\mathcal{K}}(\cdot, \cdot, k)$ for *all* the possible action-persistences in $k \in \mathcal{K}$. Now we have two distinguish two cases: first for any $\overline{\kappa}$-persistence transition collected with $\overline{\kappa} > 1$ we can create sub-transitions of $k < \overline{\kappa}$; secondly, the original transition with all others sub-transition represent partial information used to estimate any other persistence transition with $k > \overline{\kappa}$. Now we can define the first extension of the Bellman optimal operator for persistence k lower than the sampled $\overline{\kappa}$-persistence transition as follows:

$$(T^\star f)(s, a, k) = r_k(s, a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s' | s, a) \max_{(a', k') \in \mathcal{O}} f(s', a', k'). \tag{4.5}$$

If, instead, $\overline{\kappa} < k$, in order to update the value $Q^\star_{\mathcal{K}}(\cdot, \cdot, k)$, we partially exploit the $\overline{\kappa}$-persistent transition, but then, we need to *bootstrap* from a lower persistence Q-value, to compensate the remaining $k - \overline{\kappa}$ steps. To this end, we introduce the *bootstrapping* operator $T^{\overline{\kappa}} : \mathscr{B}(\mathcal{S} \times \mathcal{O}) \to \mathscr{B}(\mathcal{S} \times \mathcal{O})$ with $f \in \mathscr{B}(\mathcal{S} \times \mathcal{O})$:

$$\left(T^{\overline{\kappa}} f\right)(s, a, k) = r_{\overline{\kappa}}(s, a) + \gamma^{\overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s' | s, a) f(s', a, k - \overline{\kappa}). \tag{4.6}$$

It is important to observe that:

Lemma 4.1.1 (Decomposition of r_k). *Let $r_k(s, a)$ the expected k-persistent reward. Let $k' < k$, then it holds that:*

$$r_k(s, a) = r_{k'}(s, a) + \gamma^{k'} \int_{\mathcal{S}} P_{k'}(\mathrm{d}s' | s, a) r_{k-k'}(s', a) \qquad \forall (s, a) \in \mathcal{S} \times \mathcal{A}.$$

Proof. From the definition of r_k, it holds that $r_k(s,a) = \sum_{i=0}^{k-1} \gamma^i \big((P^\delta)^i r\big)(s,a)$. Hence:

$$
\begin{aligned}
r_k(s,a) &= \sum_{i=0}^{k-1} \gamma^i \big((P^\delta)^i r\big)(s,a) \\
&= \sum_{i=0}^{k'-1} \gamma^i \big((P^\delta)^i r\big)(s,a) + \sum_{i=k'}^{k-1} \gamma^i \big((P^\delta)^i r\big)(s,a) \\
&= r_{k'}(s,a) + \sum_{i=k'}^{k-1} \gamma^i \int_{\mathcal{S}} P_i(\mathrm{d}s'|s,a) r(s',a) \\
&= r_{k'}(s,a) + \sum_{i=k'}^{k-1} \gamma^i \int_{\mathcal{S}} P_{k'}(\mathrm{d}s''|s,a) \int_{\mathcal{S}} P_{i-k'}(\mathrm{d}s'|s'',a) r(s',a) \\
&= r_{k'}(s,a) + \gamma^{k'} \int_{\mathcal{S}} P_{k'}(\mathrm{d}s''|s,a) \sum_{j=0}^{k-k'-1} \gamma^j \int_{\mathcal{S}} P_j(\mathrm{d}s'|s'',a) r(s',a) \\
&= r_{k'}(s,a) + \gamma^{k'} \int_{\mathcal{S}} P_{k'}(\mathrm{d}s''|s,a) r_{k-k'}(s'',a).
\end{aligned}
$$

□

By combining the two operators 4.5 and 4.6, we obtain the *All-Persistence Bellman operator* $\mathcal{H}_{\overline{\kappa}} : \mathscr{B}(\mathcal{S} \times \mathcal{O}) \to \mathscr{B}(\mathcal{S} \times \mathcal{O})$ defined for every $f \in \mathscr{B}(\mathcal{S} \times \mathcal{O})$ as:

$$
(\mathcal{H}^{\overline{\kappa}} f)(s,a,k) = \big((\mathbb{1}_{k \le \overline{\kappa}} T^\star + \mathbb{1}_{k > \overline{\kappa}} T^{\overline{\kappa}}) f\big)\,(s,a,k). \tag{4.7}
$$

Thus, given a persistence $\overline{\kappa} \in \mathcal{K}$, $\mathcal{H}^{\overline{\kappa}}$ allows updating all the Q-values with $k \le \overline{\kappa}$ by means of $T^\star$, and all the ones with $k > \overline{\kappa}$ by means of $T^{\overline{\kappa}}$. The following result demonstrates the soundness of the proposed operator.

Theorem 4.1.1. *The all-persistence Bellman operator $\mathcal{H}^{\overline{\kappa}}$ fulfills the following properties:*

(i) $\mathcal{H}^{\overline{\kappa}}$ is a γ-contraction in L_∞ norm;

(ii) $Q^\star_{\mathcal{K}}$ is its unique fixed point;

(iii) $Q^\star_{\mathcal{K}}$ is monotonic in k, i.e., for all $(s,a) \in \mathcal{S} \times \mathcal{A}$ if $k \le k'$ then $Q^\star_{\mathcal{K}}(s,a,k) \ge Q^\star_{\mathcal{K}}(s,a,k')$.

Proof. (i) First, we prove the contraction property: we consider the L_∞-norm applied

to the state-action-persistence space $\mathcal{S} \times \mathcal{A} \times \mathcal{K}$:

$$
\begin{aligned}
&\left\|\mathcal{H}^{\overline{\kappa}} Q_1 - \mathcal{H}^{\overline{\kappa}} Q_2\right\|_\infty = \\
&= \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \left|\mathcal{H}^{\overline{\kappa}} Q_1(s,a,k) - \mathcal{H}^{\overline{\kappa}} Q_2(s,a,k)\right| \\
&= \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big| \mathbb{1}_{k \le \overline{\kappa}} \left(\left(T^\star Q_1\right)(s,a,k) - \left(T^\star Q_2\right)(s,a,k) \right) + \\
&\qquad\qquad + \mathbb{1}_{k \le \overline{\kappa}} \left(\left(T^{\overline{\kappa}} Q_1\right)(s,a,k) - \left(T^{\overline{\kappa}} Q_2\right)(s,a,k) \right) \Big| \\
&= \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big| \gamma^k \mathbb{1}_{k \le \overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \left[\sup_{a',k' \in \mathcal{A}\times\mathcal{K}} Q_1(s',a',k') - \sup_{a',k' \in \mathcal{A}\times\mathcal{K}} Q_2(s',a',k') \right] \\
&\qquad\qquad + \gamma^{\overline{\kappa}} \mathbb{1}_{k > \overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) \left[Q_1(s',a,k-\overline{\kappa}) - Q_2(s',a,k-\overline{\kappa}) \right] \Big| \\
&\le \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big\{ \gamma^k \mathbb{1}_{k \le \overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \Big| \sup_{a',k' \in \mathcal{A}\times\mathcal{K}} Q_1(s',a',k') - \sup_{a',k' \in \mathcal{A}\times\mathcal{K}} Q_2(s',a',k') \Big| \\
&\qquad\qquad + \gamma^{\overline{\kappa}} \mathbb{1}_{k > \overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) \Big| Q_1(s',a,k-\overline{\kappa}) - Q_2(s',a,k-\overline{\kappa}) \Big| \Big\} \\
&\le \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big\{ \gamma^k \mathbb{1}_{k \le \overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \sup_{\widetilde{a},\widetilde{k} \in \mathcal{A}\times\mathcal{K}} \Big| Q_1(s',\widetilde{a},\widetilde{k}) - Q_2(s',\widetilde{a},\widetilde{k}) \Big| \\
&\qquad\qquad + \gamma^{\overline{\kappa}} \mathbb{1}_{k > \overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) \sup_{\widetilde{s},\widetilde{a} \in \mathcal{A}\times\mathcal{K}} \Big| Q_1(\widetilde{s},\widetilde{a},k-\overline{\kappa}) - Q_2(\widetilde{s},\widetilde{a},k-\overline{\kappa}) \Big| \Big\} \\
&\le \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big\{ \gamma^k \mathbb{1}_{k \le \overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \sup_{\widetilde{s},\widetilde{a},\widetilde{k} \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big| Q_1(\widetilde{s},\widetilde{a},\widetilde{k}) - Q_2(\widetilde{s},\widetilde{a},\widetilde{k}) \Big| \\
&\qquad\qquad + \gamma^{\overline{\kappa}} \mathbb{1}_{k > \overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) \sup_{\widetilde{s},\widetilde{a},\widetilde{k} \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \Big| Q_1(\widetilde{s},\widetilde{a},\widetilde{k}) - Q_2(\widetilde{s},\widetilde{a},\widetilde{k}) \Big| \Big\} \\
&\le \sup_{s,a,k \in \mathcal{S}\times\mathcal{A}\times\mathcal{K}} \left\{ \left(\gamma^k \mathbb{1}_{k \le \overline{\kappa}} + \gamma^{\overline{\kappa}} \mathbb{1}_{k > \overline{\kappa}} \right) \|Q_1 - Q_2\|_\infty \right\} \\
&= \|Q_1 - Q_2\|_\infty \sup_{k \in \mathcal{K}} \gamma^{\min\{k,\overline{\kappa}\}} = \gamma \|Q_1 - Q_2\|_\infty .
\end{aligned}
$$

(ii): Since the contraction property holds, and being $(\mathcal{S} \times \mathcal{A} \times \mathcal{K}, d_\infty)$ a complete metric space (with d_∞ being the distance induced by L_∞ norm), the Banach Fixed-point theorem holds, guaranteeing convergence to a unique fixed point.

We now show that $Q^\star_{\mathcal{K}}$ is a fixed point of $T^\star$. We first need to define the extended Bellman expectation operators $T^\psi : \mathscr{B}(\mathcal{S} \times \mathcal{O}) \to \mathscr{B}(\mathcal{S} \times \mathcal{O})$ with $f \in \mathscr{B}(\mathcal{S} \times \mathcal{O})$ and

$\psi \in \Psi$:[1]

$$
\begin{aligned}
(T^{\psi} f)(s,a,k) &= r_k(s,a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) V(s'), \\
V(s) &= \sum_{(a,k)\in\mathcal{O}} \psi(a,k|s) f(s,a,k).
\end{aligned}
$$

As with standard Bellman operators, it trivially holds that $T^{\psi} Q^{\psi} = Q^{\psi}\ \forall \psi \in \Psi$. Thus, we can take into account the definition of value function V^{ψ} of a policy ψ and the standard Bellman Equations:

$$
\begin{aligned}
Q^{\psi}(s,a,k) &= r_k(s,a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) V^{\psi}(s'), \\
V^{\psi}(s) &= \sum_{(a,k)\in\mathcal{O}} \psi(a,k|s) Q^{\psi}(s,a,k).
\end{aligned} \tag{4.8}
$$

Following the same argument as in Puterman, 2014 [27], it holds that the optimal operator $T^{\star}$ improves the action-value function, i.e. $T^{\star} Q^{\psi} \geq Q^{\psi}$, and consequently $Q^{\star}_{\mathcal{K}}$ is the (unique) fixed point for $T^{\star}$, i.e., $T^{\star} Q^{\star}_{\mathcal{K}} = Q^{\star}$ by contraction mapping theorem.

Moreover, it holds that $T^{\overline{\kappa}} Q^{\psi} = Q^{\psi}$:

$$
\begin{aligned}
(T^{\overline{\kappa}} Q^{\psi})(s,a,k) &= r_{\overline{\kappa}}(s,a) + \gamma^{\overline{\kappa}} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) Q^{\psi}(s',a,k-\overline{\kappa}) \\
&= r_{\overline{\kappa}}(s,a) + \gamma^{\overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \Big[r_{k-\overline{\kappa}}(s',a) && (4.9) \\
&+ \gamma^{k-\overline{\kappa}} \int_{\mathcal{S}} P_{k-\overline{\kappa}}(\mathrm{d}s''|s',a) V^{\psi}(s') \Big] \\
&= \underbrace{r_{\overline{\kappa}}(s,a) + \gamma^{\overline{\kappa}} \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) r_{k-\overline{\kappa}}(s',a)}_{r_k(s,a)} && (4.10) \\
&+ \gamma^{k} \int_{\mathcal{S}} P_{\overline{\kappa}}(\mathrm{d}s'|s,a) \int_{\mathcal{S}} P_{k-\overline{\kappa}}(\mathrm{d}s''|s',a) V^{\psi}(s'') && (4.11) \\
&= r_k(s,a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) V^{\psi}(s') = Q^{\psi}(s,a,k),
\end{aligned}
$$

where in Equation (4.11) we used Lemma 4.1.1.

In conclusion,

$$
\begin{aligned}
\mathcal{H}^{\overline{\kappa}} Q^{\star}_{\mathcal{K}} &= \Big(\mathbb{1}_{k\leq\overline{\kappa}} T^{\star} + \mathbb{1}_{k>\overline{\kappa}} T^{\overline{\kappa}} \Big) Q^{\star}_{\mathcal{K}} \\
&= \mathbb{1}_{k\leq\overline{\kappa}} T^{\star} Q^{\star}_{\mathcal{K}} + \mathbb{1}_{k>\overline{\kappa}} T^{\overline{\kappa}} Q^{\star}_{\mathcal{K}} \\
&= \mathbb{1}_{k\leq\overline{\kappa}} Q^{\star}_{\mathcal{K}} + \mathbb{1}_{k>\overline{\kappa}} Q^{\star}_{\mathcal{K}} = Q^{\star}_{\mathcal{K}}.
\end{aligned}
$$

[1] The following can be extended without loss of generality to a continuous action space.

(iii) We provide the proof of monotonic property: given $(s,a) \in \mathcal{S} \times \mathcal{A}$, and given $k \le k'$, we have:

$$\begin{aligned} Q^\star_{\mathcal{K}}(s,a,k) &= (T^\star Q^\star_{\mathcal{K}})(s,a,k) \\ &= r_k(s,a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) \max_{a',k' \in \mathcal{A}\times\mathcal{K}} Q^\star_{\mathcal{K}}(s',a',k') \\ &\ge r_k(s,a) + \gamma^k \int_{\mathcal{S}} P_k(\mathrm{d}s'|s,a) Q^\star_{\mathcal{K}}(s',a,k'-k) \\ &= T^k Q^\star_{\mathcal{K}}(s,a,k') = Q^\star_{\mathcal{K}}(s,a,k'). \end{aligned}$$

□

Thus, operator $\mathcal{H}^{\overline{\kappa}}$ contracts to the optimal action-value function $Q^\star_{\mathcal{K}}$, which, thanks to monotonicity, has its highest value at the lowest possible persistence. In particular, it is simple to show that $Q^\star_{\mathcal{K}}(s,a,1) = Q^\star(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$ (Corollary 4.1.1), i.e., by fixing the persistence to $k = 1$ we retrieve the optimal Q-function in the original MDP, and consequently, we can reconstruct a greedy optimal policy.

Corollary 4.1.1 (Equivalence of $Q^\star_{\mathcal{K}}$ and $Q^\star$)**.** *For all $(s,a) \in \mathcal{S} \times \mathcal{A}$, the optimal action-value function $Q^\star$ and the optimal option-value function restricted to the primitive actions coincide, i.e.*

$$Q^\star_{\mathcal{K}}(s,a,1) = Q^\star(s,a).$$

Proof. Trivially, $Q^\star_{\mathcal{K}}$, defined on Ψ, coincides with the classic $Q^\star$ defined on the *primitive* policies $\pi \in \Pi$: in a first instance, we remark that $\Pi \subset \Psi$, hence all the policies defined on the space of primitive actions belong to the set of persistent policies. Furthermore, we can consider property (iii) of Theorem 4.1.1. As a consequence $Q^\star_{\mathcal{K}}(s,a,1) \ge Q^\star(s,a,k) \forall k \in \mathcal{K}$, i.e., for each state $s \in \mathcal{S}$ there is at least one optimal primitive action $a \in \mathcal{A}$ which is optimal among all the option set $\mathcal{O}$. Consequently, the two optimal action-value functions coincide. □

4.2 Persistent Q-learning

It may not be immediately clear what are the advantages of $\mathcal{H}^{\overline{\kappa}}$ over traditional updates. These become apparent with its empirical counterpart $\widehat{\mathcal{H}}^{\overline{\kappa}}_t = \mathbb{1}_{k \le \overline{\kappa}} \widehat{T}^\star_t + \mathbb{1}_{k > \overline{\kappa}} \widehat{T}^{\overline{\kappa}}_t$, where:

$$\begin{aligned} \left(\widehat{T}^\star_t Q\right)(s_t,a_t,k) &= r^k_{t+1} + \gamma^k \max_{(a',k') \in \mathcal{O}} Q(s_{t+k},a',k'), \\ \left(\widehat{T}^{\overline{\kappa}}_t Q\right)(s_t,a_t,k) &= r^{\overline{\kappa}}_{t+1} + \gamma^{\overline{\kappa}} Q(s_{t+k},a',k-\overline{\kappa}). \end{aligned}$$

These empirical operators depend on the current *partial history*, which we define as: $H^{\overline{\kappa}}_t := (s_t, a_t, r_{t+1}, s_{t+1}, r_{t+2}, \dots, s_{t+\overline{\kappa}})$, used by Algorithm 4.8 to update each persistence

in a backward fashion, as illustrated also in Figure 4.1. At timestep t, given a sampling persistence $\overline{\kappa}_t$, for all sub-transitions of $H_t^{\overline{\kappa}}$, starting at $t+i$ and ending in $t+j$, we apply $\widehat{\mathcal{H}}_t^{j-i}$ to $Q(s_{t+i}, a_t, k+d)$, for all $d \leq K_{\max} - k$, where $k = j - i$.

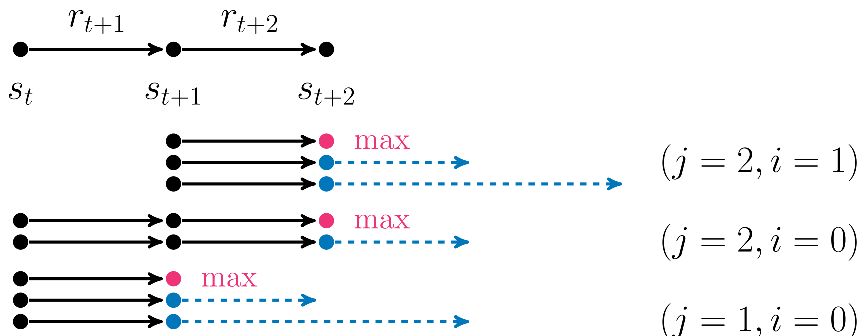


Figure 4.1: An example of the application of Algorithm 4.8 with $\overline{\kappa}_t = 2$ and $K_{\max} = 3$. Rows from top to bottom represent the updates in order. Updates involving the application of $\widehat{T}_t^{\star}$ and $\widehat{T}_t^{\overline{\kappa}_t}$ are denoted, respectively, by magenta and blue nodes, while dashed arrows represent the bootstrap persistence.

With these tools, it is possible to obtain the Persistent Q-learning algorithm (abbreviated as Per$Q$-learning), a persistent extension of Q-learning (Watkins, 1989 [39]), described in Algorithm 4.9. The agent follows a policy ψ_Q^{ϵ}, which is ϵ-greedy w.r.t. the option space and the current Q-function.

This approach extends the MSA-Q-learning algorithm presented in Schoknecht et al., 2003 [30], by bootstrapping higher persistence action values from lower ones. More precisely, the method developed in Schoknecht et al., 2003 [30] applies the same updates of Algorithm 4.9 for what concerns $\widehat{T}^{\star}$, but does not use $\widehat{T}^{\overline{\kappa}}$ instead. As it will be shown in the empirical analysis, in some domains this difference can be crucial to speed up the convergence. Similarly to MSA-Q-learning, Per$Q$-learning performs these updates backwards to allow for an even faster propagation of values. The proposed approach also differs from TempoRL Q-learning Biedenkapp et al., 2021 [5], where action-persistence is selected using a dedicated value-function, which is learned separately from the Q-function.

The asymptotic convergence of Persistent Q-learning to $Q_{\mathcal{K}}^{\star}$ directly follows from the application of the results in Singh et al., 2000 [33], thanks to the fact that $\mathcal{H}^{\overline{\kappa}}$ is a contraction, provided that their (mild) assumptions are satisfied.

Algorithm 4.8 All Persistence Bellman Update

Input: Sampling persistence $\overline{\kappa}_t$, partial history $H_t^{\overline{\kappa}_t}$,

1: Q-function Q.

Output: Updated Q-function Q'

2: $Q' \leftarrow Q$

3: **for** $j = \overline{\kappa}_t, \overline{\kappa}_t - 1 \dots, 1$ **do**

4: **for** $i = j-1, j-2, \dots, 0$ **do**

5: $k \leftarrow j - i$

6: $Q'(s_{t+i}, a_t, k) \leftarrow (1-\alpha)Q(s_{t+i}, a_t, k) +$

7: $\alpha \widehat{T}^{\star}_{t+i} Q(s_{t+i}, a_t, k)$ ▷ Optimal update

8: **for** $d = 1, 2, \dots, K_{\max} - k$ **do**

9: $Q'(s_{t+i}, a_t, k+d) \leftarrow (1-\alpha)Q(s_{t+i}, a, k+d) +$

10: $\alpha \widehat{T}^{k}_{t+i} Q(s_{t+i}, a_t, k+d)$ ▷ Bootstrapping update

11: **end for**

12: **end for**

13: **end for**

Algorithm 4.9 Persistent Q-learning (Per$Q$-learning)

Input: Learning rate α, exploration coefficient ϵ,

1: number of episodes N

Output: Q-function

2: Initialize Q arbitrarily, $Q(terminal, \cdot, \cdot) = 0$

3: **for** *episode* $= 1, \dots, N$ **do**

4: $t \leftarrow 0$

5: **while** s_t is not *terminal* **do**

6: $a_t, \overline{\kappa}_t \sim \psi_Q^{\epsilon}(s_t)$

7: **for** $\tau = 1, \dots, \overline{\kappa}_t$ **do**

8: Take action a_t, observe $s_{t+\tau}, r_{t+\tau}$

9: **end for**

10: Store partial history $H_t^{\overline{\kappa}_t}$

11: Update Q according to Alg.4.8

12: $t \leftarrow t + \overline{\kappa}_t$

13: **end while**

14: **end for**

Chapter 5

Algorithm Implementation

In this chapter, we present the implementation of two versions of the Persistent Q-Learning presented in section 4.2. In section 5.2 we extend the tabular version of Q-Learning with the new operator discussed before. In section 5.3 we introduce the TensorFlow library and Baselines framework, used to build the neural network for our experiment.

5.1 Language and Library

All the code created in this thesis is written in Python. Python is an interpreted, interactive, object-oriented programming language. It is widely used in Machine Learning field thanks to the simple syntax, which makes easy and fast to develop applications, and all the libraries already created that support ML algorithms implementation and extensions.

Gym All the environments tested in our work, presented in deep in next sections, are taken by the Gym OpenAI framework. Gym provides a common interface used to access the data retrieved from the underlying environment. One can also extend the basic environment, changing the rule or adding a wrapper, to extend or process the data exposed by the library.

TensorFlow TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. It is written with Python, C++ and CUDA and it is used for tensor computation through the GPU acceleration. We will use it to create the

neural network that represents our Q-function and to implement the back-propagation part.

Baselines Baselines (Dhariwal et al., 2017 [9]) is another useful Python framework that provides basic implementation of common Deep Reinforcement Learning algorithms such as DQN, A3C, PPO and so on. It uses TensorFlow to build all the networks and to implement back propagation. We used baselines for both test DQN and to extend already written code for DQN to produce Deep Persistence Q-Learning. It is better defined in the section 5.3.1

Tensorboard Another tool provided by TensorFlow is Tensorboard. With this tool it is possible to collect and plot custom data during exploration and training. We will see Tensorboard for data visualization in section 7.1

5.2 Persistent Tabular Q-learning

The first step to prove the soundness of our new algorithm is the implementation of the tabular version used on simple environments like a maze. We have created a class *PersistentAgent* initialized with the following parameters:

- env: the gym environment used in the run.
- max_persistence: it's the main parameter of our algorithm. It represents the maximum persistence that our agent can perform and affects Q-table dimension.
- discount_factor: this is γ introduced in Chapter 2 to compute cumulative discounted rewards.
- learning_rate: α used during the update of our Q-function.
- exploration_factor: the starting ε value for our ε-greedy exploration. During the initialization $\varepsilon =$ exploration_factor
- exploration_factor_decay: a constant in range $[0, 1]$ used to decrease ε at the end of each episode. At each episode, the agent updates $\varepsilon = \varepsilon^{decay}$

This class represents the agent used to wrap and store all data collected during exploration, encapsulate the state of our environment and to perform the main task, i.e. learn the Q-value function. During the initialization, our agent creates the model, a simple multidimensional array with dimension $(dim(s), dim(a), k_{max})$ where $dim(s)$ and $dim(a)$ are the observation space dimension and action space dimension of the Gym's *env* object and k_{max} is equal to *max_persistence* parameter. It creates also *persistence_history* to

collect persistence trace meanwhile executing the same action in the form of a list of tuples (s_t, r_t). During the first phase the Q-table is initialized with small values from a Gaussian with mean 0 and variance 1. There are two main methods exposed in this class:

- *get_action(state)*: used to choose the next action to perform.
- *train(state, action, next_state, reward, done)*: used to train our agent given an experience tuple.

Get action function The method *get_action* takes the Gym environment's state as input and computes the next action to perform.

Algorithm 5.10 Function *get_action(state)*

Input: Environment state s

Output: Action a and persistence k

1: **if** End persistence reached **then**
2: $\quad a, k \leftarrow$ derived from ε-greedy policy based on Q-table
3: **else**
4: $\quad a \leftarrow$ previously chosen action
5: $\quad$ Decrease current persistence index
6: **end if**
7: **return** a, k

The procedure is relatively simple: if the agent has reached the end of persistence it has to choose an action based on the current state. The agent recognizes the end of the persistence by just saving how many times it has repeated the action. With ε-greedy policy, with persistence extension, we mean the following:

$$\pi(a,k|s) = \begin{cases} \dfrac{\varepsilon}{|\mathcal{A}||\mathcal{K}|} + 1 - \varepsilon & \text{if } a, k \in \arg\max_{\substack{a' \in \mathcal{A} \\ k' \in \mathcal{K}}} Q(s, a', k') \\ \dfrac{\varepsilon}{|\mathcal{A}||\mathcal{K}|} & \text{otherwise} \end{cases}, \tag{5.1}$$

where ε is a value in the range $[0, 1]$ and decreased each episode. The equation 5.1 is valid only when we have a discrete action space $\mathcal{A}$ and a persistence space $\mathcal{K}$.

Train function After the agent chooses an action and a persistence it executes the action for an amount of steps equal to the chosen persistence and, during the execution, it collects the persistent transition. This will constitute the input of the *train* function,

the core of our algorithm. Now we need to update our knowledge of Q-function. This part is crucial to understand the mechanism, to simplify the process we make an example through figure 5.1. We consider a persistent transition with dimension 3.

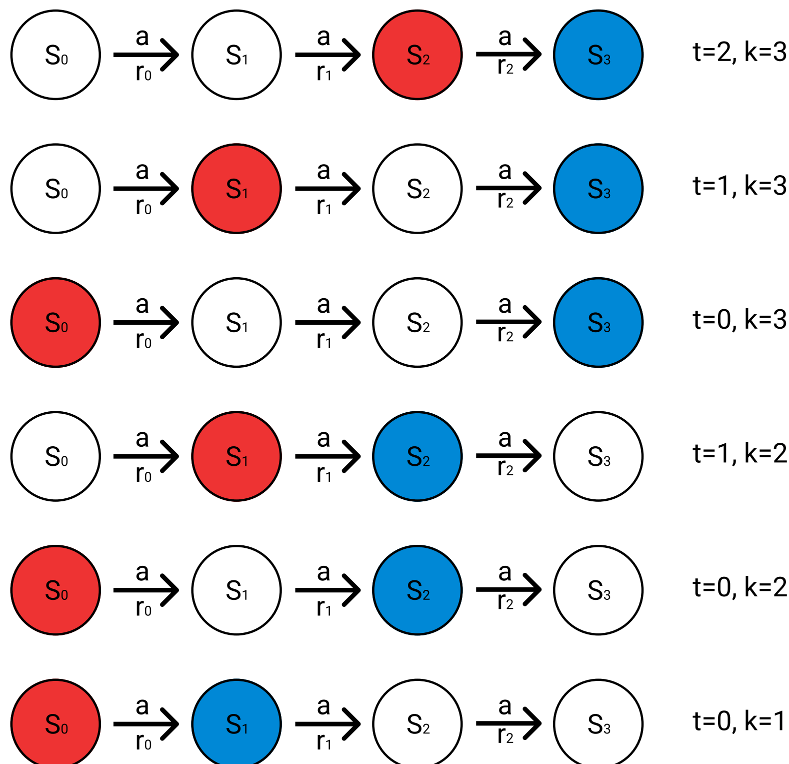


Figure 5.1: Train process with sub-persistence transitions. Red is the initial state and blue is the last state of the transition.

We need to analyze all the sub persistence transitions, starting from the last transition $s_2 \to s_3$. Then we take all the transitions moving back the initial state (the red circle) to the left one step at the time, until we reach the first state of the original persistent transition. At that point we will consider as final state the immediately previous state of the current final one (the blue circle). We will repeat the procedure until we process all the sub-persistent transition. For each sub-persistent transition we have to:

- Apply the Persistent Bellman optimal operator (see procedure 4.8 at line 7) to the transition.
- Consider the sequence of persistences value k included between the persistence of the current sub-transition and the *max_persistence* parameter. For each persistence in the sequence we consider a fictitious transition formed by two part. The first part is the original sub-transition and the second part is bootstrapped from

the Q-function. We take every fictitious transition as an input of the Bellman bootstrap operator (see procedure 4.8 at line 10).

The bootstrap part is one of the main differences with the previous algorithms. Thanks to this we can estimate persistence greater then the one sampled with the current value stored into the Q-function. In figure 5.2 we can see an example of bootstrap update, where the Q value for state s_1, action a and persistence $k = 5$ (sub-transition length of $k = 2$ and the remaining persistence of 3) is updated with the 2-persistence sub-transition and the remaining part bootstrapped from the current estimate of Q-function $Q(s_3, a, 3)$.

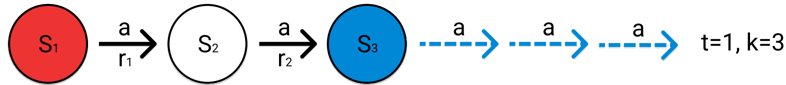


Figure 5.2: Example of bootstrap update for persistence 5 starting from a sub-persistent transition of 2. Red is the initial state and blue is the last state of the transition. The blue dotted lines represent the additional steps of the bootstrap.

Main loop The main loop of our algorithm is represented in a schematic way in figure 5.3, in which we can see the interaction between the agent and the environment. Before the main loop we have an initialization part for basic information. The following list summarizes the basic parameters:

- episodes: total number of episodes to simulate.
- seed: the seed used for the pseudo number generator to initialize all the random process. Used to guarantee the reproducibility of the experiment.
- env: the name of the environment (see the list 7.2.1).

. During the interaction, the agent collects the reward of the episodes, the mean persistence chosen by the agent and the ε value at the end each episode. We save all these information in an *npy* file to use them in the benchmark process to compare different seeds, different configurations and other algorithms.

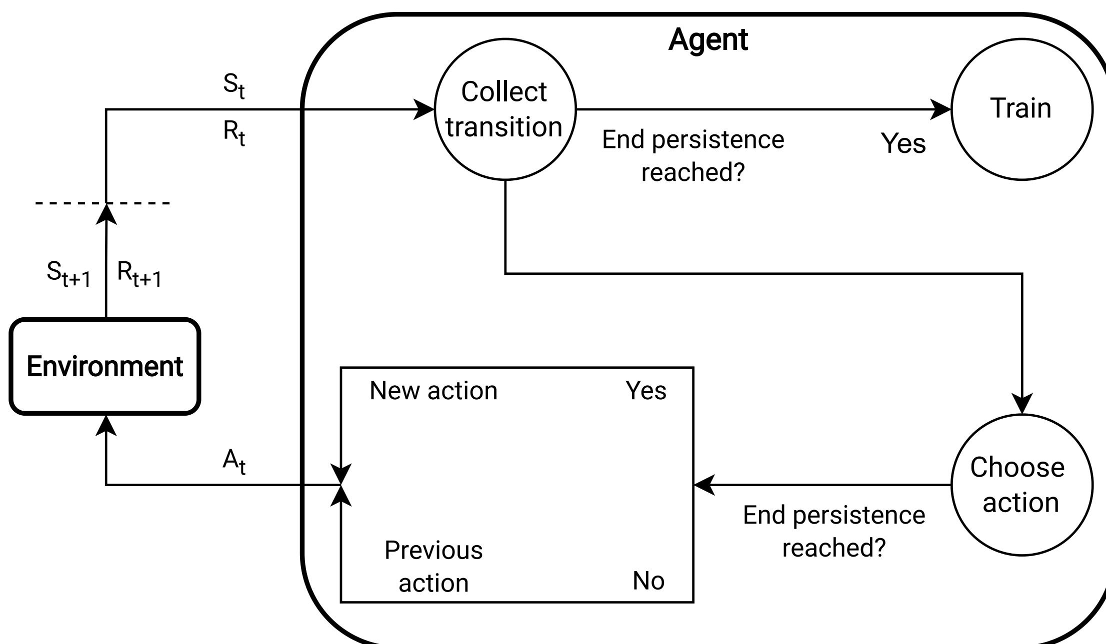


Figure 5.3: Environment and agent interaction with persistence.

5.3 Deep Persistent Q-learning

It is important to introduce how Baselines works and how it implements basic Reinforcement Learning algorithms, after that we can explain how it is possible to extend it with a new procedure.

5.3.1 Baselines

Baselines is a library provided by OpenAI to make easier for the research community to replicate, refine, and identify new ideas, and to create good baselines to build research on top of. The entire framework is organized inside the *baselines* package, with one main entry point. We can run the learn script with different parameters, the following list contains the main ones:

- Algorithm name: the name of the learner chosen (e.g. "deepq" for DQN, "perdq" for our algorithm).
- Gym environment: the name of the environment (e.g. CartPole).
- Number of time steps: how many steps to simulate.

We can add personal parameters used to build our learner or the network used by the train process. Inside the Baseline framework it is easy and fast to create and encapsulate

a new RL algorithm. This is extremely useful because we can define our learn function with all kind of parameters and then assign them when we run Baselines framework.

5.3.2 DeepQ Baselines

We have introduced in section 2.6.2 how how Deep Q-Learning works, now we can see how it is implemented in Baseline library, how to use TensorFlow and how it is possible to extend and use common code produced by OpenAI's framework.

Replay Buffer First important element is the replay buffer, used to collect transitions and sample tuples to use during the train session. The generic replay buffer is represented by the class *ReplayBuffer* that takes in the constructor only the *size* parameter, the max dimension of this FIFO buffer. We have just two methods for the replay buffer:

- *add*: it takes a tuple with current state, gained reward, action, next state and if the environment has reached the end. It is used to store a transition of the environment.
- *sample*: it takes the batch size, how many transitions we want to extract from the buffer, and returns a list of transitions.

The *PrioritizedReplayBuffer* object is an extension of *ReplayBuffer* with the same interface, but the transitions are not sampled uniformly, they are sampled with a distribution based on the priority of each sample, where the priority is back-forwarded by the TD error generated during the train step on that sample.

Base Network As we have said, in Deep Q-Learning we have to build a neural network to represent the Q-function. For example we can use a fully connected network or a convolutional network as base network. Baselines already provides the implementation of the main well-known network, as the multi layer perceptron (MLP), a multi convolutional layer network, the IMPALA network (Lasse et al., 2018 [15]) or the network proposed by Mnih, Kavukcuoglu, Silver, et al., 2015 [21]. The last one is the most common network used with image based Reinforcement Learning in games and published in Nature Journal. All the networks are configurable and ready to be used to build custom models. Thanks to the utilities defined by the Baselines framework we can build a more complex network. For the Deep Q-Learning, the network is created passing the following parameters:

- network: the base network discussed before, we can provide a string like "*mlp*" or "*conv_only*" to choose one of the predefined network utilities, or we can use a personal defined graph created with TensorFlow.

- hiddens: an array of integer values, used to add one fully connected layer each value with the specified hidden neurons. They are added at the end of the network, both for the *state value* part and the *action value* (the advantage function) if dueling is activated.
- dueling: this is a boolean parameter to use or not the dueling.
- layer_norm: this is a boolean parameter to normalize each layer defined by *hiddens.*
- network_kwargs: are passed to the utiliy of baselines to create the network, used to setup the different netwoks, for example the number of (shared) hidden layers and hidden neurons

Graph creation After the network definition, we need to build two parts of the graph, defined as follows:

- build_act: used to create the part of the TensorFlow graph to choose an action, by evaluating the Q-function, selecting the max value and following an ϵ-greedy policy based on the Q-value.
- build_train: used to create the part of the TensorFlow graph to train the network, implementing the back propagation, the target network and evaluating the loss function.

5.3.3 Deep Persistent Q-learning Implementation

In the previous section we introduced in a very general way the common facilities of what we have used to build our algorithm. We are now able to implement our new TensorFlow graph and algorithm.

Replay Buffer The replay buffer is the same discussed in 5.3.2, but we have extended the concept to the persistence. First we have just created a replay buffer for each persistence, to be able to save each persistence transition individually. We have tested both configuration with prioritized and not prioritized replay buffer. To generate each sub-transition starting from a bigger one we used the following procedure:

Algorithm 5.11 Multiple Replay Buffer Storing

Input: Maximum persistence $K_{\max}$, replay buffers $(\mathcal{D}_k)_{k=1}^{K_{\max}}$, transition tuple $(s_t, a_t, \overline{\kappa}_t, H_t^{\overline{\kappa}_t})$.

```
1: for k = 1, ..., K_max do                      ▷ Separate each k-persistence transition
2:
3:     for τ = 0, ..., max{κ̄_t − k, 0} do                 ▷ Transition for optimal update
4:         D_k ← D_k ∪ (s_{t+τ}, a_t, s_{t+τ+k}, r^k_{t+1+τ}, k, 0)
5:     end for
6:
7:     for τ = 1, ..., min{κ̄_t, k − 1} do               ▷ Transition for bootstrap update
8:         D_k ← D_k ∪ (s_{t+κ̄_t−τ}, a_t, s_{t+κ̄_t}, r^τ_{t+1+κ̄_t−τ}, τ, k − τ)
9:     end for
10: end for
```

When our agent chooses an action and a persistence, it collects all the states traversed and rewards gained and at the end it calls the method 5.11, that accepts the transition as a list, to split it in each individual sub-transition, for both optimal and bootstrap update. In code 5.11 the outer loop is used to extract each k-persistence transition to k_{max}. For example, we consider $\overline{\kappa} = 3$, as illustrated in figure 5.1, so when $k = 1$ we extract all persistence transitions with dimension one. The two inner loops are used to extract transitions for optimal update (the first loop) and for the bootstrap update (the second loop). For $k = 1$ only the first loop works, obviously there is no possibility to bootstrap an update with dimension 1, so the second loop works only with persistence greater than 2. The transition added to each $\mathcal{D}_k$ list is composed by the following elements:

- $s_{t+\tau}$ and $s_{t+\overline{\kappa}_t-\tau}$: are the initial states, corresponding to the red state in figure 5.1.
- a_t: current action, the same for each transition.
- $s_{t+\tau+k}$ or $s_{t+\overline{\kappa}_t}$: this is the last sampled state, corresponding to the blue state in figure 5.1
- $r^k_{t+1+\tau}$ or $r^\tau_{t+1+\overline{\kappa}_t-\tau}$: is the discounted sum of reward, corresponding to the sum of each reward gained from red state to blue state in figure 5.1.
- k or τ: is the number of states traversed in the actual persistence sub-transition. Corresponds to the distance from red and blue states in figure 5.1.

- 0 or $k-\tau$: these are the remaining part used by bootstrap update. With 0 we mean no bootstrap transition so the optimal updated is used. With a value greater than 0 we specify a bootstrap transition. When the learn function extracts such element from the replay buffer, this value represents the remaining part used to evaluate the Q. With value of 3, like in figure 5.2, the bootstrap update uses the value $Q(s_{t+2}, a_t, 3)$.

Network The standard DQN presented in Section 5.3.2 is not suited for our purpose, so we have extended the base network. For learning in the options space $\mathcal{O}$, the standard DQN is augmented with $K_{\max}$ distinct sets of action outputs, to represent Q-value of the options space $\mathcal{O} = \mathcal{A} \times \mathcal{K}$, while the first layers are shared, similarly to previous works (Arulkumaran et al., 2016 [2]; Lakshminarayanan et al., 2017 [14]; Biedenkapp et al., 2021 [5]).

In figure 5.4 we provide an example of the network constructed for the Mountain Car environment with an MLP shared part and the dueling activated.

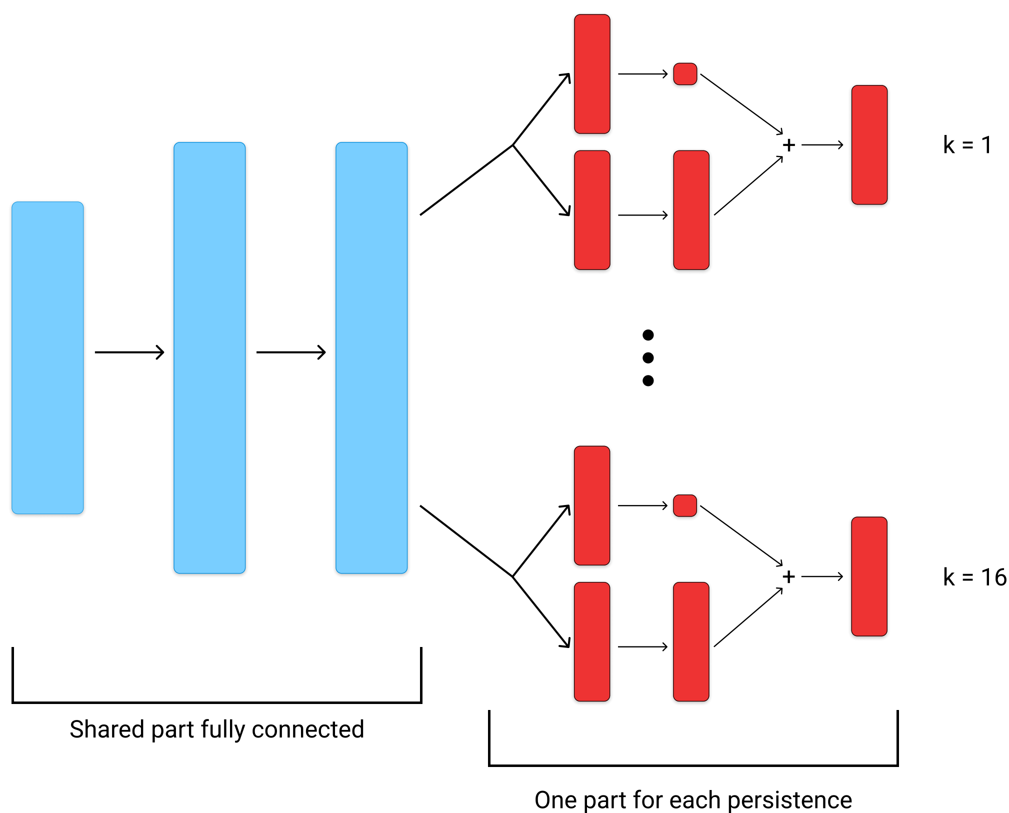


Figure 5.4: MLP example network with shared configuration

Graph creation As we have said in paragraph 5.3.2 we need two functions, *build_act* to implement the choose of an action and *build_train* to implement the back propagation part of the network. We first introduce the *build_act* function used for our algorithm.

Build act function *build_act* is called inside the *build_train* function and the result is returned to the main loop. It returns a Tensorflow wrapper to evaluate the graph when needed. In figure 5.5 is shown the graph created by the function for the environment Freeway, subdivided by logical blocks. It takes 3 input:

- *observation*: the state of the environment. For this example it is a tensor of size $84 \times 84 \times 4 \times bs$, where $84 \times 84 \times 4$ is the dimension of the image of Deep mind wrapper and bs is the batch size, the number of states in which we want to evaluate the graph. bs it is normally one, the actual state of the environment.
- *stochastic*: a boolean parameter to override the random choice.
- *eps*: the updated value of ε. It is saved inside a Tensorflow variable and used the next time we evaluate the graph.

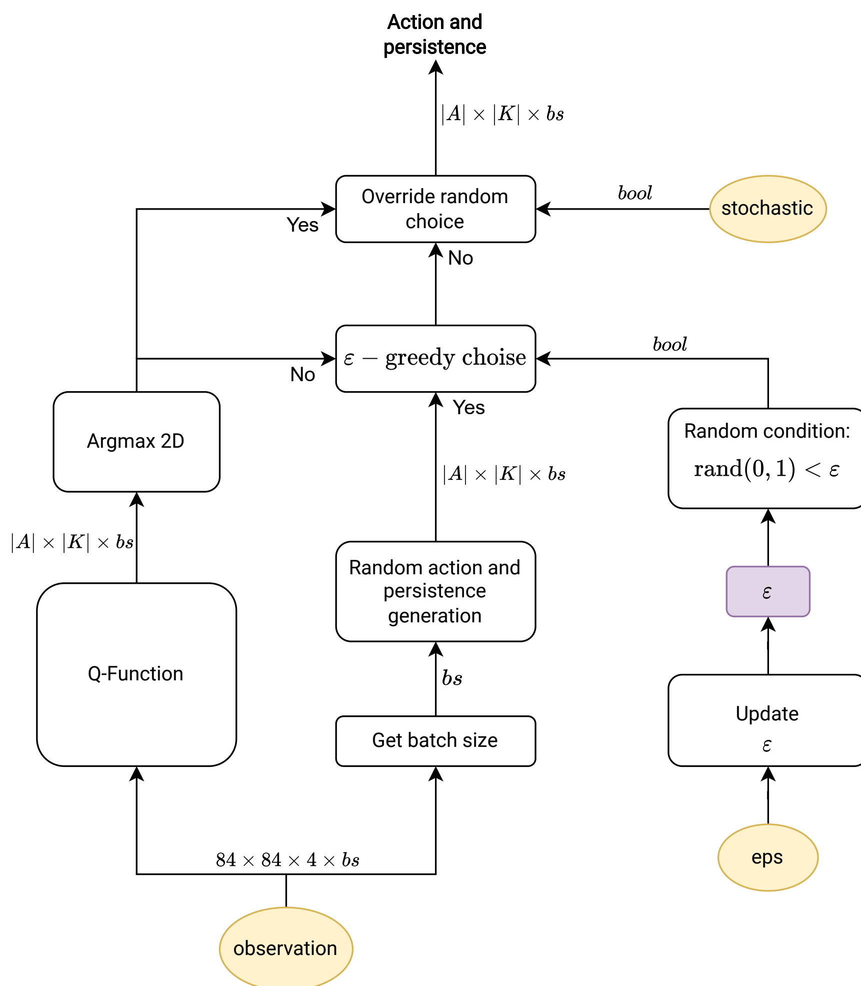


Figure 5.5: Tensorflow's Graph created by *build_act* function. Yellow nodes are input variables.

The *argmax_2d* function is used to extend the *argmax* to a multi dimensional function, like the Q-function created to integrate the persistence. The procedure takes a tri-dimensional tensor as input ($batch \times action \times persistence$), reshapes it to a mono-dimensional tensor, retrieves the *argmax* and then returns the unraveled index of each element of the batch.

Build train function We need now to use build the graph used to compute the target values, both for optimal and bootstrap update, to calculate the TD-error, create the gradients of the graph and update the weight of our networks trough back-propagation. We have schematized the graph in figure 5.6. The procedure is the following:

1. We need first to evaluate the *Q-function* and the *Target Q-function* in the *next state* (obs_{t+1}).

2. The result of the first is used to get the action and persistence with maximum. *q-value* and used both as indexes to get the value of the *Target Q-function* (this part is summarized by the *Double Q* block).
3. The evaluation of *Target Q-function* with obs_{t+1} is also used to get the q-value with the inputs *action* and *remainder* as indexes.
4. We need to choose for the current sample if it is an optimal or a bootstrap update. If the remainder is greater than 0, it is a bootstrap update.
5. Now the selected q-value is used to compute the target value for the update of the Q-function for the current sample.
6. Concurrently we have to compute the current value of the *Q-function* in the current sample, by evaluating it with in the *current state* (obs_t), the performed action and persistence.
7. We can now compute the error of our network. For our implementation we have used the Huber loss weighted for the current batch
8. Finally, we can compute the gradient for the weights of our network.

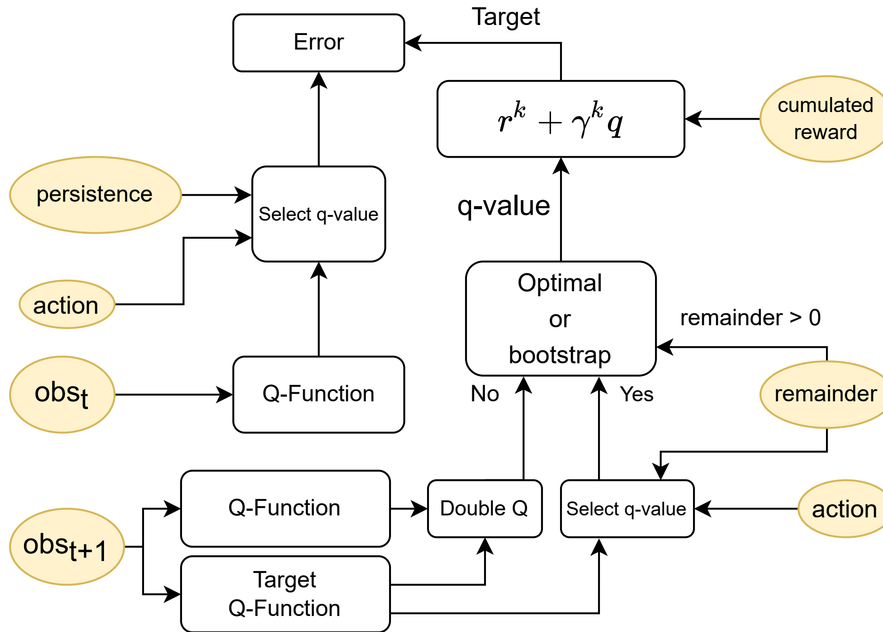


Figure 5.6: Tensorflow's Graph created by *build_train* function. Yellow nodes are input variables.

At the end of this process we have a list of gradients equal to k_{max}. In each list we have the gradients of the common part of our network (the convolutional network for example

for the Atari games) and the gradients of the weights for the single part, specialized for each persistence. Every time we need to train we just iterate over each replay buffer (one for each persistence), extract a number of samples equal to the batch size parameter and then update our network using just the gradients for the current persistence.

Chapter 6

Empirical Advantages of Persistence

In this chapter we want to discuss about the benefits of persistence. The main benefits related to the adoption of persistence are mainly two: exploration and learning. Persistence helps exploring larger regions of the state space, as we will discuss in Section 6.1; moreover, especially in environments where the reward is sparse, the related information can be learnt faster with direct updates, hence learning is improved, as seen in Section 6.3.

6.1 Exploration

When we adopt a persistent ε-greedy policy (defined by 5.1) we can modify the distribution of trajectories generated by our agent. Hence, we study the effects of a persisted exploratory policy on the MDP, i.e., a policy $\psi \in \Psi$ over persistence options $\mathcal{O}$. In different environments (like Mountain Car) this allows reaching faster states far from the starting point and, consequently, propagating faster the received reward signals. The reason is due to the increased chances of 1-persistent policies to get stuck in specific regions. In fact, as explained in Amin et al., 2020 [1], persistence helps to achieve *self-avoiding* trajectories, by increasing the expected return time in previously visited states. In order to show what we mean when we say that an agent can get stuck in a region, we have tested a full random policy $\psi \in \Psi$ over persistence options $\mathcal{O}$ in Mountain Car. In this environment, our agent can control a car in a one dimensional space. The possible actions are, accelerate left, accelerate right and do not accelerate. The target of the car is to drive up a steep hill (example in Figure 7.12).

We have collected all the states traversed by a full random agent with different values of K_{max}, both with a fixed persistence and a dynamic persistence. The results are shown

in figure 6.1. In each figure the x and y axes represent in order the position of the car and its velocity, and both constitute the state space of our environment. The value in each point represents in a logarithmic scale the counter of visited states. As we can see, when K_{max} is low our agent has less chance to reach the goal (represented by the blue dotted line). When we increase the persistence, the distribution over the states starts spreading in all the state space, covering a wider area than with low persistences. Especially, with persistence 1 and 4, our agent is not able to reach the goal. It is easy to understand why the car is stuck in the initial region when we have a low persistence, indeed we need to increase momentum in order to climb the hill. A rapid and continuous change in our decisions cannot bring any advantages to our momentum. This case will always happen when we start the learning process, in fact, the agent has not got any knowledge about the environment, so it is led to choose actions indistinctly.

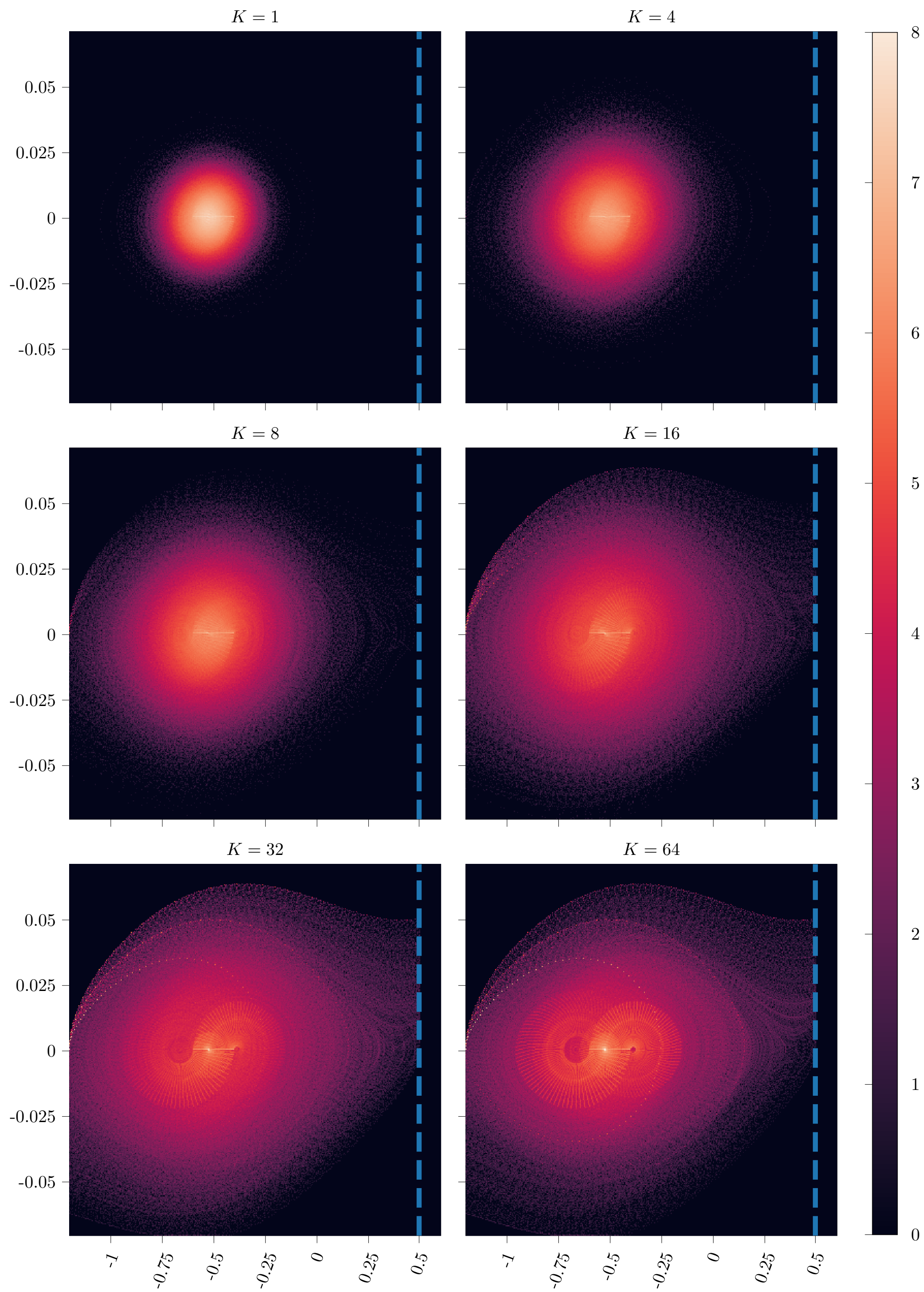


Figure 6.1: A full random policy $\psi \in \Psi$ over persistence options $\mathcal{O}$ in Mountain Car environment. The value represents in a logarithmic scale the counter of visited states. The x and y axes are respectively the position and the velocity of the car. We have used different K_{max} values. 10.000 episodes

In figure 6.2 we used a fixed persistence in the same environment. First, we can observe that, even if the goal for some values of K is almost equally visited, with a fixed persistence we have a different distribution of visited states. In the figure it is possible to observe different wide areas with a lower density (represented by the darkest colour). Our agent rarely visits these regions of state space. This means that with a dynamic persistence we can cover a wider set of trajectory from the initial point. Moreover, a fixed persistence degrades mush faster than a dynamic one, as we can see in figure 6.2 with $K = 64$ compared to the dynamic version in figure 6.1.

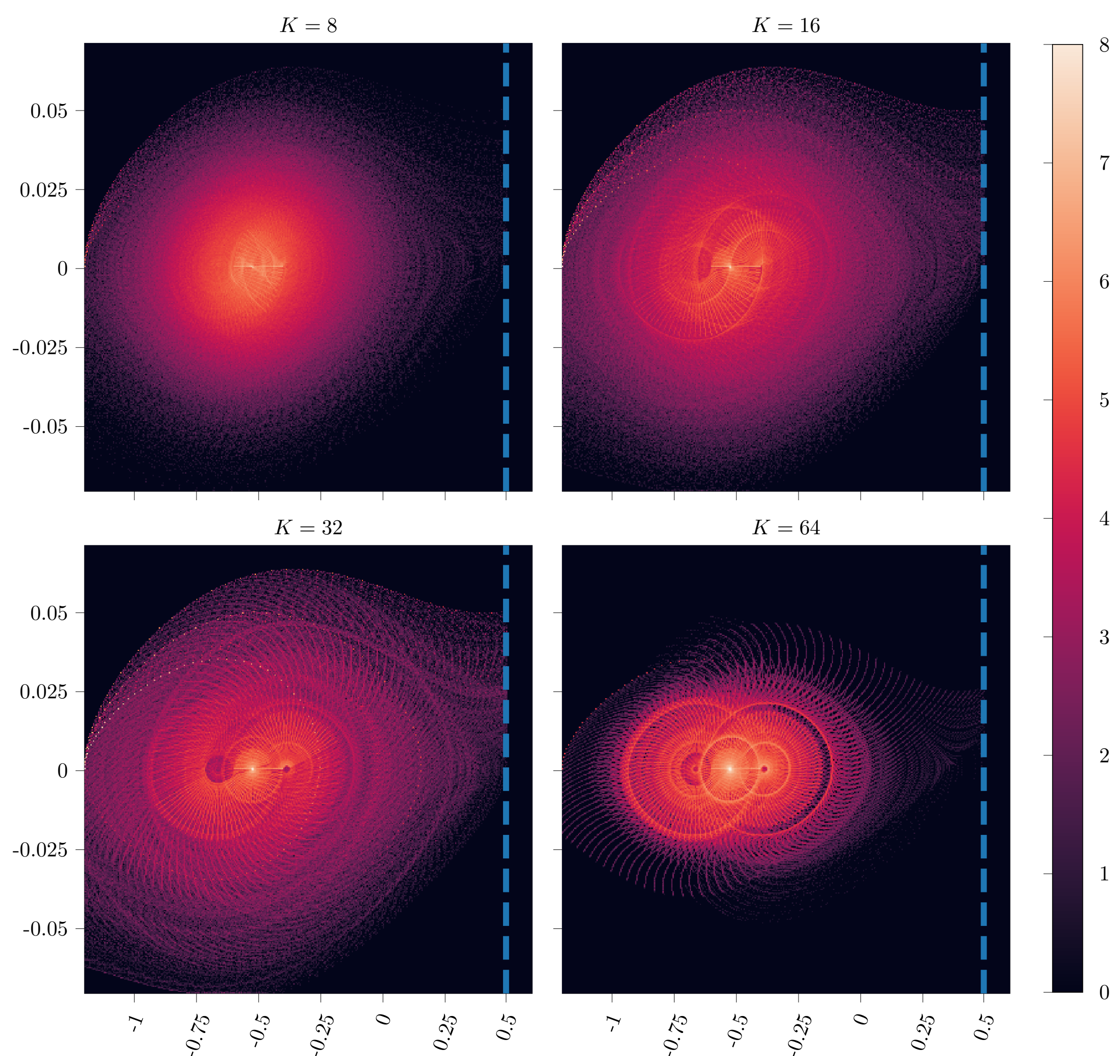


Figure 6.2: A full random policy with fixed persistence K in Mountain Car environment. The value represents in a logarithmic scale the counter of visited states. The x and y axes are respectively the position and the velocity of the car. We have used different K values. 10.000 episodes

6.2 Kemeny's Constant

As a further analysis, we can compute the *Kemeny's constant* (Catral et al., 2010 [7]; Patel et al., 2015 [25]), which corresponds to the expected first passage time from an arbitrary starting state s to another one s' under the stationary distribution induced by ψ.

First, we provide more details regarding how to obtain the transition Kernel implied from a persistent random variable acting on the environment, and how to compute its Kemeny's constant. After that, we have considered four discrete tabular environments for our evaluation: *Open* is a 10x10 grid with no obstacles, while the others, presented in Biedenkapp et al., 2021 [5], are depicted in Figure 7.10.

Consider an agent where actions are sampled from a generic policy $\pi(\cdot|s) =: \pi$ on the action space $\mathcal{A}$, independent from the current state, and where persistence is sampled from a discrete distribution ω with support in $\{1, \dots, K_{\max}\}$, independent from π. They jointly constitute the policy ψ over persistence options. The k-step Transition Chain induced by π over the state space $\mathcal{S}$ is defined as $P_k^\pi(s'|s) = \int_{\mathcal{A}} P_k(s'|s,a)\pi(da|s)$. This is equivalent to the Markov chain induced by π in the k-step MDP, where the control frequency is set to k times the base duration δ. We now consider the transition probability induced by the joint probability distributions π and ω up to a maximum of $K_{\max}$ steps, which for simplicity will here be referred as K. In order to define it, it is necessary to consider a fixed horizon H: when the total number of steps in the trajectory reaches the horizon, then the (eventual) persistence is stopped. This means that, if we start for example at the $H-j$ step, the probability of persisting j times the sampled action is equivalent to $\sum_{i \geq j} \omega_i$. This assumption is necessary for the Markov condition to hold. As a consequence, we define $\widetilde{\omega}_j = \{\widetilde{\omega}_{i,j}\}_i$ as a reduced distribution of ω to maximum j steps:

$$\widetilde{\omega}_{i,j} := \begin{cases} \omega_i & \text{if } i < j \\ \sum_{i=j}^{K} \omega_i & \text{otherwise} \end{cases}.$$

Finally we can recursively define the transition probability in H steps, induced by π and ω as:

$$\mathbb{P}_H^{\pi,\omega} := \sum_{k=1}^{K} \widetilde{\omega}_{k,H\wedge K} \mathbb{P}_{H-k}^{\pi,\widetilde{\omega}_{H-k}} P_k^\pi, \tag{6.1}$$

where $\mathbb{P}_0^{\pi,\omega} = \mathbb{1}_{\mathcal{S}\times\mathcal{S}}$ and $a \wedge b = \min\{a,b\}$. Equation (6.1) is not trivial and needs some clarifications. Let's consider an example, where $K = 4$ and $H = 3$. In this case the persistence distribution is $\omega = \{\omega_1, \omega_2, \omega_3, \omega_4\}$.

- With probability $\omega_3 + \omega_4$, the sampled persistence is equal to 3, and the related transition is P_3^π (since $H = 3$, sampling persistence 4 leads to repeat the action for 3 times);

- With probability ω_2, the sampled persistence is equal to 2. The first two steps evolve as P_2^π, and the last step follows $\mathbb{P}_1^{\pi,\widetilde{\omega}_1} = P^\pi$;
- With probability ω_1, the action is selected only once, and at the next step it has to be sampled again and eventually persisted for two steps w.p. $\omega_2 + \omega_3 + \omega_4$.

In other terms, denoting $\widetilde{\omega}_1 = \omega_1$, $\widetilde{\omega}_2 = \omega_2$, and $\widetilde{\omega}_3 = \omega_3 + \omega_4$:

$$
\begin{aligned}
\mathbb{P}_3^{\pi,\omega} =& \widetilde{\omega}_3 P_3^\pi + \widetilde{\omega}_2 P^\pi P_2^\pi + \widetilde{\omega}_1 [\widetilde{\omega}_1 P^\pi P^\pi + (\widetilde{\omega}_2 + \widetilde{\omega}_3) P_2^\pi] P^\pi \\
=& \widetilde{\omega}_3 \mathbb{1} P_3^\pi + \widetilde{\omega}_2 \underbrace{[(\widetilde{\omega}_1 + \widetilde{\omega}_2 + \widetilde{\omega}_3) P^\pi]}_{=\mathbb{P}_1^{\pi,\widetilde{\omega}_1}} P_2^\pi + \\
& + \widetilde{\omega}_1 \underbrace{[\widetilde{\omega}_1 (\widetilde{\omega}_1 + \widetilde{\omega}_2 + \widetilde{\omega}_3) P^\pi P^\pi + (\widetilde{\omega}_1 + \widetilde{\omega}_2) P_2^\pi]}_{=\mathbb{P}_2^{\pi,\widetilde{\omega}_2}} P_1^\pi \\
=& \widetilde{\omega}_3 \mathbb{P}_0^{\pi,\omega} P_3^\pi + \widetilde{\omega}_2 \mathbb{P}_1^{\pi,\omega} P_2^\pi + \widetilde{\omega}_1 \mathbb{P}_2^{\pi,\omega} P_1^\pi
\end{aligned}
$$

The meaning of the modified distribution $\widetilde{\omega}$ is related to the fact that, once the trajectory evolved for k steps, the remaining $H - k$ are still sampled, but when the last step H is reached, then the agent stops repeating in any case.

Kemeny's constant computation The formula used to compute the Kemeny's constant from the transition Kernel $\mathbb{P}_H^{\pi,\omega}$ can be obtain thanks to the following Proposition Kirkland, 2010 [13].

Proposition 6.2.1 (Kemeny's constant of an irreducible Markov Chain)**.** *Consider a Markov chain with an irreducible transition matrix P with eigenvalues $\lambda_1 = 1$ and $\lambda_i, i \in \{2, \dots, n\}$. The Kemeny constant of the Markov chain is given by*

$$
Kem = \sum_{i=2}^{n} \frac{1}{1 - \lambda_i}.
$$

The introduction of the parameter H is necessary to retrieve an irreducible transition matrix $\mathbb{P}_H^{\pi,\omega}$ maintaining the Markov property.

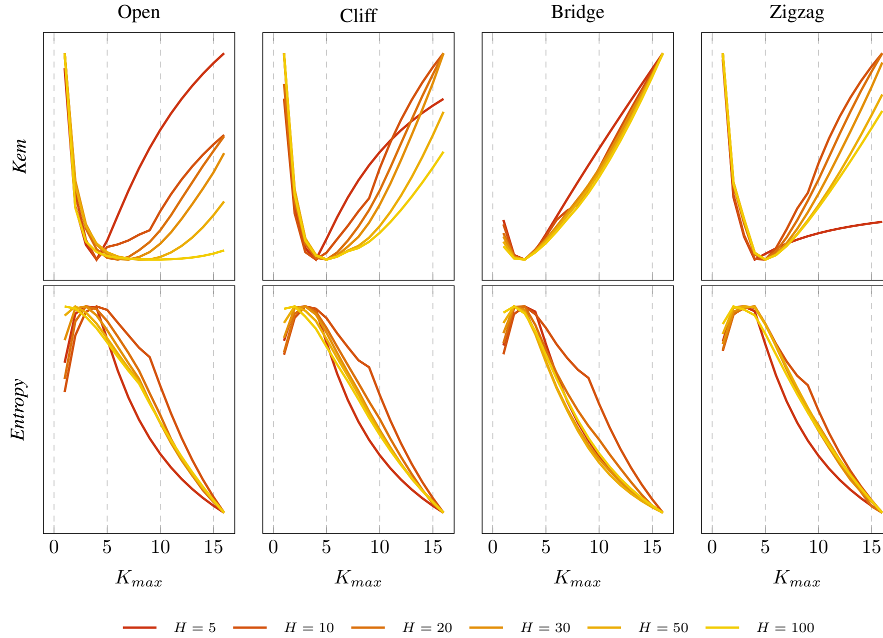


Figure 6.3: Normalized Kemeny's constant and entropy in Tabular environments as function of the maximum persistence and horizon H.

In order to compute the curves of Kemeny's constant in Figure 6.4, we consider K_{max} as a variable, and exploration is performed by a discrete uniform random variable in $\mathcal{O} = \bigcup_{k \in \mathcal{K}} \mathcal{O}^{(k)}$, i.e., the distribution π is uniform over the action space $\mathcal{A} = \{left, down, right, up\}$, and ω uniform over $\mathcal{K}$. In Figure 6.3 we show the curves of Kemeny's constant and entropy with different value of K_{max} and H, and Figure 6.4 refers to the same Kemeny's curves selected for $H = 30$.

As we can see in the figure, for each value of H there is a similar pattern: increasing K_{max}, the related values for Kemeny's constant initially tend to decrease, indicating that persistence helps for a faster exploration through the state space. Persisting actions for long times does not help exploration, since agents might be more frequently standing in front of walls. Consequently, depending on the different design of the environments, Kemeny's values begin to increase. In the bottom plots of Figure 6.3 we can observe also the curves related to the entropy induced by $\mathbb{P}_H^{\pi,\omega}$: again, the maximum value of entropy is attained by $K_{max} > 1$. However, the curves soon start to decrease dramatically, indicating that reaching distant states sooner is not strictly related to its visitation frequency.

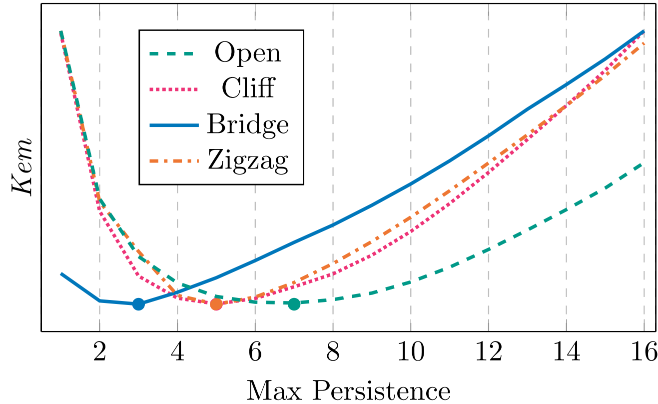


Figure 6.4: Normalized Kemeny's constant in tabular environments as function of the maximum persistence $K_{\max}$. Bullets represent the minimum value of the constant.

In Figure 6.4, we plot the variations of Kemeny's constant as a function of the maximum persistence $K_{\max}$, while following a uniform policy ψ over $\mathcal{O}$. We observe that increasing $K_{\max}$ promotes exploration, and highlights the different values of $K_{\max}$ attaining the minimum value of the constant, due to the different complexity of the environments.

6.3 Sample Complexity

The second relevant effect of persistence concerns with the sample complexity. The intuition behind persistence relies on the fact that the most relevant information propagates faster through the state-action space, thanks to multi-step updates. Moreover, these updates are associated to a lower discount factor, which allows for better convergence rates. In order to evaluate the sample efficiency of Per$Q$-learning, separately from its effects on exploration, we considered a *synchronous* setting (Kearns et al., 1999 [12]; Sidford et al., 2018 [32]) in a deterministic 6x6 Gridworld. At each iteration t, the agent has access to a set of independent samples for each state-action pair. In standard Q-learning the samples are used to update the action value function $Q(s,a)$ for each $(s,a) \in \mathcal{S} \times \mathcal{A}$. In Per$Q$-learning, the samples are combined to obtain each possible set of $\overline{\kappa}$-persistent transitions, i.e., the tuples related to each possible $(s,a,k) \in \mathcal{S} \times \mathcal{O}$, with $K_{\max} = 6$; finally, the persistent action value function is updated.

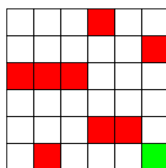


Figure 6.5: Grid environment. Red cell denote holes, green cells the goal.

The environment considered is depicted in Figure 6.5: a 6x6 deterministic Gridworld, with holes. The set of (primitive) actions is $\mathcal{A} = \{left, down, right, up\}$. Transitions are deterministic. Going outside the borders and falling off a hole result in a punishment with a negative reward, respectively equal to -100 and -10 (hence, outer borders are not blocking the movement of the agent). The reward for reaching the goal instead is equal to +100. In all these cases the episode terminates; all other states result in a small negative reward (-1), to incentivize finding the shortest paths towards the goal.

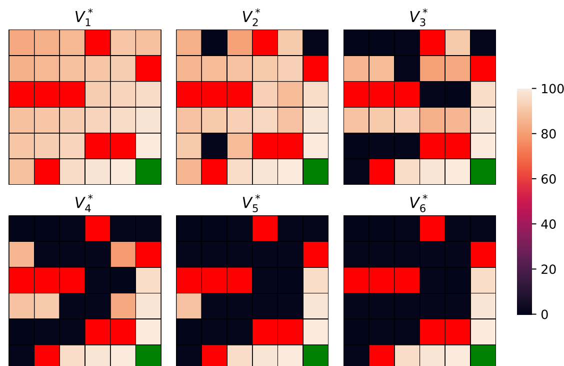


Figure 6.6: k-value function representation for different persistence values k ($K_{\max} = 6$). Red cells denote holes, green cell the goal.

In order to exploit only the convergence properties of Per$Q$-learning algorithm, without considering the exploration factor, we consider a synchronous learning framework: we assume to have access to the whole transition model, in such that, at each iteration t, we are able to collect an independent sample $s' \sim P(\cdot|s, a)$ for every state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$. Since the environment is deterministic, this means that we have access to the whole transition matrix P. For each simulation, the value estimation for each state-action pair (or state-option pair, in the case of Per$Q$-learning) is initialized sampling from a standard Gaussian random variable. In each iteration of Q-learning, the algorithm performs a full update of the Q-function estimates. In each iteration of Per$Q$-learning, before performing the full update, the *primitive* tuples are combined together, in order to collect a sample for each possible (s, a, k) pair in $\mathcal{S} \times \mathcal{A} \times \mathcal{K}$.

The representations of the $k-$step value functions $V_k^\star$ are shown in Figure 6.6, where $V_k^\star(s) = \max_{a \in \mathcal{A}} Q^\star(s, a, k)$. It is useful to remark that this value function do not coin-

cide with the optimal value function in the $k-$persistent MDP $\mathcal{M}_k$, as $V_k^\star(s)$ represent the value function at the state s restricted to persist k times only the first action, and the following the optimal policy $\pi^\star$.

Parameters used for experiments:

- Initial estimation: $Q(s,a,k) \sim \mathcal{N}(0,1)\ \forall (s,a,k) \in \mathcal{S} \times \mathcal{A} \times \mathcal{K}$;
- Discount factor: $\gamma = 0.99$;
- Learning rate: $\alpha = 0.1$;
- Maximum number of iterations: 400 (plots truncated at 200).

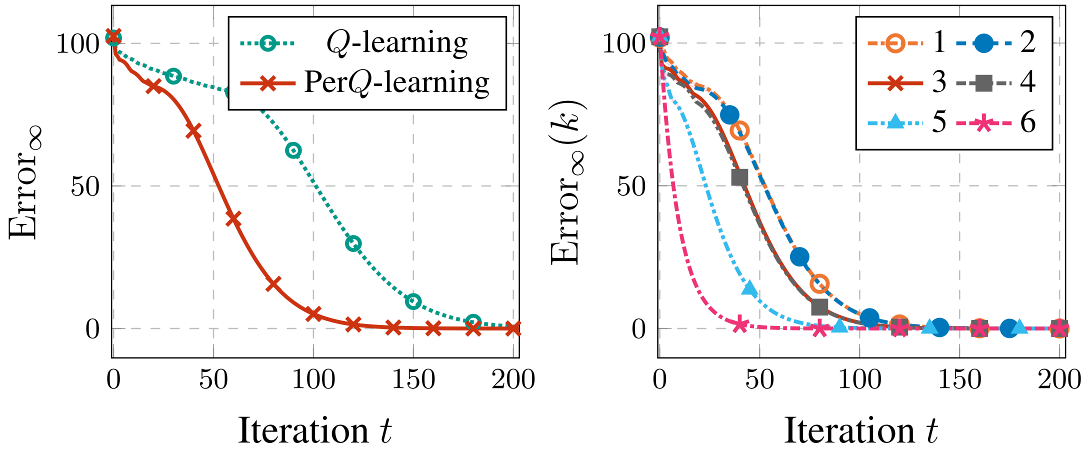


Figure 6.7: L_∞ error on 6x6 grid-world between synchronous Q-learning and Per$Q$-learning (left) and for different persistence options $k \in \{1, ..., 6\}$ of Per$Q$-learning (right). (100 runs, avg $\pm$ 95 % c.i.)

In Figure 6.7 left, we compare the L_∞ error of Q-learning estimating $Q^\star(s,a)$, i.e., $\max_{s,a \in \mathcal{S}\times\mathcal{A}} |Q_t(s,a) - Q^\star(s,a)|$, and that of Per$Q$-learning estimating $Q^\star_{\mathcal{K}}(s,a,k)$, i.e., $\max_{s,a,k \in \mathcal{S}\times\mathcal{O}} |Q_t(s,a,k) - Q^\star_{\mathcal{K}}(s,a,k)|$, as a function of the number of iterations t. We observe that, although estimating a higher-dimensional function (as $Q^\star_{\mathcal{K}}(s,a,k)$ is a function of the persistence k too), Per$Q$-learning converges faster than Q-learning. In Figure 6.7 right, we plot the L_∞ error experienced by Per$Q$-learning for the different persistence options $\mathcal{O}^{(k)}$, i.e., $\text{Error}_\infty(k) := \max_{s,a \in \mathcal{S}\times\mathcal{A}} |Q_t(s,a,k) - Q^\star(s,a,k)|$ for $k \in \mathcal{K}$. We note that, as expected, higher values of k lead to faster convergence; consequently, the persistent Bellman operator helps improving the estimations also for the lower option sets. Indeed, we can see that also $Q_t(\cdot,\cdot,1)$, which represents the action value function for the primitive actions, converges faster than classic Q-learning.

Conclusion In this chapter we analysed many of the advantages introduced with dynamic persistence. First of all, we have seen how we can improve exploration by

changing the distribution of probability over trajectories in the state space. As a result, in some environments the possibility of avoiding stuck regions is increased, but it is also possible to reach the goal states in a minor amount of time.

The second problem is instead related to sample complexity, and therefore to the number of samples needed by the agent to be able to learn the model. Dynamic persistence theoretically guarantees a reduction of sample complexity, in such a way the model will eventually converge to an optimal solution in a brief time.

In the next Chapter we will see whether the mainly theoretical analyses, exposed in the latter paragraphs, will be confirmed by the experiments.

Chapter 7

Experimental Evaluation

In this chapter we will report the main experimental results of our work. We start with the description of the tools used to collects and analyze the data with the chosen key performance indicators (KPI). After that, we show our results compared with the classic DQN and then the comparison with the TempoRL algorithm.

7.1 Methodologies

As mentioned in Section 5.1, Tensorboard is a great tool provided in the Tensorflow package used to collect and show data in real time. It provides a Python library and a stand alone executable that implements a local server and a web page to visualize all the chart and other useful information. Thanks to Tensorboard we can save different type of information during the train of our agent. The data saved are the following:

- Text data: useful to distinguish different run (results showed in figure 7.2)
- Scalar data: used to create charts, like mean reward or persistence frequency.
- Graph: created by Tensorflow, from which we can debug what we have created.

After the execution of the local server we can navigate through the data. In figure 7.1 we can see the loaded page.

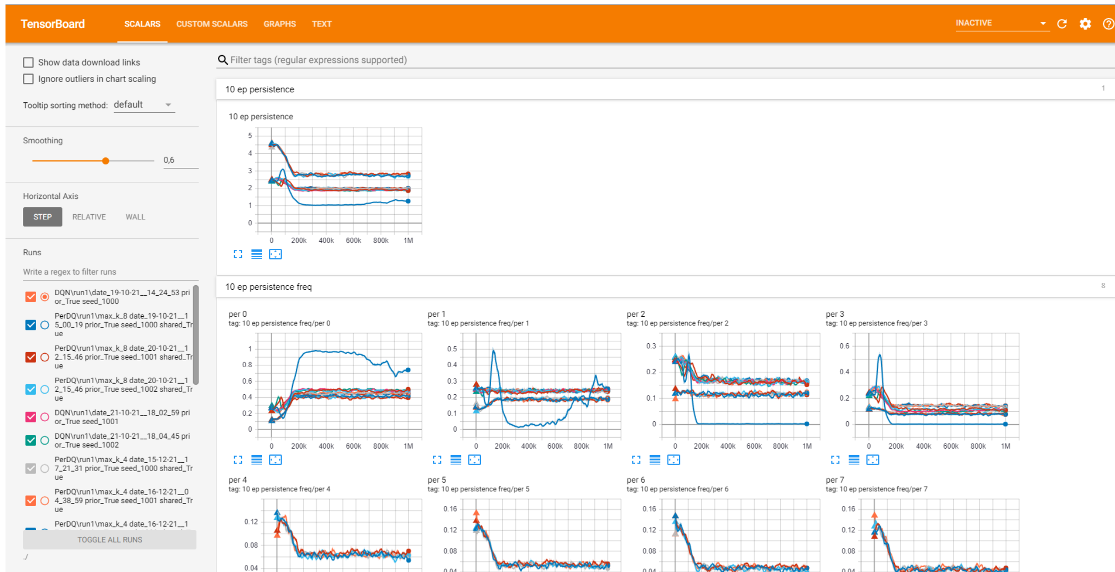


Figure 7.1: Main page for the local server of Tensorboard.

In the left panel we have different options. For example we can enable and disable different runs to compare the charts that we want. We can also modify the smoothing parameter to smooth the chart. To select a different type of data we have to use the panel above, for example we can choose "*text*".

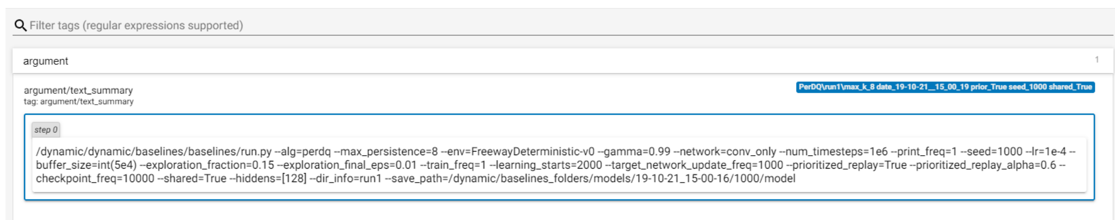


Figure 7.2: Text parameter saved by Tensorboard.

It is also possible to visualize a *margin chart*, with a mean and a boundary, through the definition of three scalar values. The results is the following:

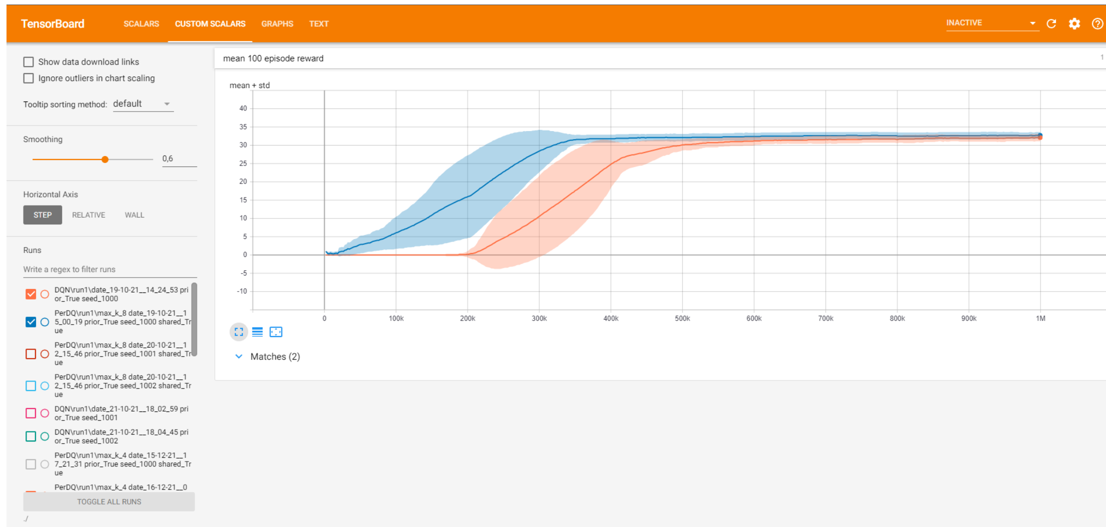
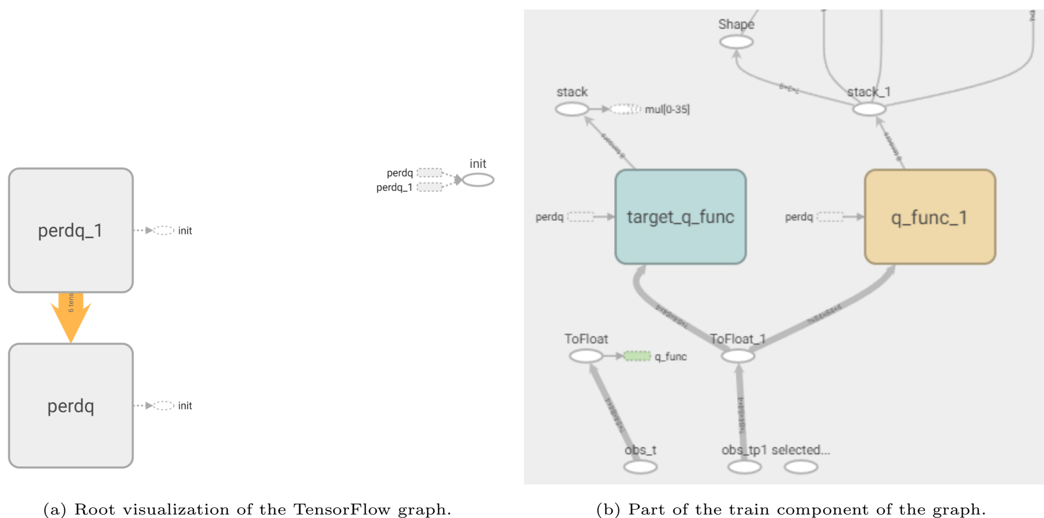


Figure 7.3: Margin chart with mean and standard deviation.

Another very important tool is the graph visualization. It is useful to analyze and debug the network creation. We can see different part of the network in figure 7.4 and 7.5



(a) Root visualization of the TensorFlow graph.

(b) Part of the train component of the graph.

Figure 7.4

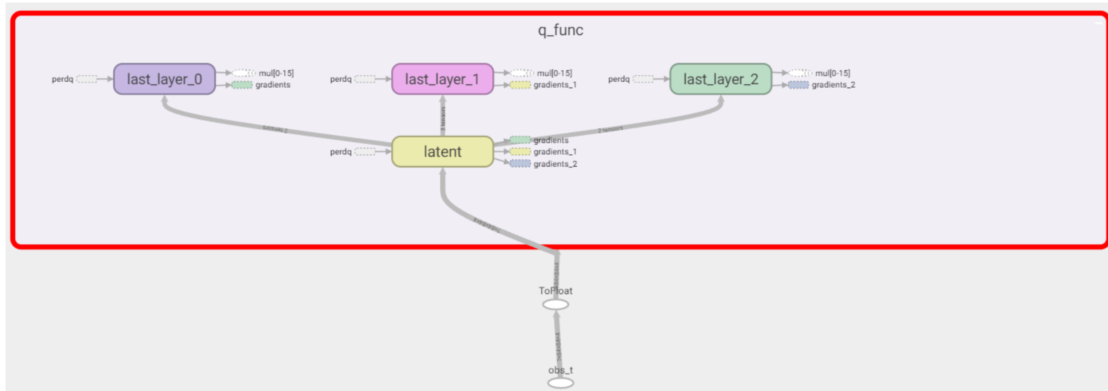
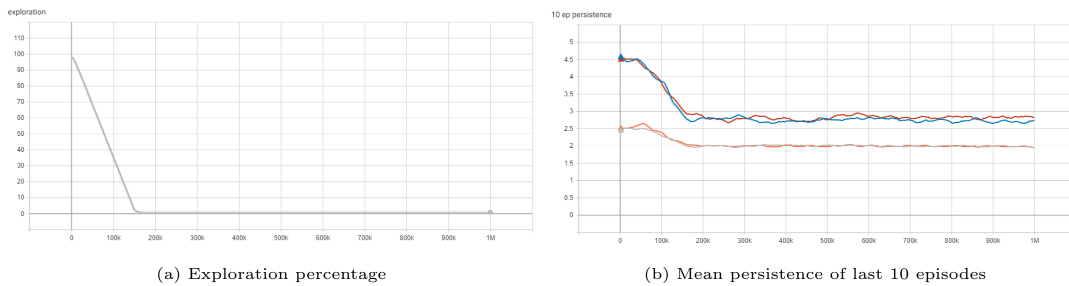


Figure 7.5: Q function for for the environment Freeway with persistence 3.

Through our work we have collected the following metrics:

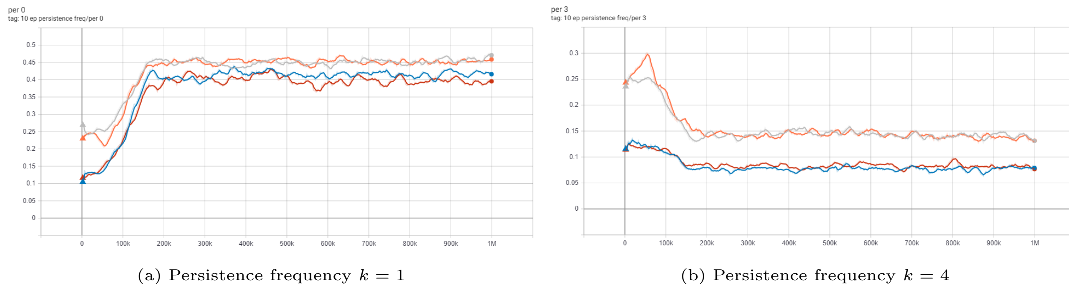
- Exploration percentage for each episodes (figure 7.6a)
- Mean persistence of last 10 episode to understand the agent's behaviour (figure 7.6b)
- Chosen frequency for each persistence to analyze how many times our agents choose each persistence (figure 7.7)
- Mean reward in the last 100 episodes as main key performance indicator to compare each run with different parameters (figure 7.8)



(a) Exploration percentage

(b) Mean persistence of last 10 episodes

Figure 7.6: Exploration percentage and mean persistence collected by Tensorboard



(a) Persistence frequency $k = 1$

(b) Persistence frequency $k = 4$

Figure 7.7: Persistence frequency collected by Tensorboard

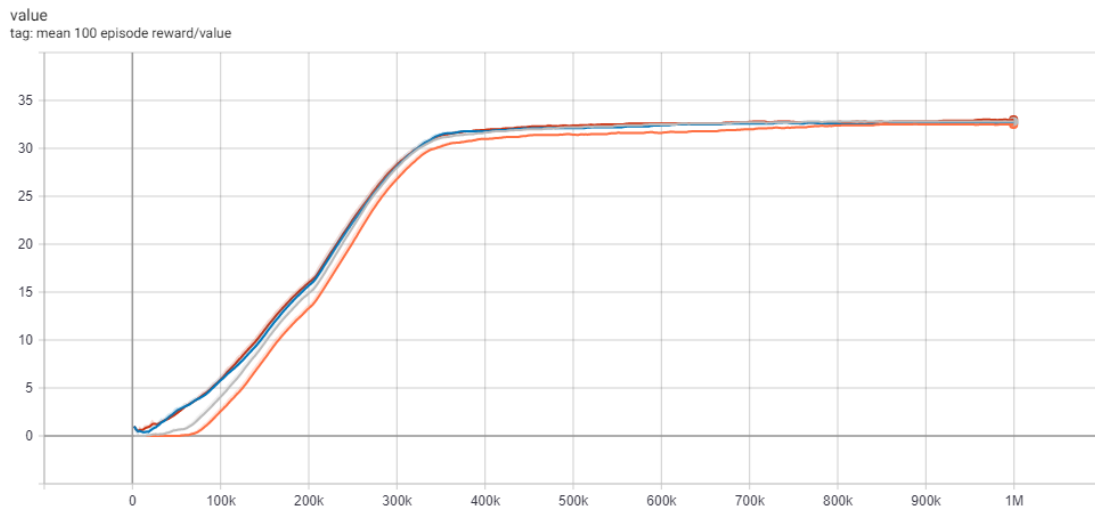


Figure 7.8: Mean reward of last 100 episodes.

After the collection and analysis of results, we have evaluated the mean and the confidence interval through each seed of the same run. The data are collected each episode, so we have aligned the data for the time step, through step interpolation of the mean reward of the last 100 episodes. We have finally saved the data and plotted it (figure 7.9).

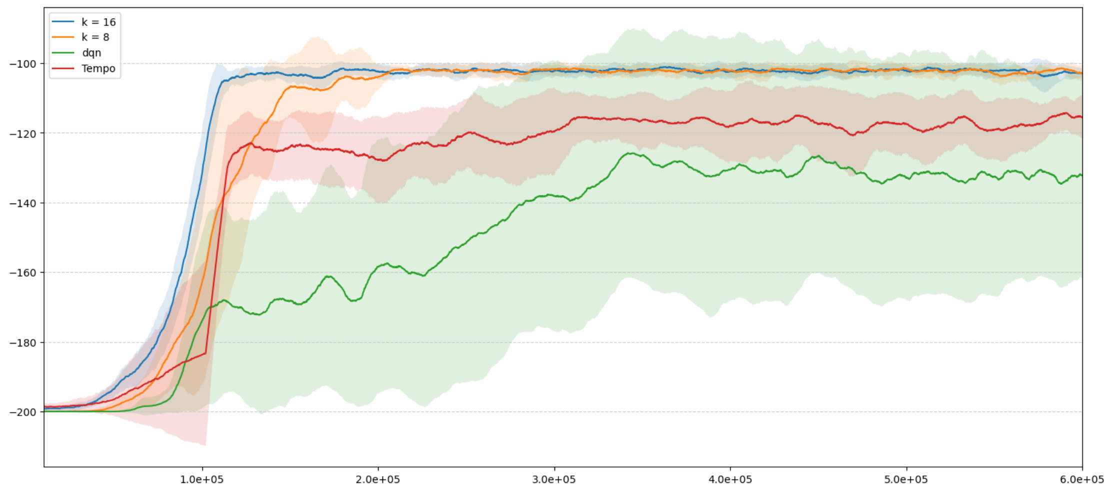


Figure 7.9: Example plot with mean and standard deviation through seeds of the same run (MountainCar environment).

7.2 Details on Experimental Evaluation

We can show now the details of our experiments. First we will show the results of the tabular version (Per$Q$-learning algorithm) and then the results of the Deep Reinforcement Learning version (PerDQN algorithm), both for the Mountain Car and the Atari games environments. First we need to describe the machine used during our research.

Infrastructure The experiments have been run on a machine with two GPUs: a Nvidia Tesla K40c and a Nvidia Titan Xp with a Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz (4 cores, 4 thread, 6 MB cache) and 32 GB RAM.

7.2.1 Tabular Environments

For Persistent Q-learning we have used the following environments:

- FrozenLake, with different dimension.
- Cliff.
- Bridge.
- ZigZag.

They are all toy environments used to evaluate in a fast and simple way the performance of our algorithm, particularly suited for our work because of the sparsity of rewards.

They are an extension of *FrozenLake* environment found in *Gym* library. *FrozenLake* is just a simple maze, implemented as grid-world, with 4 type of blocks:

- Start block: where our agent starts.
- Frozen block: the safe block, where the agent can walk. This block can be either slippery or not, this means that there is some probability to succeed in our action, otherwise, a random direction is chosen. It is used to have a stochastic environment.
- Hole block: if the agent steps in this block the run finishes and we have to reset our environment.
- Goal block: the goal of our environment is to reach this block.

The accepted actions of our agent are simply *go_left*, *go_right*, *go_up* and *go_down*. The rewards are +1 if we reach the goal, 0 in other cases. Another parameter of these environments is the boolean *is_slippery*: if it is equal to $True$ our agent will move in intended direction with probability of 1/3 and in either perpendicular direction with equal probability of 1/3 in both directions. For example, if the action is *go_left*, then:

- P(move left)=1/3
- P(move up)=1/3
- P(move down)=1/3

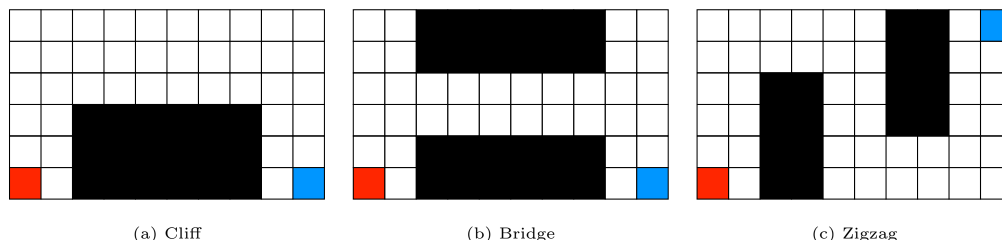
Another important parameter is *desc*, a list with a textual representation of our environment, used to build a specified map as shown in 7.1.

```
map = [
        "SFFFFFFF",
        "FFFFFFFF",
        "FFFHFFFF",
        "FFFFFHFF",
        "FFFHFFFF",
        "FHHFFFHF",
        "FHFFHFHF",
        "FFFHFFFG",
    ]
```

Code 7.1: Map textual example. S: start, F: frozen, H: hole, G: goal

We have extended this environment with a slightly different one. First of all we have disabled the *is_slippery* parameter, and we have added a negative reward for the holes. We have also added a parameter *dim* to specify the dimension of a square map if we

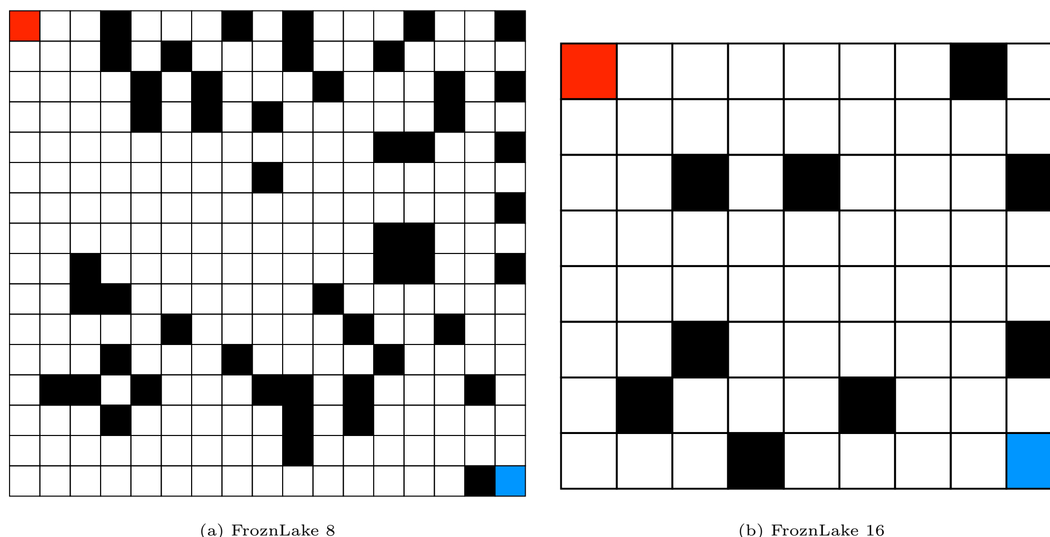
don't directly provide *desc*. If we provide this parameter as an integer, the environment produces a random map with dimensions $dim \times dim$ and probability of 85% that a block is a safe block. We have simply called this version of *FrozenLake* as *MyFrozenLake*. Cliff, bridge and ZigZag are just an instance of *MyFrozenLake* with a fixed 6x10 grids map (in figure 7.10) described in Biedenkapp et al., 2021 [5].



(a) Cliff (b) Bridge (c) Zigzag

Figure 7.10: Tabular Gridworlds. Red cells denote the starting state and blue cells the goal state.

These fixed environments have the purpose to represent common pattern found in maze, to test how agents react to these obstacles. We have also tested our tabular version with different seeds of a general random *MyFrozenLake* with dimension 8 and 16. An instance of these environments is provided in figure 7.11



(a) FroznLake 8 (b) FroznLake 16

Figure 7.11: Generated MyFroznLake example. Red cells denote the starting state and blue cells the goal state.

These environments are deterministic, and the outer borders block the agent from moving outside the grid (for example, an agent being at the top left cell will not move with an *Up* action). All the environments have a maximum step limit of 100 steps. For all the environments we have used the following parameters.

Parameters:

- Initial estimation: $Q(s, a, k) \sim \mathcal{N}(0, 1)\ \forall (s, a, k) \in \mathcal{S} \times \mathcal{A} \times \mathcal{K}$;
- Discount factor: $\gamma = 0.99$;
- Learning rate: $\alpha = 0.01$;
- Maximum number of iterations: 6000 for FrozenLake, otherwise 600;
- Random policy probability: Exponentially decreasing: $\epsilon_t = 0.99^t$.

7.2.2 Mountain Car

Mountain Car (Moore, 1991 [22]) is a well known environment in the Reinforcement learning landscape. Our agent can control a car in a one dimensional space. The possible actions are, accelerate left, accelerate right and do not accelerate. The target of the car is to drive up a steep hill (Figure 7.12).

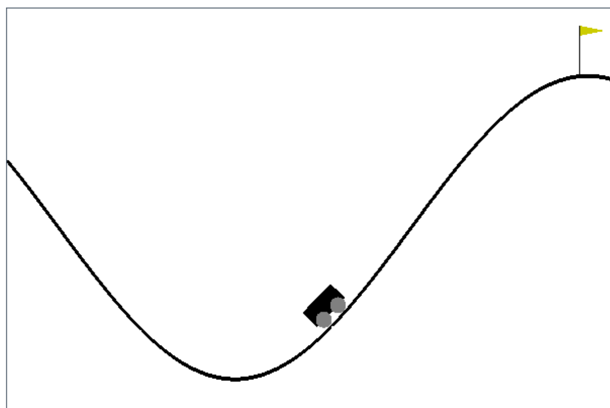


Figure 7.12: Mountain Car OpenAI Gym implementation.

The architecture chosen is a MLP: the first two hidden layers, consisting of 128 rectifier linear units, are shared among all persistences. The third hidden layer instead is diversified for each persistence value k, and each one is composed of 64 rectifier neurons and connected to three outputs, one for each action with its own persistence value.

The parameters adopted for the experiments are the following.

Parameters:

- Discount factor: $\gamma = 1$;
- Maximum number of iterations: 6×10^5 (truncated to 5×10^5 in the plots);
- Batch size: 32 for each persistent value;

- Random policy probability: linearly decreasing, starting from $\epsilon_0 = 1$, to a final value $\epsilon_f = 0.01$, reached at 15% of the total number of iterations;
- Target update frequency: every 1000 steps;
- Prioritized replay buffers, of size 5×10^4 for each persistence value, and default prioritization coefficients $\alpha_p = 0.6$, $\beta_p = 0.4$;

7.2.3 Atari

We will introduce all Atari environments used during our tests. They are all Atari 2600 games simulated through the Arcade Learning Environment.

Freeway We have to control a chicken up and down and our purpose is to cross ten lanes of an highway where cars drive horizontally across the map, they are our enemy. Every time we successfully cross the street we get a point. If a car hits the chicken, it is forced to go back some steps. The game lasts 2 minutes and the winner is the player that has scored the most points in the slot time.

Enduro It is a racing video game. The target is simple, every day during the race we have to surpass a certain number of cars to be able to continue the next day. The player can accelerate and steer left and right.

Qbert The main character of the game is Q*bert trapped in a pyramidal grid composed by cubes. He has to jump above one cube to change its color. Our goal is to change the color of each cube, but the rules change every match. Sometimes if we jump in the same cube two times the color returns to the original one or we have to jump twice in every block. There are also enemies that are able to kill the player.

Kangaroo We are a kangaroo and we need to save our child. The player can walk left and right, can climb the stairs and kick the enemies.

MsPacman This is the classic Pacman game, a maze arcade game. The player needs to collect all the sphere and avoid ghosts. Every match the ghosts' speed increase.

Seaquest We are under the sea with a submarine. The player needs to rescue every divers and avoid sharks and enemies' submarine. He can also shoot enemies. The submarine has an oxygen bar and all the time it needs to fill the tank again by reaching the surface.

For Atari games, the architecture chosen is based on that presented in (Mnih, Kavukcuoglu, Silver, et al., 2015 [21]), with three convolutional layers. the first hidden layer takes as input an $84 \times 84 \times 4$ image and convolves 32 filters of 8×8 with stride 4, with the application of rectifier nonlinearities. The second has 32 input channels, 64 output channels, a kernel size of 4 and a stride of 2, again with ReLu activations, as well as the third convolutional layer, with kernel size of 3 and a stride of 1, and 64 output channels. The convolutional structured is shared among all persistences, while the fully-connected hidden layer, consisting of 128 rectifier units, is differentiated for each persistence value k. Each one of these layers is fully-connected with the output layer, with a single output for each possible action.

All the OpenAi Gym environments used are in in the *deterministic-v0* version (e.g. *FreewayDeterministic-v0*), which does not make merging operations among the 4 input frames, but considers only the last one.

The parameters adopted are the following.

Parameters:

- Discount factor: $\gamma = 0.99$;
- Maximum number of iterations: 2.5×10^6;
- Batch size: 32 for each persistent value;
- Random policy probability: linearly decreasing, starting from $\epsilon_0 = 1$, to a final value $\epsilon_f = 0.01$, reached at 17% of the total number of iterations;
- Target update frequency: every 500 steps;
- Prioritized replay buffers, of size 5×10^4 from each persistence value, and default prioritization coefficients $\alpha_p = 0.6$, $\beta_p = 0.4$.

7.3 Results

We will present now the results of our experiments. We have launched our algorithm with different K_{max} to see how the persistence parameter influences the convergence of our agent. We have also compared our method with classic Q-learning and DQN, for Mountain Car and Atari games, and also with TempoRL (Biedenkapp et al., 2021 [5]) discussed in section 3.2.2.

7.3.1 Tabular

In figure 7.13 are shown the performances of Persistent Q-learning with different $K_{max} \in \{4, 8, 16\}$. We can detect a faster convergence when passing from $K_{max} = 4$ to 8. However, the largest value of K_{max} is not always the best one: while Bridge and Cliff show a slight improvement, performances in ZigZag and FrozenLake degrade. This is probably due to the nature of the environment. When there are many obstacles, high persistences might be inefficient, as the agent can get stuck or reach holes more easily.

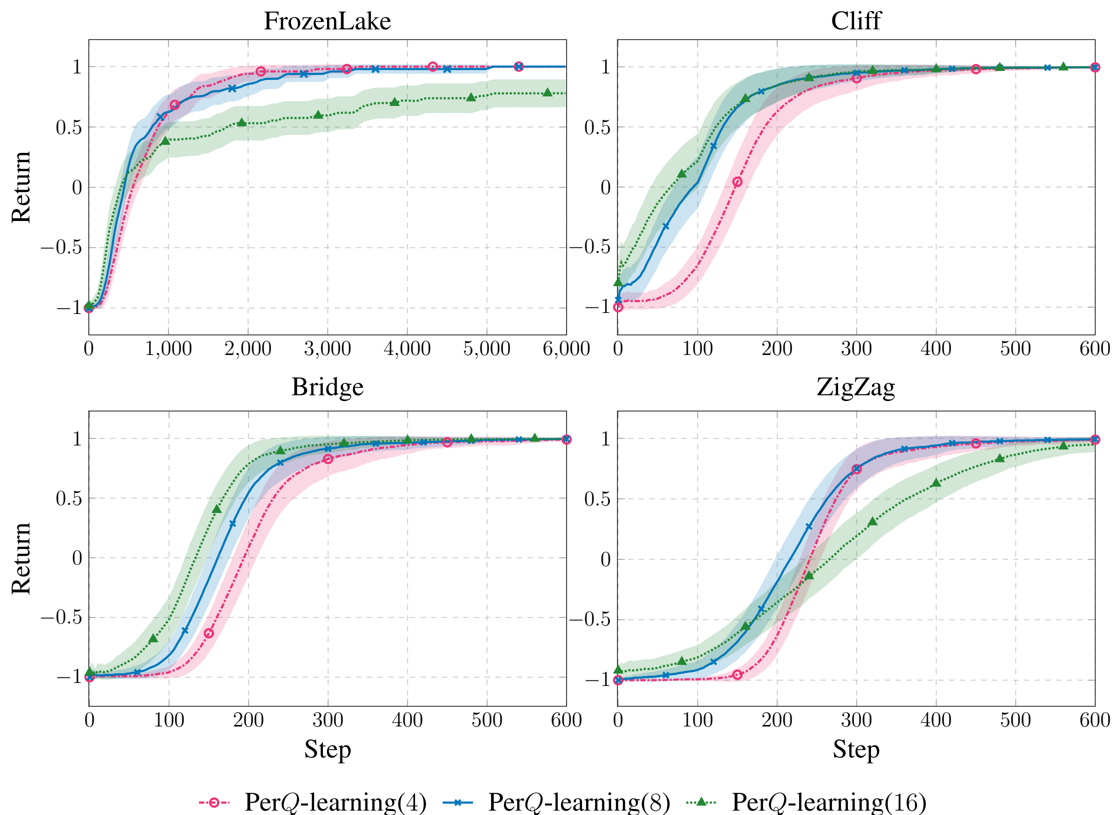


Figure 7.13: Comparison of Persistent Q-learning with different persistences in all tabular environments. In the legend, parenthesis denote the selected $K_{\max}$. 50 runs (avg± 95% c.i.).

In figure 7.14 we present the comparison between Q-learning, Per$Q$-learning, with $K_{max} = 8$ and TempoRL, with skip-length $J = 8$. With this configuration our method outperform others algorithms, especially Q-learning.

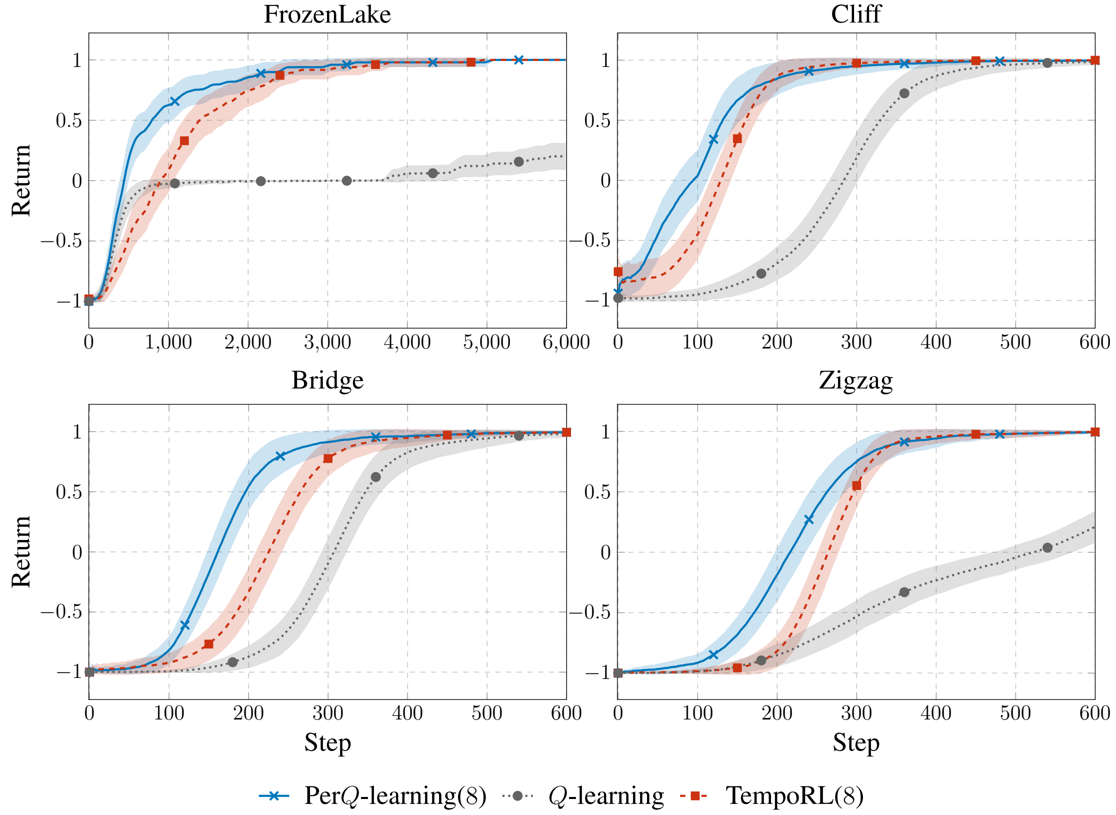


Figure 7.14: Comparison of Q-learning, Persistent Q-learning and TempoRL. Per$Q$-learning and TempoRL with persistence 8. 50 runs (avg± 95% c.i.).

However, increases the K_{max} parameters is not always the best solution. As we can see in figure 7.15 and 7.16, our method shows faster rates of improvements than TempoRL, especially in first learning iterations, but this advantage may not be consistent for every value of K_{max}. TempoRL is more robust with higher persistences, as in complex environments such as ZigZag and FrozenLake, the performance does not degrade as Persistent Q-learning.

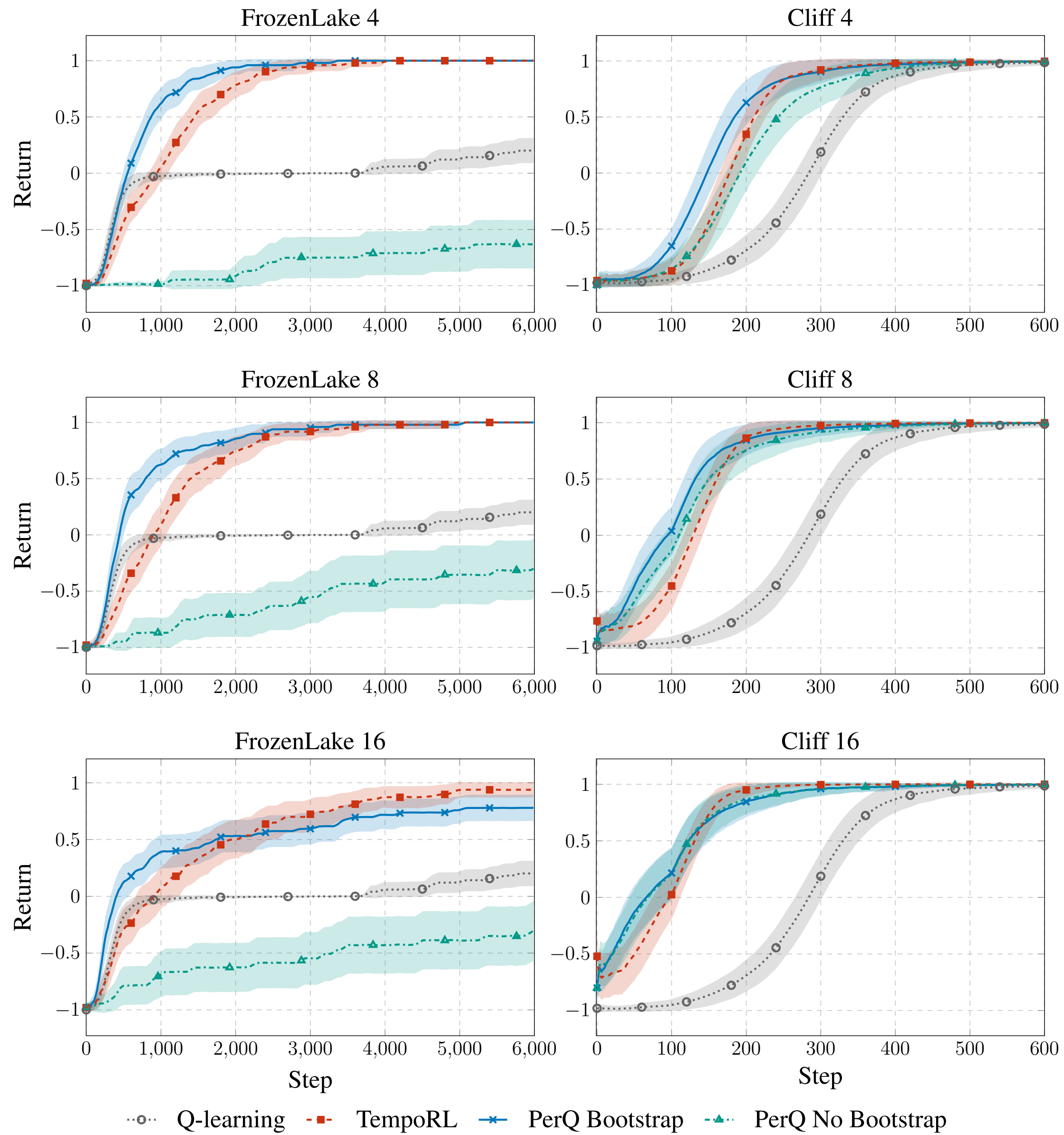


Figure 7.15: Results of Q learning, TempoRL, PerQ with and without bootstrap in different tabular environments and maximum persistences. On each row, a different maximum persistence is selected for both algorithms. 50 runs (avg± 95% c.i.).

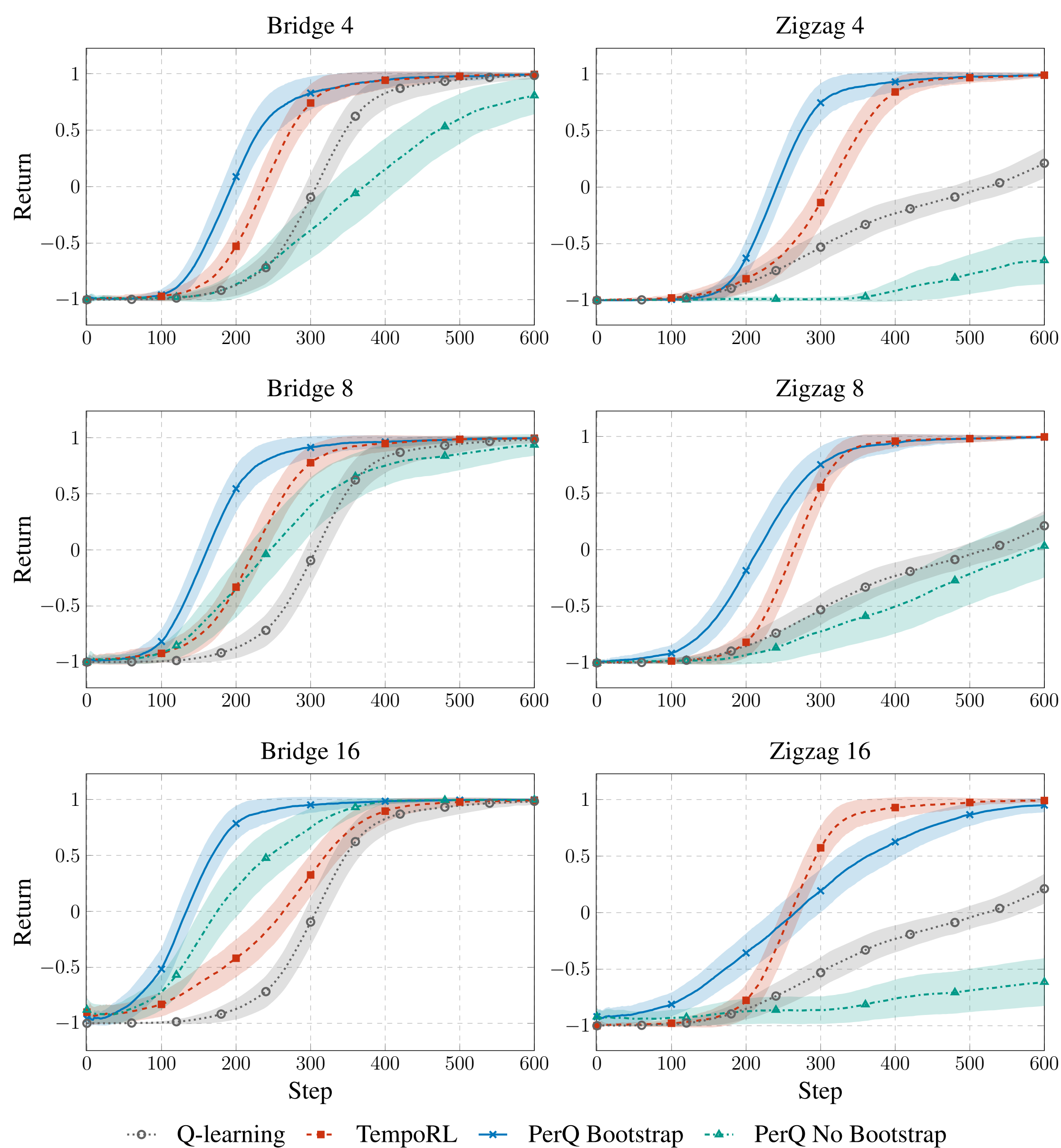


Figure 7.16: Results of Q learning, TempoRL, PerQ with and without bootstrap in different tabular environments and maximum persistences. On each row, a different maximum persistence is selected for both algorithms. 50 runs (avg± 95% c.i.).

7.3.2 Mountain Car

Our algorithm is well suited for this environment, in fact, as shown in Metelli et al., 2020 [19], 1-step explorative policies usually fail to reach the goal because of their low probability to commit to an action for long times. As we can see in figure 7.17 both DQN and TempoRL are not able to converge, in the time step window considered, to the optimal policy, while PerDQN can reach the optimal solution in a fast way compared to the others methods. To show the effectiveness of our bootstrapping methods, we have also compared PerDQN with different K_{max} and bootstrap feature disabled. As shown in figure 7.18, bootstrap is an essential feature for PerDQN to converge rapidly to the optimal policy and to be robust.

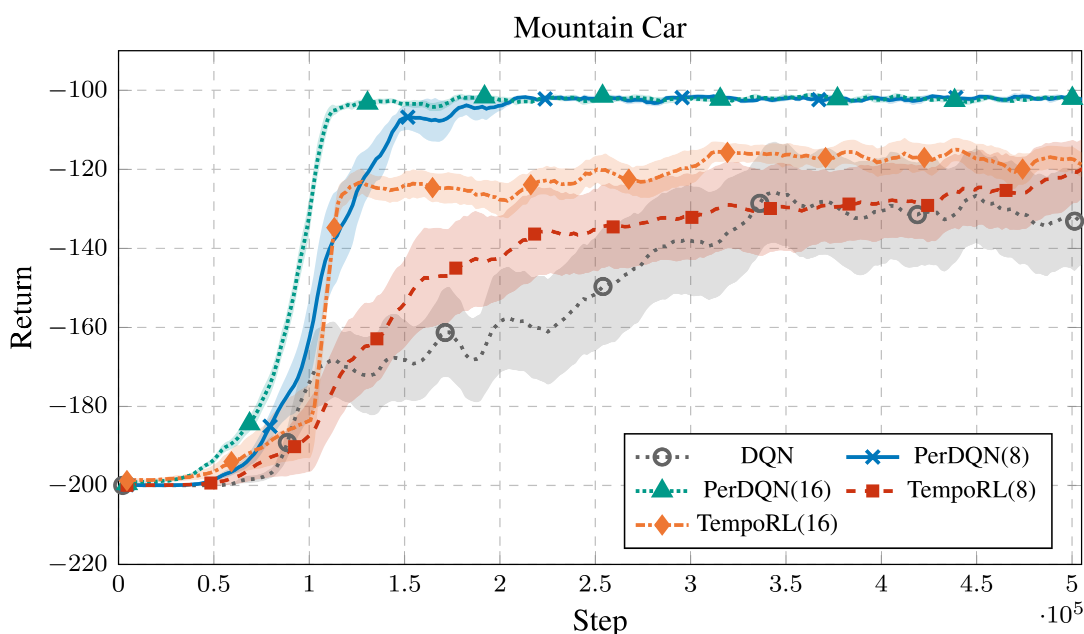


Figure 7.17: MountainCar results for DQN, PerDQN and TempoRL. Parenthesis in the legend denote $K_{\max}$. 20 runs (avg± 95% c.i.).

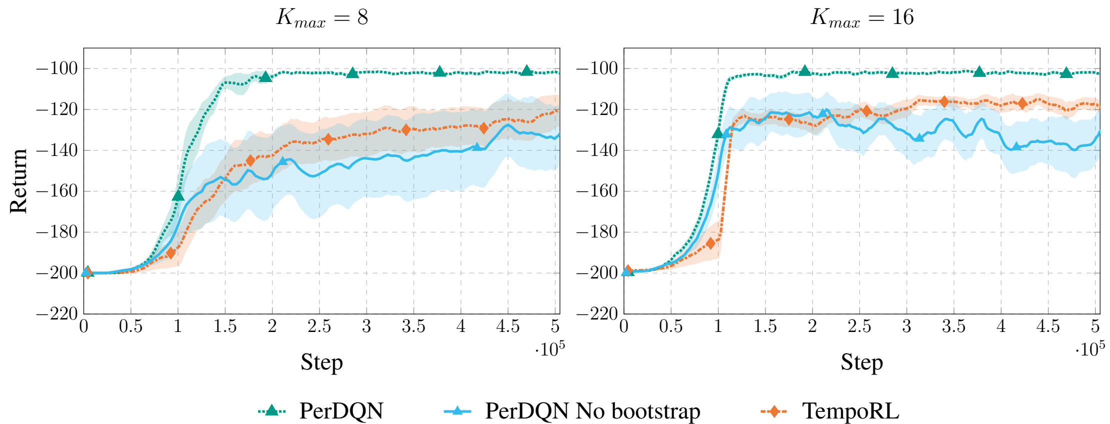


Figure 7.18: PerDQN additional results on MountainCar. Return comparison of PerDQN with and without bootstrap and TempoRL. 20 runs (avg ± 95%c.i.)

7.3.3 Atari

In Figure 7.19 we compare PerDQN, with $K_{\max} = 4$ and $K_{\max} = 8$, and classic DQN. In five games out of six, our PerDQN displays a faster learning curve thanks to its ability of reusing experience, although in some cases (e.g. Kangaroo) PerDQN seems to inherit the same instability issues of DQN, we conjecture due to the overestimation bias (Van Hasselt et al., 2016 [37]). Moreover, we notice that, surprisingly, in Seaquest (figure 7.19 the last chart), persistence seems to be detrimental for learning, as DQN clearly outperforms PerDQN. In this task, agents have to choose either to move or to shoot some moving targets. Persisting the shooting action, thus, may force the agent to stay still for a long time, hitting nothing. A possible solution could consist in the introduction of *interrupting persistence*, in a similar fashion to interrupting options (Richard S. Sutton et al., 1999 [35]; Mankowitz et al., 2014 [18]), which is an interesting future research direction.

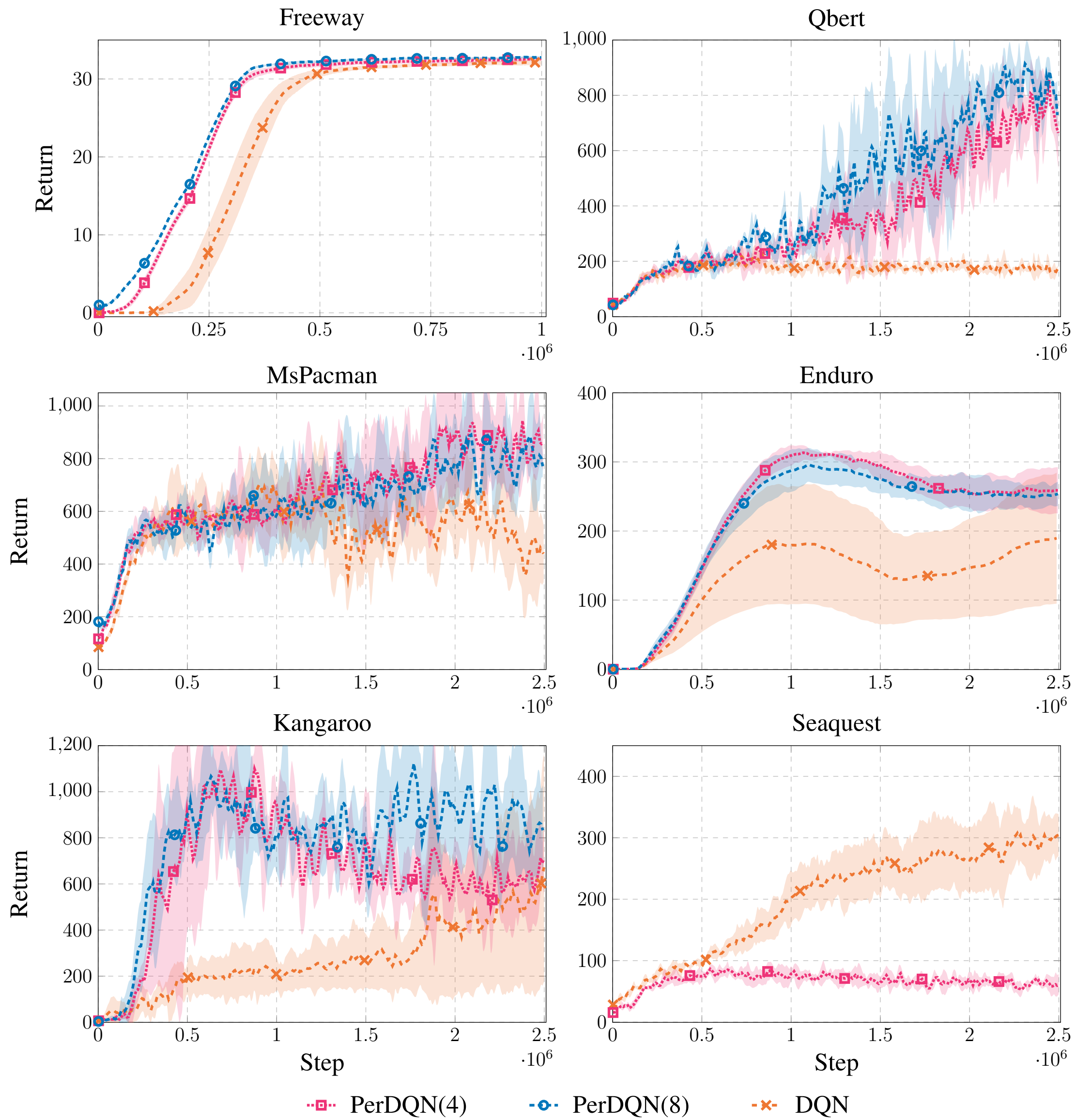


Figure 7.19: Atari games results for DQN and PerDQN, with $K_{\max} = 4$ and 8. 5 runs (avg$\pm$ 95% c.i.).

In order to better understand which beneficial effects are provided by action persistence alone and which ones derive from the use of bootstrap operator, we run an ablation experiment. The results, reported in Figure 7.21 and 7.20 as well, show that PerDQN always dominates over its counterpart without bootstrap. The latter one performs similarly to standard DQN on different environments, without reaching the outstanding performance of PerDQN with bootstrap, and it even obtains an average lower performance w.r.t. to the classic DQN on the other tasks.

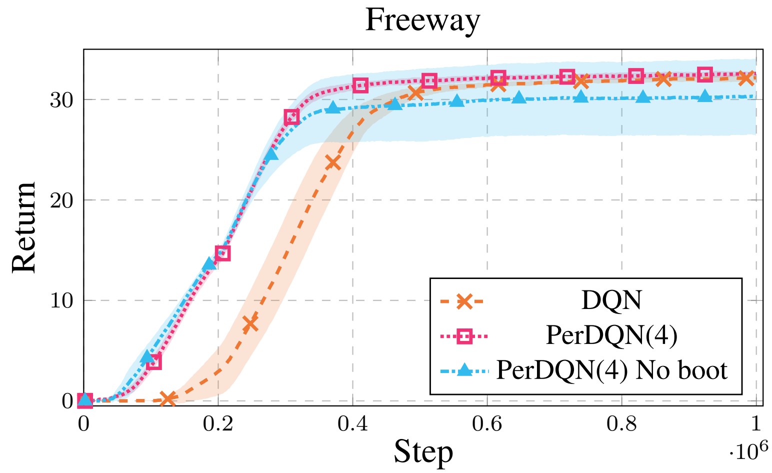


Figure 7.20: Freeway games results for DQN and PerDQN, comparing the version with bootstrapping disabled, with $K_{\max} = 4$. 5 runs (avg± 95% c.i.).

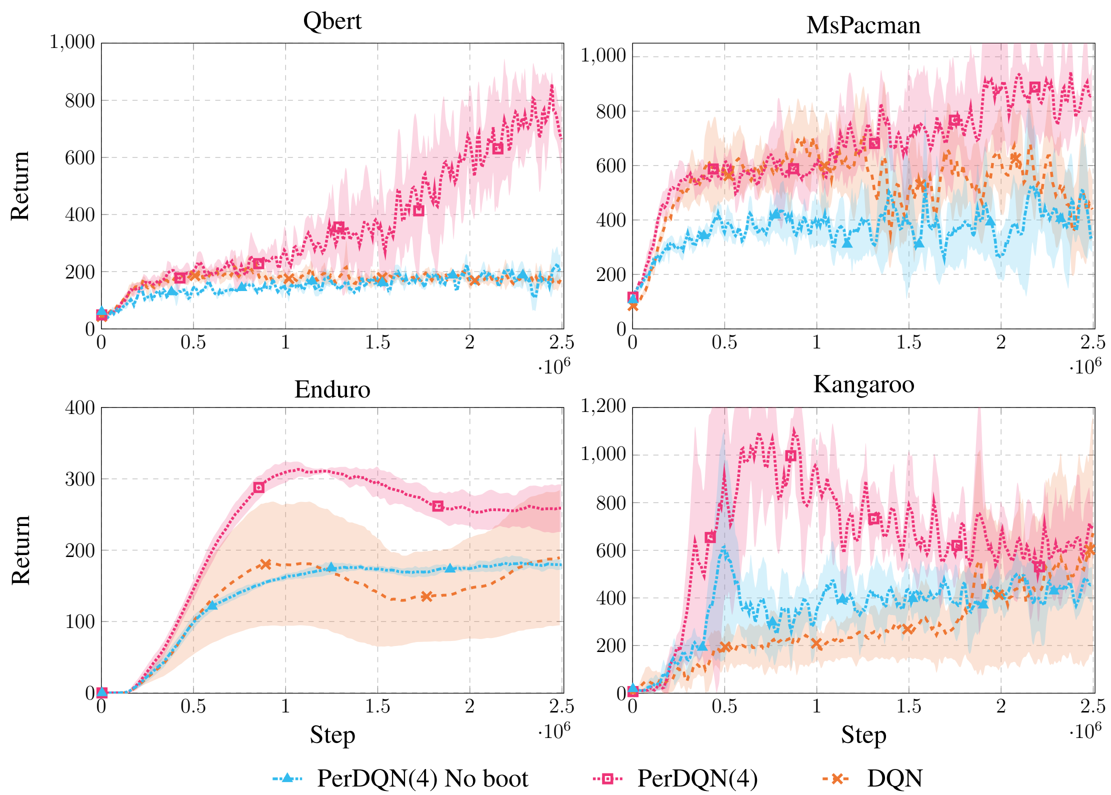


Figure 7.21: Atari games results for DQN and PerDQN, comparing the version with bootstrapping disabled, with $K_{\max} = 4$. 5 runs (avg± 95% c.i.).

We have also compared our method with the one described by TempoRL (Biedenkapp et al., 2021 [5]). Their code is implemented with Pytorch (`https://github.com/automl/TempoRL`). The results are shown in Figures 7.23 and 7.22. As we can see, PerDQN outperforms TempoRL in all the environments evaluated.

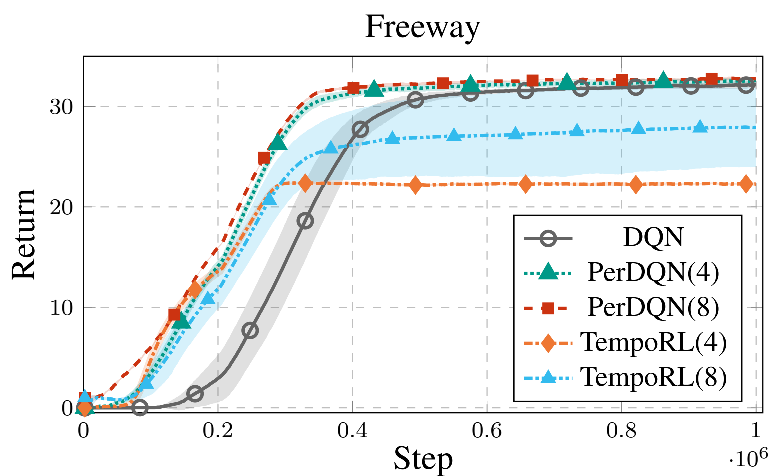


Figure 7.22: Freeway games results for DQN, PerDQN and TempoRL, with $K_{\max} = 4$ and $K_{\max} = 8$. 5 runs (avg± 95% c.i.).

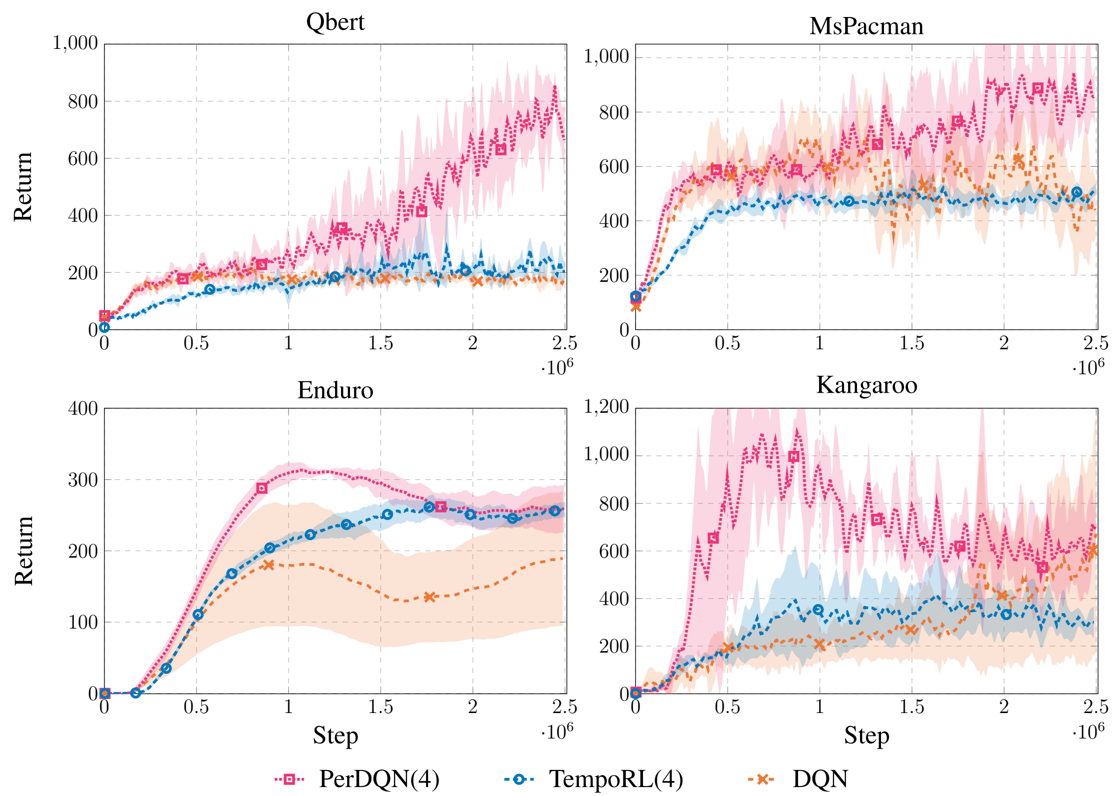


Figure 7.23: Atari games results for DQN, PerDQN and TempoRL, with $K_{\max} = 4$. 5 runs (avg± 95% c.i.).

Chapter 8

Conclusions and Future Work

The last chapter provides a summary of the results obtained and some suggestions about possible future research.

8.1 Conclusion

This thesis aimed to find a way to include a control frequency mechanism inside a Reinforcement Learning context. Starting from a previous work (Metelli et al., 2020 [19]) about a deep analysis of *action persistence* in a batch Reinforcement Learning settings and is advantages compared to a classical settings, we have developed an agent able to choose, in an online context, the right persistence.

We have presented a general discussion about the basic tools in the Reinforcement Learning field, after that, we have briefly explained techniques developed by other authors to take into account persistence.

We have developed a formalism that implements the action persistence as *persistence options*. Thanks to this, our agent is able to select at the same time both a primitive action and its duration. After that we have defined a new operator, the *all-persistence* Bellman operator, which allows for an effective use of the experience collected from the interaction with the environment at any time scale. Thanks to this operator, action-value function estimates can be updated simultaneously on the selected persistence set. In particular, low persistences (and primitive actions) can be updated by splitting the samples in their sub-transitions; action value functions for high persistences can instead be improved by *bootstrap*, a procedure that takes into account the estimation of the partial missing information.

After proving that the new operator is a contraction, we applied it to extend classic Q-learning and DQN with their persistent version. We performed an empirical analysis to underline the benefits of the new operator for exploration and estimation. Further-

more, the experimental campaign on tabular and deep RL settings demonstrated the effectiveness of our approach and the importance of considering temporal extended actions, as well as some limitations.

8.2 Future work

Future research directions include:

Other exploration methods We have used an extension of the ε-greedy exploration that include persistence. It would be interesting to extend other exploration methods.

Policy gradient methods One limitation of value-based approaches is that they cannot operate with continuous state and action spaces. One possible future work can be the development of an extension of gradient-based method, like Actor-critic algorithms, to incorporate the persistence.

Persistence interruption When we introduce the persistence in Reinforcement Learning, we renounce to the ability of selecting action in the middle states. Our agent chooses the action and a persistence and it executes the first for an amount of time steps equal to the persistence. If from one side we have the benefit to travel between different regions of state space, from the other side we can get stuck in some states. For example, if our agent chooses an high persistence in front of a wall, it is forced to go on for a long time, without the ability to change idea. It would be great to implement a mechanism to recognize a blocked region and, eventually, stop the persistence.

Separate persistence If some actions have a low intrinsic persistence, for example the shot action in Seaquest Atari game, it would be great to incorporate this prior information into our model. A possible solution is to separate these actions from the original model, using a second one with a lower max persistence parameter.

Persistence Advantage function With dueling in Deep Q-network settings we use an Advantage function to capture more specific feature. A possible improvement is the creation of an advantage function that include also the persistence, mixing, for example, the Q-function or the Value function with different persistence.

Bibliography

[1] Susan Amin, Maziar Gomrokchi, Hossein Aboutalebi, Harsh Satija, and Doina Precup. "Locally Persistent Exploration in Continuous Control Tasks with Sparse Rewards". In: *arXiv preprint arXiv:2012.13658* (2020).

[2] Kai Arulkumaran, Nat Dilokthanakul, Murray Shanahan, and Anil Anthony Bharath. "Classifying options for deep reinforcement learning". In: *arXiv preprint arXiv:1604.08153* (2016).

[3] Leemon C Baird. "Reinforcement learning in continuous time: Advantage updating". In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*. Vol. 4. IEEE. 1994, pp. 2448–2453.

[4] Richard Bellman. "Dynamic programming and stochastic control processes". In: *Information and Control* 1.3 (1958), pp. 228–239. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(58)80003-0. URL: https://www.sciencedirect.com/science/article/pii/S0019995858800030.

[5] André Biedenkapp, Raghunandan Rajan, Frank Hutter, and Marius Thomas Lindauer. "TempoRL: Learning When to Act". In: *ICML*. 2021.

[6] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. "Frame skip is a powerful parameter for learning to play atari". In: *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[7] Minerva Catral, Stephen J Kirkland, Michael Neumann, and N-S Sze. "The Kemeny constant for finite homogeneous ergodic Markov chains". In: *Journal of Scientific Computing* 45.1 (2010), pp. 151–166.

[8] Will Dabney, Georg Ostrovski, and Andre Barreto. "Temporally-Extended ϵ-Greedy Exploration". In: *International Conference on Learning Representations (ICLR)*. 2020.

[9] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[10] Thomas G. Dietterich. “The MAXQ Method for Hierarchical Reinforcement Learning”. In: *Proceedings of the Fifteenth International Conference on Machine Learning (ICML)*. Ed. by Jude W. Shavlik. Morgan Kaufmann, 1998, pp. 118–126.

[11] Faustino J. Gomez and Jürgen Schmidhuber. “Co-Evolving Recurrent Neurons Learn Deep Memory POMDPs”. In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. GECCO ’05. Washington DC, USA: Association for Computing Machinery, 2005, pp. 491–498. ISBN: 1595930108. DOI: 10.1145/1068009.1068092. URL: https://doi.org/10.1145/1068009.1068092.

[12] Michael Kearns and Satinder Singh. “Finite-sample convergence rates for Q-learning and indirect algorithms”. In: *Advances in Neural Information Processing Systems (NIPS)* (1999), pp. 996–1002.

[13] Steve Kirkland. “Fastest expected time to mixing for a Markov chain on a directed graph”. In: *Linear Algebra and its Applications* 433.11-12 (2010), pp. 1988–1996.

[14] Aravind S. Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. “Dynamic Action Repetition for Deep Reinforcement Learning”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 2133–2139.

[15] Espeholt Lasse, Soyer Hubert, Munos Remi, Simonyan Karen, Ward Tom, Doron Yotam, Mnih Volodymyr, Firoiu Vlad, Harley Tim, Dunning Iain, Legg Shane, and Kavukcuoglu Koray. “IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures”. In: (2018). arXiv: 1802.01561. URL: https://arxiv.org/pdf/1511.05952.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[17] Richard Lippmann. “An introduction to computing with neural nets”. In: *IEEE Assp magazine* 4.2 (1987), pp. 4–22.

[18] Daniel J Mankowitz, Timothy A Mann, and Shie Mannor. “Time regularized interrupting options”. In: *Internation Conference on Machine Learning (ICML)*. 2014.

[19] Alberto Maria Metelli, Flavio Mazzolini, Lorenzo Bisi, Luca Sabbioni, and Marcello Restelli. “Control frequency adaptation via action persistence in batch reinforcement learning”. In: *International Conference on Machine Learning (ICML)*. PMLR. 2020, pp. 6862–6873.

[20] Volodymyr Mnih, Koray Kavukcuoglu, Alex Graves, Ioannis Antonoglou, David Silver, Daan Wierstra, and Martin Riedmiller. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[22] Andrew William Moore. "Efficient memory based learning for robot control". In: *PhD Thesis, Computer Laboratory, University of Cambridge* (1991).

[23] Seohong Park, Jaekyeom Kim, and Gunhee Kim. "Time Discretization-Invariant Safe Action Repetition for Policy Gradient Methods". In: *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021).

[24] Ronald Parr and Stuart Russell. "Reinforcement learning with hierarchies of machines". In: *Advances in Neural Information Processing Systems (NIPS)* (1998), pp. 1043–1049.

[25] Rushabh Patel, Pushkarini Agharkar, and Francesco Bullo. "Robotic surveillance and Markov chains with minimal weighted Kemeny constant". In: *IEEE Transactions on Automatic Control* 60.12 (2015), pp. 3156–3167.

[26] Doina Precup. "Temporal abstraction in reinforcement learning." PhD thesis. University of Massachusetts Amherst, 2001.

[27] Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 2014.

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

[29] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[30] Ralf Schoknecht and Martin Riedmiller. "Reinforcement learning on explicitly specified time scales". In: *Neural Computing & Applications* 12.2 (2003), pp. 61–80.

[31] Sahil Sharma, Aravind Srinivas, and Balaraman Ravindran. "Learning to repeat: Fine grained action repetition for deep reinforcement learning". In: *arXiv preprint arXiv:1702.06054* (2017).

[32] Aaron Sidford, Mengdi Wang, Xian Wu, Lin F Yang, and Yinyu Ye. "Near-optimal time and sample complexities for solving Markov decision processes with a generative model". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems.* 2018, pp. 5192–5202.

[33] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. "Convergence results for single-step on-policy reinforcement-learning algorithms". In: *Machine learning* 38.3 (2000), pp. 287–308.

[34] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[35] Richard S. Sutton, Doina Precup, and Satinder P. Singh. “Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning”. In: *Artif. Intell.* 112.1-2 (1999), pp. 181–211. DOI: 10.1016/S0004-3702(99)00052-1.

[36] Corentin Tallec, Léonard Blier, and Yann Ollivier. “Making Deep Q-learning methods robust to time discretization”. In: *Proceedings of the 36th International Conference on Machine Learning (ICML).* Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6096–6104.

[37] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on Artificial Intelligence (AAAI).* Vol. 30. 1. 2016.

[38] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. “Dueling network architectures for deep reinforcement learning”. In: *International conference on machine learning.* PMLR. 2016, pp. 1995–2003.

[39] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. PhD thesis. King’s College, University of Cambridge, 1989.

[40] Haonan Yu, Wei Xu, and Haichao Zhang. “TAAC: Temporally Abstract Actor-Critic for Continuous Control”. In: *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021).