Master Thesis

Internship Report

Introduction To Safe Reinforcement Learning

Supervisor: **Goran FREHSE** (ENSTA Paris)

Tutor: **Luc JAULIN** (ENSTA Bretagne)

September 2020

David BRELLMANN - CI 2020 - ROB

Abstract

Recent advances in artificial intelligence (AI) have contributed to a proliferation of autonomous agents such as vehicles, drones, and inspection robots. Often, these autonomous agents are used in environments where it is essential that the AI avoids accidents (collisions, battery depletion) and they must thus respect hard constraints. With the popularization of Reinforcement Learning methods, it is even more difficult to check the safety of the system because they are based on approximations.
The aim of this project is to study Reinforcement Learning methods using neural networks, also called Deep Reinforcement Learning methods, on a simulated Cart-Pole environment as a case study. Deep RL methods are compared with each other based on their training performance as well as on their safety on both deterministic and non-deterministic versions of the environment. In addition to the choice of a Reinforcement Learning method, this report highlights the importance of choosing a good features encoding. Finally, we work towards defining suitable metrics for quantifying the safety of a controller.

All the implementations made during this project are available at `https://github.com/DavidBrellmann/DeepRL`.

Keywords: Reinforcement Learning, Deep Reinforcement Learning, safety, neural network, features encoding, metrics, external disturbance, Cart-Pole.

Résumé

Les récents progrès en l'Intelligence Artificielle (IA) ont contribué à la prolifération d'agents autonomes comme des robots, des drones et des robots d'inspection. Souvent, ces agents autonomes sont utilisés dans des environnements où il est nécessaire pour l'IA d'éviter les accidents (collisions, batterie vide) et par conséquent ils doivent donc respecter des contraintes strictes. Avec la popularisation des méthodes d'Apprentissage par Renforcement, il est encore plus difficile de vérifier la sécurité d'un système parce que ces méthodes sont basées sur des approximations.
Le but de ce projet est d'étudier les méthodes d'Apprentissage par Renforcement utilisant les réseaux de neurones (également appelées méthodes d'Apprentissage par Renforcement Profond) sur un environnement de Cart-Pole simulé. Les méthodes d'apprentissage par renforcement profondes sont comparées entre elles selon leurs performances à l'entraînement et selon leurs capacités à générer des contrôleurs sûrs sur des versions déterministes et non-déterministes de l'environnement. Outre le choix d'une méthode d'apprentissage du renforcement, ce rapport souligne l'importance de choisir de bons descripteurs. Enfin, ce projet vise à définir des métriques appropriées pour pouvoir quantifier la sécurité d'un contrôleur.

Toutes les implémentations réalisées lors de ce projet sont disponible sur `https://github.com/DavidBrellmann/DeepRL`.

Mots-clés: Apprentissage par Renforcement, Apprentissage par Renforcement Profond, sécurité, réseaux de neurones, descripteurs, métriques, perturbation extérieure, Cart-Pole.

Acknowledgements

I would like to thank Goran FREHSE profusely for the opportunity of doing my final project with him, but above all for guiding me relentlessly and for all the benefits he provided me during this internship. It was not easy with Covid but he was always available and listening to help me.
I could not forget to mention David FILLIAT and Pavan VASISHTA for their external view on my work and for their valuable advice. I thank both for their interest on my project. I also would like to thank all the members of U2IS for the great ambiance at the office and for welcoming me at ENSTA Paris.
Finally, I would like to thank Luc JAULIN who coordinates the Robotics branch at ENSTA Bretagne. His teaching, his advice and his pedagogical qualities encouraged me to continue in research

Contents

Chapter 1

Introduction

Reinforcement Learning (RL) is an important subfield of Machine Learning which addresses the problem of automatic learning of optimal decisions over time. Reinforcement Learning gained popularity in the 1990s and and since the 2010s received more traction due to the use of neural networks in Machine Learning. Contrary to Supervised Learning, there are no external supervisors to help the learner to make the right decision with a training set of labelled input/output pairs.
The learner needs to interact with its environment in uncharted territories. A **learner/agent** interacts directly with its environment in order to know which action to take at each time step. The goal is to maximize a numerical reward signal by choosing correct actions. However, the learner is not told which actions to take, but instead must discover which action yields the most reward by trying them (**trial and error**). Besides, actions can not affect only the immediate reward and situation but also the next situation and, through that, all subsequent rewards (**delayed rewards**). For this reason, one of the major problems in Reinforcement Learning is the trade-off between **exploration** (by trying different actions) and **exploitation** (by using the current knowledge).

One of the main reasons of the popularity of Reinforcement Learning is that these methods do not require any specific knowledge of a model. As no model is required, RL methods are able to offer a solution to problems where modeling is very difficult or almost impossible and thus may be an interesting alternative to "conventional methods".
Besides, it is easier to use a RL method on several different tasks without spending time re-adapting the method except for the choice of hyper-parameters. Although, it has been shown that a same set of hyper-parameters can be used on a large number of environments, this was the case for example for the DQN which was tested on 49 Atari games [1]. Nevertheless, their use in robotic control is challenging and it is the focus of several recent works [2, 3, 4]. Indeed, the behavior and the efficiency of these methods in real-world domains is not very well known. Because there is a learning process, one can hope that the method learns, understands and anticipates external disturbances on the system such as noise, delay...

One of the main objectives to be met by the agent in the real-world is the **safety** of the system, both after and during the learning process. In most applications, some states are dangerous because they may deteriorate the system, they lead the system to its self-destruction or its malfunction. Such states are called **forbidden states** and they are defined by **constraints**.

States from which the controller can ensure that the system never enters the forbidden states are called **safe**. A priori, we only know which states are forbidden, not which states are safe. A controller is said safe if the controller always evolves in safe states. Respecting safety requirements are essential for the proper functioning of the system because a violation of these requirements leads to a deterioration or a destruction of the system. Several recent papers treat the safety of RL methods during training but this report considers the safety of a controller during operations, i.e., when the training is over. For quantitative evaluation, safety metrics are discussed that can be used for both conventional and learned controller.

In this context, the objectives of this internship are:

- to study and understand the most well-known model-free RL methods,
- to define a protocol and metrics in terms of both performance and safety to compare RL methods with each other as well as with conventional methods (PID),
- to test the RL methods on a robotic control task (the Cart-Pole task) on deterministic and non-deterministic environments with disturbances (such as noise, delay, ...).

1.1 The Host Organization

ENSTA Paris, formerly known as École Nationale Supérieure de Techniques Avancées, is a prestigious French engineering school. Founded in 1741, it is the oldest engineering school in France. It is located in Palaiseau in the south of Paris, on the Paris-Saclay campus. ENSTA Paris is also a founding member of Institut Polytechnique de Paris (IP Paris), a leading educational and research institution in France and internationally that brings together five prestigious French engineering schools: École Polytechnique, ENSTA Paris, ENSAE Paris, Télécom Paris and Télécom SudParis.

The Computer Science and Systems Engineering Laboratory (U2IS) is developing research in the field of design and reliability of systems integrating autonomous decision-making processes with applications in intelligent transport, robotics, defense and energy. The laboratory brings together the research activities of ENSTA Paris in computer science, robotics, vision, embedded systems, signal and image processing and hybrid system design and analysis. The laboratory is focusing on application-oriented research. Even if they can come from fundamental fields, the motivation of most of its work is to be able to be integrated in products (components, robotic systems, design or validation tools) that are evaluated finally by the industry and can give place to technology transfers.

1.2 Contributions

In this report, we are going to propose:

- a state of the art of the main Reinforcement Learning and Deep Reinforcement Learning methods (Chapter 2 and Chapter 4), with some slight reformulations for unifying several research works (Chapter 2).
- the use of Fourier Basis as features encoding for Deep RL methods to improve the performance (Chapter 4). As in Supervised Machine Learning, there exists several features encodings for classical Reinforcement Learning methods. Fourier Basis [5] is known as efficient features encoding for classic RL methods (Chapter 3). However, to the best of my knowledge, it seems that few, if any, features encoding are used with Deep RL methods.
- a comparison of the training performance of Deep RL methods and of the safety after the training. Performance were evaluated on a standard simulated environment (OpenAI GYM) as well as on a non-deterministic environments with the addition of noise and delays (Chapter 4). Such external disturbances are important for closing the simulation to real gap.
- a set of metrics measuring the safety requirements based on state constraints (Chapter 6) for a given controller that can be either conventional or learned by a RL method. Three types of metrics are suggested to correctly quantify the safety of a controller: a distance to the forbidden states, a quantitative interpretation of the Signal Temporal Logic and an approximation of the confidence interval for the probability distribution.

All the implementations made during this project are available at `https://github.com/DavidBrellmann/DeepRL`.

1.3 Structure of the Report

Chapter 2 describes the basic framework of Reinforcement Learning and the main classical RL methods. A comparison of the combinations of RL methods with different features encodings, in particular Fourier Basis, is realized in Chapter 3.
Chapter 4 is an overview of RL methods using neural networks, also called Deep RL methods. In chapter 5, the Deep RL methods are evaluated both on a deterministic environment and on a non-deterministic environment.
The chapter 6 is a discussion of metrics for measuring safety requirements based on states constraints of a given controller that can be conventional or learned by a RL method.
Throughout this report, all simulations were made on a simulated Cart-Pole environment described in section 3.1.

Chapter 2

Introduction To Reinforcement Learning

This chapter recalls the basis of Reinforcement Learning, taking mainly from Sutton's classic book [6] and from Starchurski's book [7]. The purpose of this chapter is to standardize notations and notions of Reinforcement Learning found in the litterature. Notations of both books are combined for a better clarity. The following chapter uses the notions developed in this chapter on the Cart-Pole environment.

2.1 Reinforcement Learning Framework

2.1.1 Agent-Environment Interface

An **agent** is both the learner and the decision maker. Anything outside the agent is called the **environment**. The agent and the environment are constantly interacting: the agent by selecting and performing **actions** and the environment by responding to these actions and presenting new situations, the **states**, to the agent. Moreover, the environment also gives rise to **rewards** that the agent seeks to maximize over time through its choice of actions.

2.1.2 Markov Decision Process

Markov Decision Process (MDP) are a mathematically formalism used to describe Reinforcement Learning problems [6, 7]. We denote by $\mathscr{B}(\mathbb{R}^n)$ the Borel subset of $\mathbb{R}^n$ (appendix A.1). A **Markov Decision Process** [1] corresponds to a tuple $(\mathcal{S}, \mathcal{A}, \Gamma, R, P, \psi)$ where:

- $\mathcal{S} \in \mathscr{B}(\mathbb{R}^n)$ is the **state space**.
- $\mathcal{A} \in \mathscr{B}(\mathbb{R}^m)$ is the **action space**

[1] There is no really universal definition for MDP, the proposed definition aims to unify most definitions found in the litterature

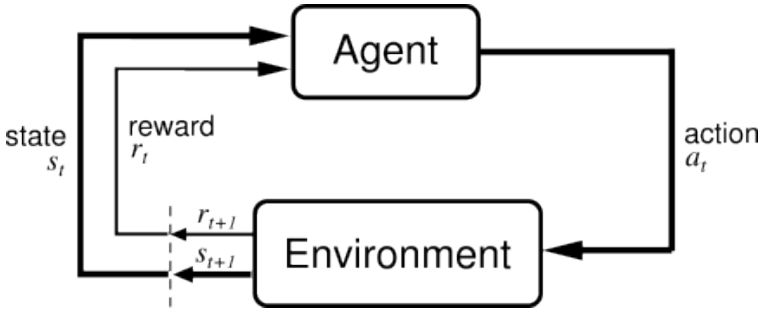


Figure 2.1: Interactions Agent-Environment (MDP process) - At each time step t, the agent receives from the environment a **state** $s_t \in \mathcal{S}$, and on that basis takes an **action** $a_t \in \mathcal{A}$. One time step later, the environment gives a **reward** $r_{t+1} = R(s_t, a_t) \in \mathbb{R}$ to the agent for its previous action. The agent finds itself in a new state s_{t+1} and the process is repeated - taken from [6]

- Γ is a mapping $\mathcal{S} \to \mathscr{B}(\mathcal{A})$ where $\Gamma(s)$ corresponds to the collection of all feasible actions for the agent when the state is s. We define $\operatorname{gr}\Gamma$ as the set of feasible states/actions pairs.

$$\operatorname{gr}\Gamma := \{(s, u) \in \mathcal{S} \times \mathcal{A} : u \in \Gamma(s)\}$$

- R is a bounded **reward function** $R : \operatorname{gr}\Gamma \to \mathbb{R}$
- P is stochastic kernel (appendix A.4) on $\operatorname{gr}\Gamma$ called the **transition function** which captures the dynamics of the environment.
- $\psi \in \mathscr{P}(\mathcal{S})$ is a probability measure (appendix A.2) where $\mathscr{P}(\mathcal{S})$ is the set of all probability measures on $(\mathcal{S}, \mathscr{B}(\mathcal{S}))$. At time zero, $s_0 \in \mathcal{S}$ is drawn from ψ.

At time zero, the initial state $s_0 \in \mathcal{S}$ is drawn from ψ. At the start of time t the agent observes the state $s_t \in \mathcal{S}$ and responds with action $a_t \in \Gamma(s_t) \subseteq \mathcal{A}$. After choosing a_t, the agent receives a reward $R(s_t, a_t)$ and the state is updated according to $s_{t+1} = P(s_t, a_t)$. The whole process then repeats, with the agent choosing a_{t+1}, receiving reward $R(s_{t+1}, a_{t+1})$ and so on. The Figure 2.1 resumes the process.

The transition probabilities P depends on nothing other than the current location of the state and not of the other past states. This is called the **Markov Assumption**.

The reward function R is the way of communicating to the agent what to achieve. A same goal may be defined by multiple reward functions. The reward function can be seen as a *"carrot-and-stick"* policy : the agent is rewarded when it performs well and it is punished otherwise.

In practice, it is difficult to find a subset of $\mathbb{R}^n$ which is not in $\mathscr{B}(\mathbb{R}^n)$. Therefore, the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are always elements of $\mathscr{B}(\mathbb{R}^n)$ and $\mathscr{B}(\mathbb{R}^m)$ respectively.

2.1.3 Returns And Episodes

The aim of the agent is to maximize rewards it receives in the long run. RL problems are actually optimization problems.

Indeed, only choosing the action with the highest current reward is not the best strategy: selecting an action with the highest reward may lead the agent to a subspace of the state space $\mathcal{S}$ where there are only low rewards.
For this reason, the agent seeks instead to maximize the **expected return** G_t [6] :

$$G_t = \sum_{i=1}^{T-t} r_{t+i} = \sum_{i=1}^{T-t} R(s_{t+i-1}, a_{t+i-1}) \tag{2.1}$$

where $T \in \mathbb{N}$ is the **time horizon**.

The definition 2.1 makes sense for **episodic tasks** when agent-environment interactions break naturally into subsequences called **episodes**. Each episode ends in a special state called the **terminal state** which is followed by a reset where the next state is drawn from ψ. From one episode to another the time horizon T is not necessarily the same. In this report, the subset $\mathcal{S}^+ \subseteq \mathcal{S}$ denotes the terminal states set.

For some tasks, agent-environment interactions do not always break naturally into identifiable episodes ($T = \infty$). Such tasks are called **continuous tasks**. For continuous task, with the definition 2.1, G_t diverges.
The expected return need to be redefined for continuous task:

$$G_t = \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} = \sum_{i=1}^{\infty} \gamma^{i-1} R(s_{t+i-1}, a_{t+i-1}) \tag{2.2}$$

where $\gamma \in [0, 1]$ is the **discount factor** [6, 7].

G_t in definition 2.2 is the **discounted expected return** [6, 7]. If $\gamma < 1$, the infinite sum in 2.2 is finite. The discount factor γ quantifies how much value the reward will lose in one time-step. It depends on the definition of the problem, because it regulates how much the agent pays attention to the future. It may be used for episodic tasks too as a trick to find a faster suboptimal solution.

- γ close to zero means a **"myopic" evaluation** which is more focused on the present and immediate rewards.
- γ close to one means **"far-sighted" evaluation** which takes future rewards into account more strongly.

The definition 2.1 is equivalent to the definition 2.2 by introducing an **absorbing state** at the end of episodic tasks. Episode termination for episodic task is now considered to be the entering in the absorbing state that transitions only to itself and which generates only null rewards. This process is depicted in the Figure 2.2.

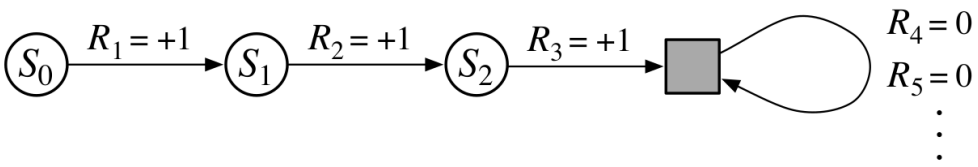


Figure 2.2: Example of an episodic task with an absorbing state (taken from [6])

2.1.4 Policy

In a MDP, the actions that are taken by the agent affect the future path of a state variable. Actions are specified in terms of a **policy** function π. A **policy** π maps the current state $s \in \mathcal{S}$ of the system to a probability measure in $\mathscr{P}(\Gamma(s))$ [6]:

$$\pi(s) \in \mathscr{P}(\Gamma(s))$$

where $\mathscr{P}(\Gamma(s))$ depicts the set of all probability measures on $(\Gamma(s), \mathscr{B}(\Gamma(s)))$[2].

For a better clarity, $\pi(a|s)$ refers to $\pi(s)(a)$ for each state $s \in \mathcal{S}$ and for each action $a \in \Gamma(s)$. Therefore, if an agent is following a policy π at time step t, $\pi(a|s)$ is then the probability to take the action $a_t = a \in \Gamma(s)$ if the state $s_t = s$.

2.1.5 Value Functions

In a MDP, a **value function** determines "how it is good" for an agent following a policy π to be in a state $s \in \mathcal{S}$ using expected discounted returns [6, 7].
Two types of value functions are considered for estimating the value for a specific state and for a pair of state-action.
The **state-value function** $V_\pi : \mathcal{S} \to \mathbb{R}$ of an MDP defined for a policy π is the expected return when starting from $s \in \mathcal{S}$ and then following π:

$$V_\pi(s) = \mathbb{E}_\pi \left[G_t | s_t = s \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R(s_{t+i}, a_{t+i}) | s_t = s \right] \tag{2.3}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows the policy π in a MDP. In addition to the probability π, the transition function P appears in this expectation.
The **action-value function** $Q_\pi : \operatorname{gr}\Gamma \to \mathbb{R}$ of an MDP defined for a policy π is the expected return when starting from s, taking the action $a \in \Gamma(s)$ and then following π.

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | s_t = s, a_t = a \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R(s_{t+i}, a_{t+i}) | s_t = s, a_t = a \right] \tag{2.4}$$

[2]In the same way as for MDP, the definition of a policy is not universal. It varies from a mapping to the action space to a mapping to a probability measure.

2.1.6 Optimal Policy

RL problems are optimization problems. Indeed, the agent aims to increase the cumulative rewards as much as possible and thus to find an **optimal policy** $\pi^* = \arg\max_\pi V_\pi = \arg\max_\pi Q_\pi$ [6, 7].

Value functions define a partial ordering over policies. A policy π is said to be better than or equal to π' if :

$$\pi \geq \pi' \Leftrightarrow V_\pi(s) \geq V_{\pi'}(s), \forall s \in \mathcal{S} \tag{2.5}$$

With the order defined by (2.5), an optimal policy π^* can be also defined as:

$$\pi^* \geq \pi, \forall \pi \tag{2.6}$$

There may be several optimal policies. However all optimal policies share the same value functions :

$$V^*(s) = \max_\pi V_\pi(s), \forall s \in \mathcal{S} \tag{2.7}$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a), \forall (s,a) \in \operatorname{gr}\Gamma \tag{2.8}$$

2.1.7 On-Policy VS Off-policy

Two kind of learning are distinguished in RL. The **On-Policy learning** learns about policy π from experiences sampled from π. So, it learns action values for a near-optimal policy that still explores. Whereas the **Off-Policy learning** learns about policy π from experiences sampled from b. So the **target policy** π is different from the **behavioral policy** b used to interact with the environment. In order to use episodes from b to learn π, the **coverage assumption** is required : $\pi(a|s) > 0 \Rightarrow b(a|s) > 0, \forall (s,a) \in \operatorname{gr}\Gamma$.

Usually, off-policy learning are more powerful and general because it can learn from non-learning controller or human expert. With the same behavioral policy b, multiple target policies π can be learned. Besides, the behavioral policy b may be more exploratory than the policy used in on-policy learning. It includes on-policy learning in the specific case where the target policy and behavioral policy are the same. Frequently, off-policy learning methods have higher variance and slower convergence than on-policy learning.

2.2 Generalized Policy Iteration

The purpose of this section is to explain the pattern of almost all model-free RL methods.

2.2.1 Policy Evaluation

To know if the current policy π is good, RL method seeks to determine the effectiveness of π. Such process is called **Policy Evaluation** [6]. Value functions can give some information about the efficiency of the policy π because by definition they represent how it is good for an agent following a policy π to be in a state $s \in \mathcal{S}$. Methods using value functions for the policy evaluation are named **value function methods** [6]. Value function methods work on deterministic policies.

2.2.2 Policy Improvement

After evaluating the effectiveness of the policy π, the next step of RL methods is to find a better policy π' from π. Before defining the **Policy Improvement** [6] step for value-function methods, it is important to introduce the **Policy Improvement Theorem**.

Theorem 2.2.1 (Policy Improvement Theorem [6])**.** Let π and π' be any pair of deterministic policies such that $\forall s \in \mathcal{S}$,

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s) \tag{2.9}$$

Then the policy π' must be as good as, or better than π

$$V_{\pi'}(s) \geq V_\pi(s) \tag{2.10}$$

If the condition (2.9) is a strict inequality then result (2.10) is a strict inequality too.

The proof of the Policy Improvement Theorem is given in the Sutton's book [6].

For value function methods, according to the Policy Improvement Theorem, it is possible to improve a policy π by acting **greedy** on its value function. The new greedy policy π' with respect to π is defined for any state $s \in \mathcal{S}$ as :

$$\pi'(s) = \arg\max_{a \in \Gamma(s)} Q_\pi(s, a)$$

The Policy Improvement Theorem is applied because:

$$\begin{aligned} Q_\pi(s, \pi'(s)) &= \max_{a \in \Gamma(s)} Q_\pi(s, a) = \sum_{a' \in \Gamma(s)} \pi(a'|s) \max_{a \in \Gamma(s)} Q_\pi(s, a) \\ &\geq \sum_{a' \in \Gamma(s)} \pi(a'|s) Q_\pi(s, a') = V_\pi(s) \end{aligned}$$

If $V_\pi = V_{\pi'}$ then the optimal policy is found. Note the sum can be replaced by an integral for infinite action space $\mathcal{A}$.

Therefore, the sequence $(\pi_k)_{k \in \mathbb{N}}$ defined by:

$$\pi_{k+1}(s) = \arg\max_{a \in \Gamma(s)} Q_\pi(s, a); \forall s \in \mathcal{S}$$

converges to the optimal policy π^*. See the Stachurski's Book [7] for a complete proof of the convergence of the sequence (π_k).

Nevertheless, there is problem by acting greedy and creating deterministic policy because there is no more **exploration**. Indeed, many state-action pairs in $\operatorname{gr}\Gamma$ may never be visited. If the policy is deterministic, returns would be observed only for one action from each state and estimates of other actions would not improve with experiences. This problem is very important, because to improve a policy with the best action selection, values of all actions from each state need to be known.
For maintaining **exploration**, one solution would be to use the assumption of **exploring starts** [6]. This method assumes that any state-action in $\operatorname{gr}\Gamma$ pair has a non-zero probability of being selected at the beginning of each episode. This assumption guarantees that every state-action pair will be visited in the limit of an infinite number of episodes. However, it can not always be used when learning from actual interactions.

Another popular way is to add directly exploration into the policy with a soft policy. A **soft policy** π is a policy which has a nonzero probability for every action that can be selected.

$$\pi(a|s) > 0, \forall (s,a) \in \operatorname{gr}\Gamma$$

An **ϵ-soft policy** π (with $\epsilon > 0$) is a policy where each action has at least a $\epsilon/|\Gamma(s)|$ probability of being selected.

$$\pi(a|s) \geq \frac{\epsilon}{|\Gamma(s)|}, \forall (s,a) \in \operatorname{gr}\Gamma$$

An **ϵ-greedy policy** π (with $\epsilon > 0$) is an ϵ-soft policy with a probability equal to $1-\epsilon+\epsilon/|\Gamma(s)|$ to select the greedy action and a probability of $\epsilon/|\Gamma(s)|$ to select any other actions. As well as for the greedy policy, the Policy Improvement Theorem for an ϵ-greedy policy can be applied to justify there is an improvement. A proof is given in the Sutton's book [6].

2.2.3 Generalized Policy Iteration

The term of **Generalized Policy Iteration** (GPI) [6] refers to the general idea of interactions between two processes :

- **Policy Prediction** : To estimate the effectiveness of a policy π.
- **Policy Improvement** : To improve the current policy π.

Each process modifies the basis for the other as it is illustrated by the Figure 2.3. The process is guaranteed being stable for optimal policies. A rigorous demonstration of this stability is given in the John Stachurski's book [7].

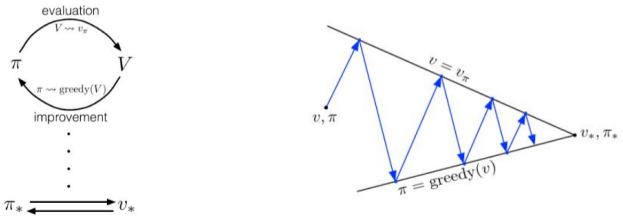


Figure 2.3: Illustration of the Generalized Policy Iteration for value-function methods (taken from [6])

2.3 On-Policy Control With Approximation

This section presents a class of on-policy **approximate value function methods** [6] for an infinite or large state space $\mathcal{S}$ and a finite action space $\mathcal{A}$.

As any value functions methods, value functions need to be estimated for the policy evaluation. Because the state space $\mathcal{S}$ may be infinite or quite large, value functions are approximated with parameterized functions $\widehat{v}(s, \mathbf{w}) \approx V_\pi(s)$ where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector. $\widehat{v}$ might be a linear function or computed with a multi-layers artificial neural network.
Usually, the number of weights ($\dim \mathbf{w}$) is much less than the number of states. Consequently, modifying one weight changes the estimated values of many states. Such **generalization** makes the learning potentially more powerful but also potentially more difficult to manage and understand. Because value functions are approximated with parametrized functions, most of Supervised Learning methods can be applied. However, the challenge is harder in RL because Supervised Learning methods assume a static training set over which multiple passes are made whereas in RL the learning is online while the agent interacts with its environment. Moreover, approximation methods need to handle non-stationary target functions because the policy changes.

2.3.1 The Prediction Objective $\overline{VE}$

With the realistic assumption that there are far more states than weights, making one state's estimate more accurately invariably means making others less accurate. The idea is to consider which states are the most important and having the best estimate for them. Let the probability measure $\mu \in \mathscr{P}(\mathcal{S})$ be a state distribution representing how much the error is important in a given state $s \in \mathcal{S}$. For example, μ could be the fraction of time spent in each state.

The **Mean Squared Value Error** $\overline{VE}$ [6] is defined as:

$$\overline{VE} = \sum_{s \in \mathcal{S}} \mu(s)(V_\pi(s) - \widehat{v}(s, \mathbf{w}))^2 \tag{2.11}$$

$$= \mathbb{E}_\pi \left[(V_\pi(s) - \widehat{v}(s, \mathbf{w}))^2 \right] \tag{2.12}$$

The aim is now to minimize the error $\overline{VE}$ to approximate $\widehat{v}(s, \mathbf{w}) \approx V_\pi(s), \forall s \in \mathcal{S}$ if the agent follows the policy π.

2.3.2 Stochastic-Gradient And Semi-Gradient Methods

At every time step t, one assumes observing an example $V_\pi(s_t)$ where $s_t \in \mathcal{S}$ is given by the state distribution μ defined in the section 2.3.1. The **Stochastic Gradient-Descent** (SGD) is a method approximating a function with a parameterized function. In RL, The SGD method adjusts the weight vector $\mathbf{w} \in \mathbb{R}^d$ after each example $V_\pi(s_t)$ by a small amount in the direction that would most reduce the error $\overline{VE}$ (section 2.3.1) on that example [6]. The update of the weight vector $\mathbf{w}$ is given by:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha_t \nabla (V_\pi(s_t) - \widehat{v}(s_t, \mathbf{w}_t))^2 \\ &= \mathbf{w}_t + \alpha_t (V_\pi(s_t) - \widehat{v}(s_t, \mathbf{w}_t)) \nabla \widehat{v}(s_t, \mathbf{w}_t)\end{aligned}$$

where α_t is the positive step-size parameter at time t, $\mathbf{w}_t$ the weight vector at time step t and $\nabla \widehat{v}(s_t, \mathbf{w}_t)$ is the derivative of $\widehat{v}(s_t, \mathbf{w}_t)$ with respect to the components of the vector $\mathbf{w}_t$. There is guarantee of convergence if :

$$\sum_{t \geq 1} \alpha_t = \infty \text{ and } \sum_{t \geq 1} \alpha_t^2 < \infty \tag{2.13}$$

In practice, a constant step-size parameter α_t is used because the convergence is faster. Besides, it is more suited for non-stationary rewards because it gives more weights to recent rewards. However, there is no guarantee of convergence to an optimal solution.

SGD is the iterative version of **gradient-descent**. In the SGD method, the target output $V_\pi(s_t)$ is considered to be known but it is not the case in practice. Let the target U_t be a noise-corrupted version of $V_\pi(s_t)$ ($\mathbb{E}[U_t | s_t = s] = V_\pi(s_t)$). For each time step t, the update of the weight vector $\mathbf{w}$ is :

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t (U_t - \widehat{v}(s_t, \mathbf{w}_t)) \nabla \widehat{v}(s_t, \mathbf{w}_t) \tag{2.14}$$

$\mathbf{w}_t$ is still guaranteed to converge to a local optimum if conditions (2.13) are respected [6]. Therefore, with the expected return G_t as target, SGD method converges to a locally optimal solution.

A **bootstrapping** [6] of $V_\pi(s_t)$ estimates $V_\pi(s_t)$ with estimates of the values of successor states. For example,

$$\begin{aligned}V_\pi(s_t) &= \mathbb{E}[G_t | s_t = s] = \mathbb{E}[R(s_t, a_t) + \gamma G_{t+1} | s_t = s] \\ &= \mathbb{E}[R(s_t, a_t) + \gamma V_\pi(s_t) | s_t = s] \\ &\approx \mathbb{E}[R(s_t, a_t) + \gamma \widehat{v}(s_t, \mathbf{w}_t) | s_t = s] = \mathbb{E}\left[\widehat{U}_t | s_t = s\right]\end{aligned}$$

However, convergence is no more guaranteed when bootstrapping is used. The target $\widehat{U}_t = R(s_t, a_t) + \gamma \widehat{v}(s_t, \mathbf{w}_t)$ depends on the current value of the weight vector $\mathbf{w}_t$ and it is therefore biased. It is not anymore a true gradient-descent method. Such methods using the update 2.14 with bootstrapping are called **semi-gradient methods** [6]. Even if the convergence is less robust than true gradient methods, semi-gradient methods converge reliably in some cases such as the linear case [6]. They are preferred because they are significantly faster, and they insure continual and online learning.

A linear approximation of the value function V_π is given by:

$$\widehat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s) \tag{2.15}$$

where $\mathbf{x}(s)$ is the **feature vector** representing the state $s \in \mathcal{S}$ (see section 2.4 for example of features encoding). Each component $x_i(s)$ of $\mathbf{x}(s)$ is a function $x_i : \mathcal{S} \to \mathbb{R}$.
The update (2.14) with a linear approximation is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t(U_t - \widehat{v}(s_t, \mathbf{w}_t))\mathbf{x}(s_t) \tag{2.16}$$

For semi-gradient methods under linear function approximation, the Sutton's book [6] provides a proof of convergence to a solution $\mathbf{w}_{TD}$ (TD point) close to the true solution. Therefore, such methods do not converge exactly to the optimal solution.

2.3.3 Semi-Gradient SARSA

With a model, state value functions are sufficient to determine a policy: one looks ahead one step and chooses whichever action leads to the best combination of reward and next state according to a greedy or ϵ-greedy strategy. Without a model, however, state value functions are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thereby, one prefer to work with action value functions.

The same strategies developed in the section 2.3.2 can be used for action value functions. For a policy π, the action value function Q_π is approximated with a parameterized function $\widehat{q}$ such that $\widehat{q}(s, a, \mathbf{w}_t) \approx Q_\pi(s, a), \forall (s, a) \in \operatorname{gr}\Gamma$ where $\mathbf{w} \in \mathbb{R}^d$ is the weight vector. The semi-gradient methods are the most commonly used because of their speed. To converge, the parameterized function $\widehat{q}$ is considered as a linear approximation Q_π. The update made at each time t for the weight vector $\mathbf{w}$ is thus:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t(U_t - \widehat{q}(s_t, a_t, \mathbf{w}_t))\nabla\widehat{q}(s_t, a_t, \mathbf{w}_t) \tag{2.17}$$

where $\mathbb{E}\left[U_t | s_t = s, a_t = a\right] = Q_\pi(s_t, a_t)$.

The **Semi-Gradient SARSA** [6] uses the update 2.17 with the target $U_t = R(s_t, a_t) + \gamma\widehat{q}(s_t, a_t, \mathbf{w}_t)$. The Pseudo-Code of this method is given by Algorithm 1. In the same way, all on-policy tabular methods based on SARSA such as **Expected SARSA** [6], **n-step SARSA** [6], SARSA(λ)... have a corresponding semi-gradient version. They all have the same update rule 2.17 but with their respective targets.

For continuous tasks, **differential returns** are used. For further details see the Sutton's book [6].

Algorithm 1 Semi-Gradient SARSA $\pi \approx \pi^*$ (taken from [6]))

Require: step-size $\alpha \in [0, 1]$, small $\epsilon > 0$
Require: a differentiable action-value function parameterization $\widehat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Initialize :
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily ($\mathbf{w} = 0$)
loop
Initialize s
for each step of the episode **do**
Choose $a \in \Gamma(s)$ thanks to $\widehat{q}(s, ., \mathbf{w})$ (e.g. ϵ-greedy strategy)
Observe $R(s, a), s'$
if s' is terminal **then**
$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R(s, a) - \widehat{q}(s, a, \mathbf{w}))\nabla\widehat{q}(s, a, \mathbf{w})$
Go to the next episode
end if
Choose $a' \in \Gamma(s')$ thanks to $\widehat{q}(s', ., \mathbf{w})$ (e.g. ϵ-greedy strategy)
$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R(s, a) + \gamma\widehat{q}(s', a', \mathbf{w} - \widehat{q}(s, a, \mathbf{w}))\nabla\widehat{q}(s, a, \mathbf{w})$
$s \leftarrow s'$
$a \leftarrow a'$
end for
end loop

2.4 Features Encoding

This section introduces different **feature constructions** for $\mathbf{x}(s)$ where $s \in \mathcal{S}$. Features add prior domain knowledge to RL systems and they should correspond to aspects of the state space along which generalization may be appropriate. The case where the state space $\mathcal{S}$ is not encoded corresponds to the linear case. Of course, a linear choice may be chosen but it cannot take into account any interactions between features. For the Cart-Pole task, a high angular velocity can be either good or bad depending on the angle. If the angle is high then a high angular velocity may be dangerous.

2.4.1 Tile Coding

Tile Coding [6] is one of the most practical feature representation for multi-dimensional infinite state space in RL problems. In Tile Coding, the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a **tiling**, and each element of the partition is called a **tile**. Each tile is a receptive field for one binary feature. Every state $s \in \mathcal{S}$ is exactly in one tile of each tiling, the number of features encoding a state s corresponds thus to the number of tilings used. Figure 2.4 illustrates an example of Tile Coding on a two-dimensional state space $\mathcal{S}$.

Tilings are offset from each other by a fraction of the tile width w_i for each dimension $i \in [1, n]$. If there are m tilings then the quantity $\frac{w_i}{m}$ is the unit distance for the offset in the dimension i. **Uniformly offset** tiling are offset from each other by exactly the unit distance $\frac{w_i}{m}$ for every dimension $i \in [1, n]$. We say that each tiling is offset by the **displacement vector**

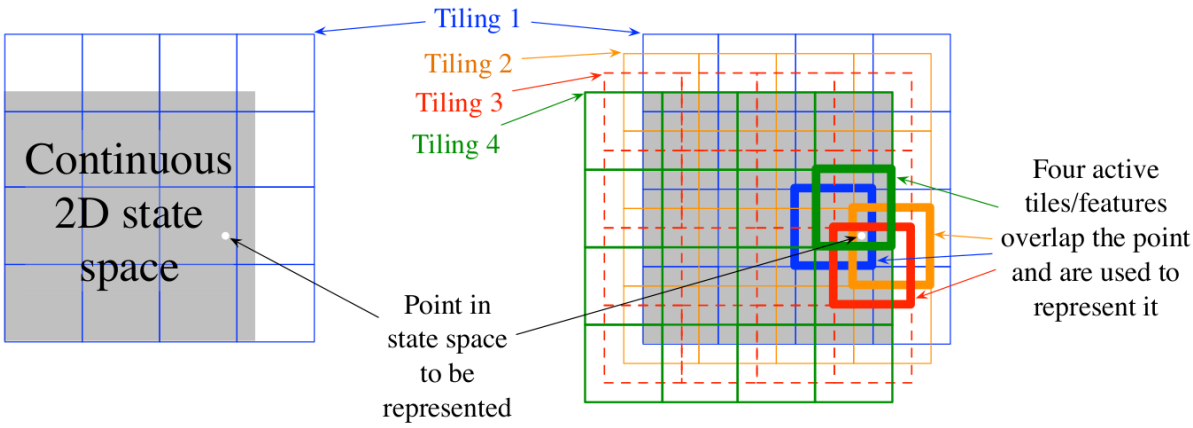


Figure 2.4: Multiple, overlapping grid-tilings on a limited two-dimensional space. Tilings are offset from one another by a uniform amount in each dimension (taken from [6])

$\mathbf{u} = (1, \ldots, 1) \in \mathbb{R}^n$. If the displacement vector $\mathbf{u}$ is not collinear to the vector $(1, \ldots, 1) \in \mathbb{R}^n$, the offest is said to be **asymmetric**.

Offsets between each tiling impacts the generalization as described by Figure 2.5. In Figure 2.5, eight tilings are used. Each of the eight subfigures shows the pattern of generalization from a state to its neighbors. Artefacts on diagonal are observed in many patterns with uniform offsets whereas the generalization is better for the asymmetric offest with the displacement vector $\mathbf{u} = (1, 3)$. Miller and Glanz recommended using displacement vectors consisting with first odd integers. In particular for infinite state space $\mathcal{S} \subseteq \mathbb{R}^n$, a good choice is to use the first odd integers $(1, 3, 5, \cdots, 2n-1)$ with m the number of tilings set to an integer power of 2 or greater than $4n$. For a good convergence, Sutton [6] suggests using the step-size parameter $\alpha = \frac{1}{10m}$.

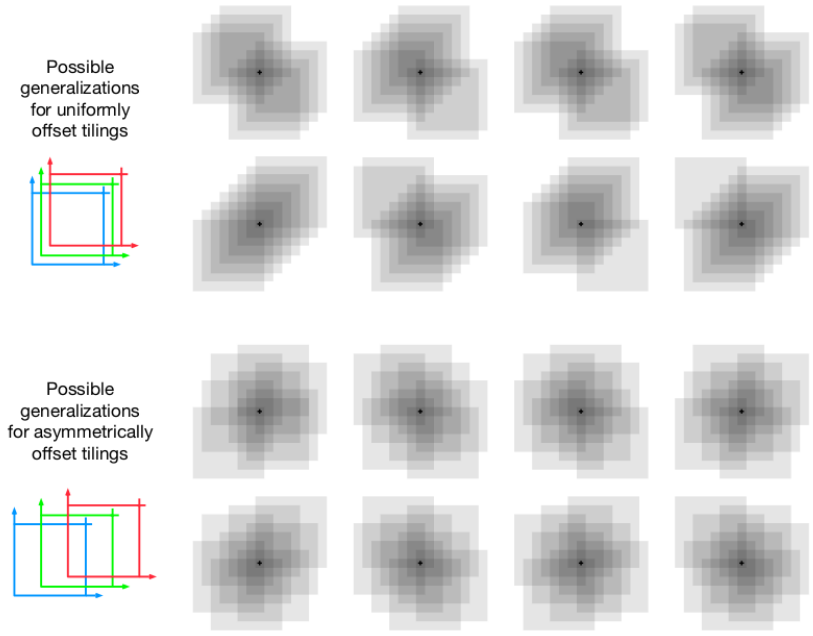


Figure 2.5: Strength of generalization for uniformly and asymmetrically offset tilings (taken from [6])

A Tile Coding with one tiling is named a **State Aggregation**. State Aggregation is a sort of discretization of the state space $\mathcal{S}$ because $\mathcal{S}$ is represented by a single feature whose tile it falls within.

2.4.2 Fourier Basis

Any τ- periodic function f (f is τ-periodic if $f(x+\tau) = f(x)$) can be described as a weighted sum of sine and cosine basis functions of different frequencies : the **Fourier Series**. The function f can be approximated by truncating the Fourier Series.

For a τ-periodic function f, the m^{th} degree Fourier expansion $\overline{f}$ is:

$$\overline{f}(x) = \frac{a_0}{2} + \sum_{k=1}^{m} (a_k \cos(k\frac{2\pi}{\tau}x) + b_k \sin(k\frac{2\pi}{\tau}x)) \tag{2.18}$$

where $a_k = \frac{2}{\tau}\int_0^\tau f(x)\cos(\frac{2\pi kx}{\tau})\mathrm{d}x$ and $b_k = \frac{2}{\tau}\int_0^\tau f(x)\sin(\frac{2\pi kx}{\tau})\mathrm{d}x$.

As well as for periodic function, aperiodic functions defined over a bounded interval can be approximated with Fourier Series [5]. Indeed, if a bounded aperiodic functions is considered as a τ-periodic function where τ is equal to the length of interval then the function of interest is just one period of the periodic linear combination of sine and cosine features.

An even and odd function is just a linear combination of sine and cosine features respectively. For aperiodic function bounded on an interval I, if τ is set to twice the length of the interval of interest I and attention is restricted only to the approximation over the half interval $[0, \tau/2]$, just the cosine features or sine features can be used.
In general, it is better to use the "half-even" approximation and drop the sine terms because this causes only a slight discontinuity at the origin (Figure 2.6).

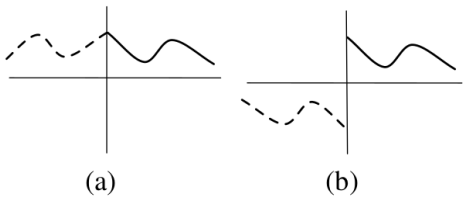


Figure 2.6: Even and odd functions (taken from [5])

The **univariate m^{th} order Fourier basis** [5] corresponds to the $m+1$ features defined as:

$$x_i(s) = \cos(i\pi s) \in [-1, 1] \tag{2.19}$$

for all $i \in [0, m]$ where the input variable s is defined over the the interval $I = [0, 1]$. Figure 2.7 depicts a few of the resulting basis functions.

A similar basis can be introduced for the multi-variate case. If for each state $\mathbf{s} = (s_1, s_2, \cdots, s_n)^\top \in$

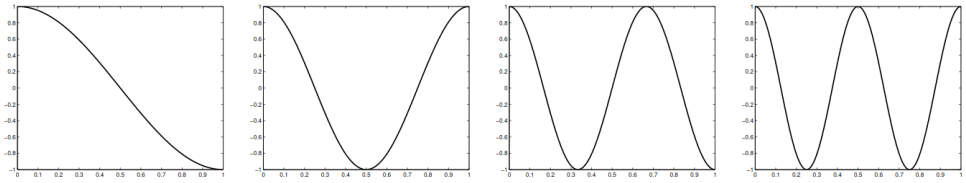


Figure 2.7: One dimensional Fourier cosine-basis features x_i for $i = 1, 2, 3, 4$ (taken from [5])

$\mathcal{S}$, such that $s_i \in [0, 1], \forall i \in [0, 1]$ then the i^{th} feature for the **order-m Fourier cosine basis** [5] is :

$$x_i(s) = \cos(\pi \mathbf{s}^\top \mathbf{c}^i) \tag{2.20}$$

where $\mathbf{c}^i = (c_1^i, c_2^i, \cdots, c_k^i)$ for $c_j^i \in \{0, \cdots, m\}$ for $j = \{1, \cdots, n\}$ and $i = \{0, \cdots, (m+1)^k\}$. The $\mathbf{s}^\top \mathbf{c}^i$ value is an integer determining the feature's frequency along that dimension. Example of basis functions over a two-dimensional state space $\mathcal{S}$ are shown in Figure 2.8. Therefore, when

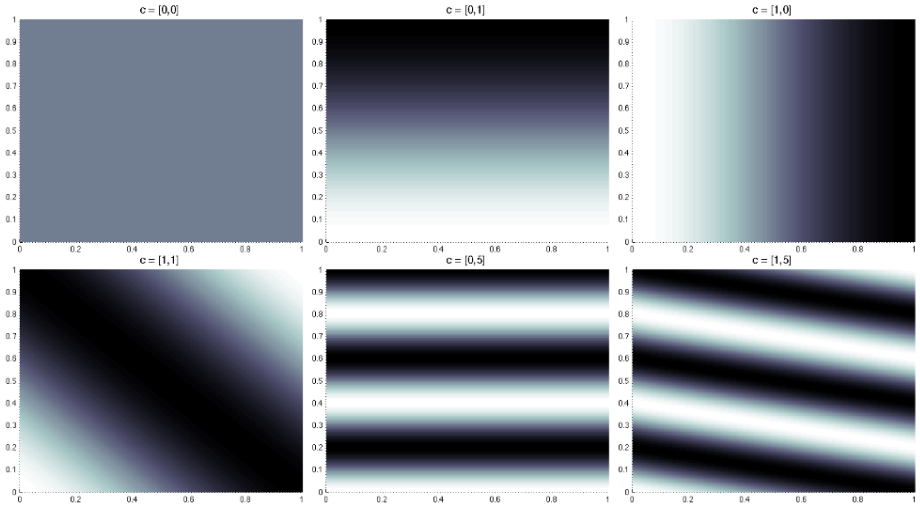


Figure 2.8: A selection of 6 two-dimensional Fourier cosine features, each labeled by the vector $\mathbf{c}^i$ that defines it (taken from [5])
$c = [0, 0]$ results in a constant feature along both dimensions. When $c = [0, k_y]$ or $c = [k_x, 0]$ for positive integers k_x and k_y, the feature varies along the corresponding non-zero component. k_x and k_y determine the frequency. Only when $c = [k_x, k_y]$ does it depend on both; this basis function represents an interaction between both dimensions. The ratio between k_x and k_y describes the direction of the interaction, while their values determine the basis function's frequency along each dimension.

the Fourier Basis is used in RL problems, the methods learn the coefficients of the m^{th} degree Fourier expansion of the function that they want to approximate (the action value function for example).

Chapter 3

Tile Coding Versus Fourier Basis

The objective of this chapter is to study the influence of Tile Coding and Fourier Based (introduced in the previous chapter) on classical Reinforcement Learning methods. Experiments have been made on the Cart-Pole problem. All methods implemented in this chapter were implemented from scratch.

3.1 Introduction To The Cart-Pole Environment

A classic **Cart-Pole** environment was used for experiments based on OpenAI gym Cart-Pole-Environment (`https://gym.openai.com/`). This environment corresponds to the version of the Cart-Pole problem described by Barton, Sutton, and Anderson [6].
The objective in the Cart-Pole - known also as an Inverted Pendulum - problem is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over. There are two cases of failure: either the pole falls past a given angle from vertical or the cart runs over the track. Besides, Gym considers the Cart-Pole has an episodic task having a maximum number of steps. Therefore, the conditions of terminations of an episode are when:

- the pole angle is **more than** 12 **degrees**
- the cart position is **more than** 2.4
- the episode length **is greater than a predefined number steps** (usually 200 or 500 steps)

In Gym, the state space $\mathcal{S}$ of the Cart-Pole task is a subset of $\mathbb{R}^4$. A state $s = (s_1, s_2, s_3, s_4)^\top \in \mathcal{S}$ is a vector where $s_1 \in [-4.8, 4.8]$ is the cart's **position**, $s_2 \in \mathbb{R}$ is the cart's **velocity**, $s_3 \in [-24°, 24°]$ is the pole's **angle**, and $s_4 \in \mathbb{R}$ is the pole's **angular velocity**.
At the beginning of the task, a state $s_0 = (s_{0,1}, s_{0,2}, s_{0,3}, s_{0,4})^\top$ is randomly chosen where all state variable $s_{0,i}$ are assigned into a uniform random value in $[-0.05, 0.05]$. The position 0 corresponds to the middle of the ray.

At any state $s \in \mathcal{S}$, the cart has only two possible actions: either **move to the left** (force of -1) or **move to the right** (force of $+1$). Consequently, the action space $\mathcal{A}$ is **discrete** and

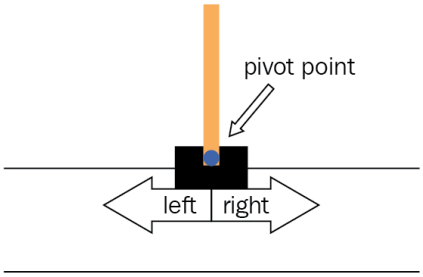


Figure 3.1: Cart-Pole used by OpenAI Gym

for every state $s \in \mathcal{S}$ we have $\Gamma(s) = \{-1, 1\}$.

A reward of $+1$ is provided for every step of the episode implying $R(s, a) = 1, \forall (s, a) \in \operatorname{gr} \Gamma$.

The Figure 3.1 depicts the Cart-Pole used in Gym.

Note that the cart moves along a **frictionless track** and **no noise is added**.

Firsts tests were made on the Cart-Pole task with classic tabular RL methods (SARSA, Q-Learning, SARSA(λ), ...) introduced in the Sutton's book [6] after discretization of the state space $\mathcal{S}$ but the performance was very poor. Therefore, even if it is known that approximation methods do not converge to the optimal policy, they remain more efficient than tabular methods for large state spaces or infinite state spaces. All methods in this section take as input the four observations returned by the environment.

In order to use the Features Encoding defined in the previous chapter, it is necessary to apply a preprocessing f on the the speed and the angular velocity before using the features encoding methods because they are not bounded.
Therefore, for every state $s = (s_1, s_2, s_3, s_4) \in \mathcal{S}$, the preprocessing $f : \mathbb{R} \to [-3, 3]$ is applied on the state variables s_2 and s_4 where:

$$f(x) = \begin{cases} x & \text{if } x \in [-3, 3] \\ -3 & \text{if } x < -3 \\ 3 & \text{if } x > 3 \end{cases}$$

In practice the speed and the angular velocity are almost always in $[-3, 3]$.

3.2 Tile Coding

This subsection describes the use of the Tile Coding as features encoding for the Cart-Pole task. According to the section 2.4.1, the optimal number of tilings is 16 (because $\dim \mathbb{R}^4 = 4$ and $2^4 = 16$) and the displacement vector for each tiling is collinear to $[1, 3, 5, 7]^\top$.
Likewise, the literature suggests the step-size parameter:

$$\alpha = \frac{1}{10 \times 16} = \frac{1}{160} \approx 0.01666666666 \tag{3.1}$$

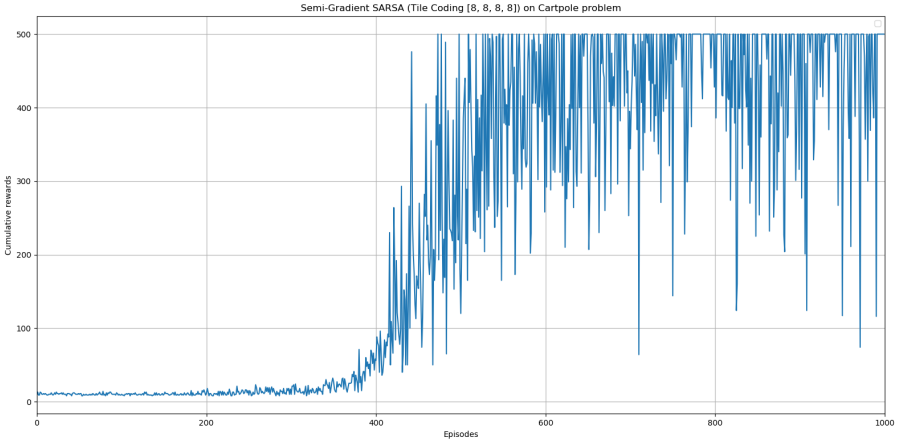
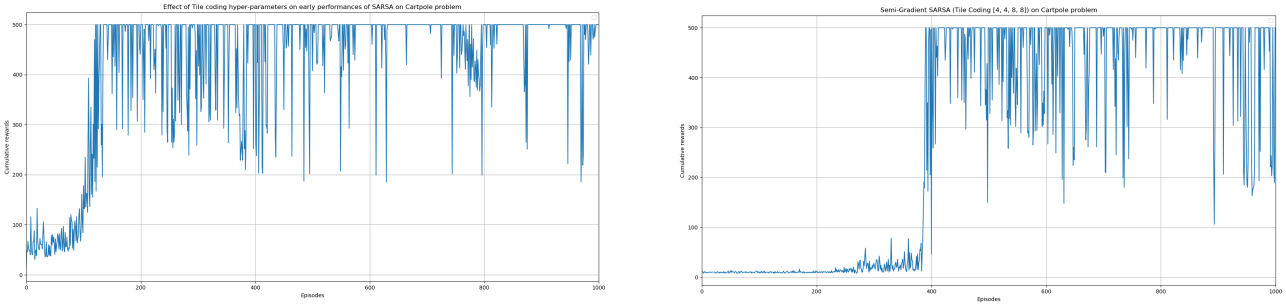


Figure 3.2: Training performance of Semi-Gradient SARSA combined with Tile-Coding using 8 tiles per dimension



(a) Training 1

(b) Training 2

Figure 3.3: Difference between two trainings for Semi-Gradient SARSA combined with Tile-Coding $[4, 4, 8, 8]$

A first test with the same tile width for all dimensions of the state space $\mathcal{S}$ was made but poor performance was observed suggesting the use of a different tile width per dimensions.

A Grid-Search on the Semi-Gradient SARSA method was made to find the best number of tiles dividing each dimension of the state space $\mathcal{S}$. Figure 3.2 depicts the performance of the Semi-Gradient SARSA combined with Tile Coding using the result found by the Grid-Search (8 tiles per dimension) for episodes with a maximum length of 500 steps. It is clear the method learns but it is highly unstable after convergence.

The instability can be reduced by considering a different number of tiles for each dimension. In other words, it seems that some dimensions need a better precision than others. A new Grid-Search on the Semi-Gradient SARSA method was made to find the best number of tiles per dimension. $[4, 4, 8, 8]$ was the best number of tiles dividing the cart's position, the cart's speed, the pole's angle, and the pole's angular velocity respectively. More attention is given to the angle than the position. Figure 3.3 shows performance of Semi-Gradient SARSA combined with Tile-Coding using the $[4, 4, 8, 8]$ configuration for episodes with a maximum length of 500 steps. The training is very different from one simulation to another. Indeed, in the Figure 3.3a, there are few instabilities and a fast convergence (achieved before the 200^{th} episode) whereas

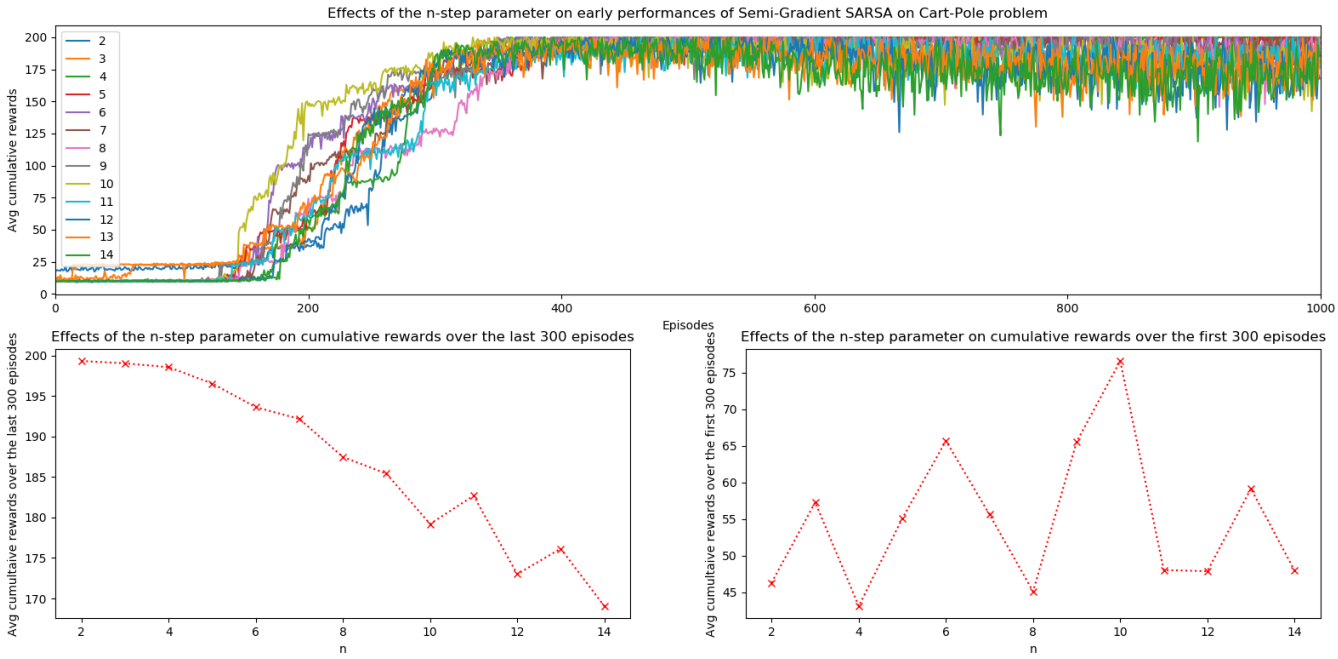


Figure 3.4: Training performance of Semi-Gradient SARSA n-step combined with Tile-Coding $[8, 8, 8, 8]$ (over 20 trainings) for various values of n

in in the Figure 3.3b the opposite is true : many instabilities and a slow convergence (achieved at the 420^{th} episode). Therefore, it very important to consider a large number of simulations to know the general behavior. It is clear that despite a high variance between simulations, dividing each dimension with a different number of tiles is better. There are less and less instabilities as the number of episodes increases suggesting that the RL methods try to reduce these instabilities and still learn.

Despite better results, it is not enough for a task as easy as the Cart-Pole task. Other methods need to be considered such as the **Semi-Gradient n-step SARSA** [6]. It is the same semi-gradient update rule used by Semi-Gradient SARSA except with the target:

$$\widehat{U}_t = R(s_{t+1}, a_{t+1}) + \gamma R(s_{t+2}, a_{t+2}) + \gamma^2 R(s_{t+3}, a_{t+3}) + \cdots + \gamma^{n-1} R(s_{t+n}, a_{t+n}) + \gamma^n \widehat{q}(s_{t+n}, s_{t+n}, \mathbf{w}_{t+n})$$

For many tasks, there exists $n > 1$ where Semi-Gradient n-step SARSA has better performance than Semi-Gradient SARSA. Figure 3.4 resumes performance of Semi-Gradient SARSA n-step combined with the Tile-Coding $[4, 4, 8, 8]$ configuration over 20 training for various values of n. Surprisingly, it seems that using a n-step parameter greater than one do not improve performances suggesting that the Semi-Gradient SARSA is already the best Semi-Gradient SARSA method. Despite the convergence speed is faster for Semi Gradient SARSA n-steps methods, performance is degraded over the long term as it is shown with $n = 10$. It is clear that increasing n reduces the performance over the long term but it is not clear that increasing n reduce or improve performance on early episodes.

With a lot of instabilities and a slow convergence, Tile Coding is not the best feature encoding for the Cart-Pole task. These poor results can be explained because the number of features is huge. For example, for the best Tile Coding [4, 4, 8, 8] configuration there are 31399 features and thus 31399 weights to learn ! The more weights there are, the longer it takes to learn.

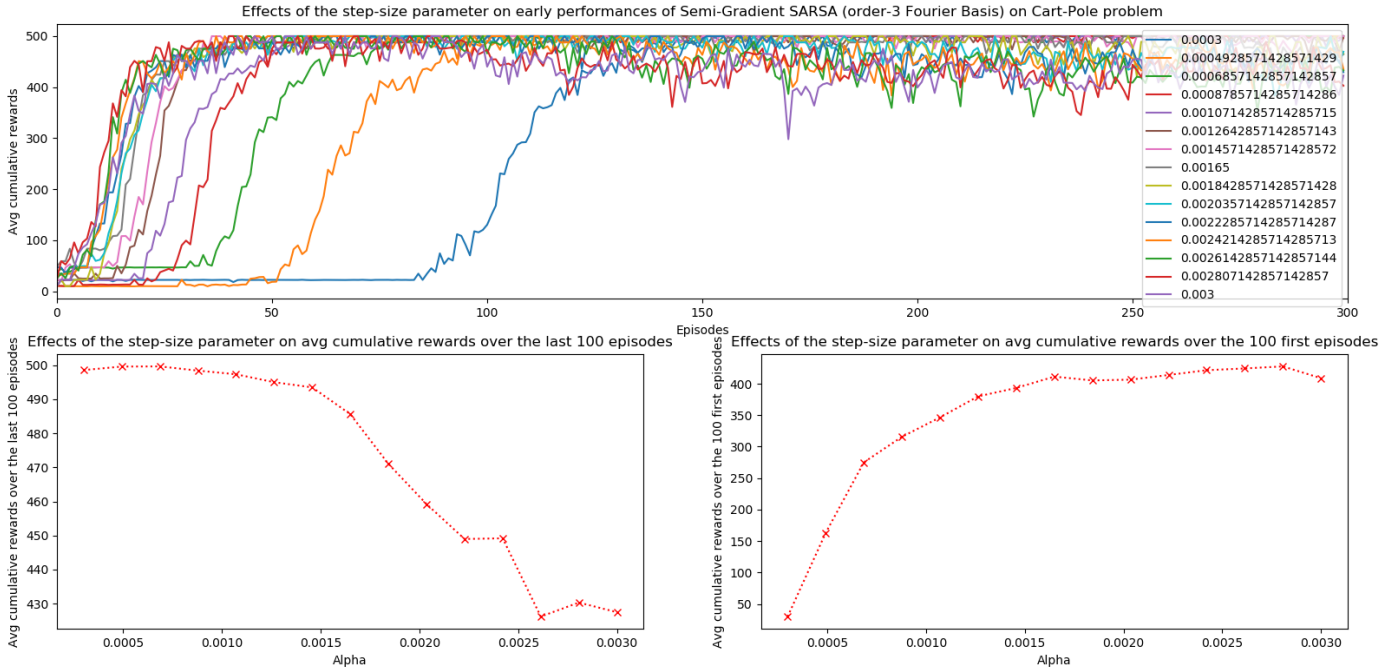


Figure 3.5: Training performance of Semi-Gradient SARSA combined with order-3 Fourier Basis (over 40 trainings) for various values of the step-size parameter α

3.3 Fourier Basis

This subsection describes the use of the Fourier Basis (subsection 2.4.2), in particular the order-3 and order-5 Fourier Basis, as features encoding for the Cart-Pole task. Only episodes with a maximum length of 500 steps are considered. The first advantage of the Fourier Basis over the Tile Coding is the number of features with only 256 features and 1296 features for order-3 and order-5 Fourier Basis, respectively. A lower number of features means a lower number of features and thus a higher speed of convergence and a lower computation time.

Even if Konidaris, Osentoski, and Thomas (2011) suggest a different step-size parameter for each feature, the literature does not suggest any trivial constant step-size α for every feature from what I know so far. A Grid Search over the step-size parameters α was thus necessary. The Figure 3.5 resumes results of this research on Semi-Gradient SARSA using order-3 Fourier Basis. Semi-Gradient SARSA is much more efficient and faster with order-3 Fourier Basis than with Tile Coding. In the worst case, for $\alpha = 0.003$, the convergence is achieved at the 200^{th} episodes (versus at the 400^{th} episodes for Tile Coding).

Average cumulative rewards over the last 100 episodes are indicators of the performance of a method after training. Therefore, Semi-Gradient SARSA with low α seem to get better results. The maximum average cumulative rewards over the last 100 episodes is achieved for $\alpha = 0.0007$. It should be noted that by lowering the step-size parameter α too much, performance begins to deteriorate. On the contrary for high α, the average cumulative rewards over the 100 first episodes is bigger than for low α. Similarly to Tile Coding, methods with a fast convergence are not the more efficient. These results are consistent with the literature, choosing a high step size parameter implies a fast convergence but a convergence to a suboptimal solution. It is a known problem for gradient descent methods. Similarly, a low α implies a

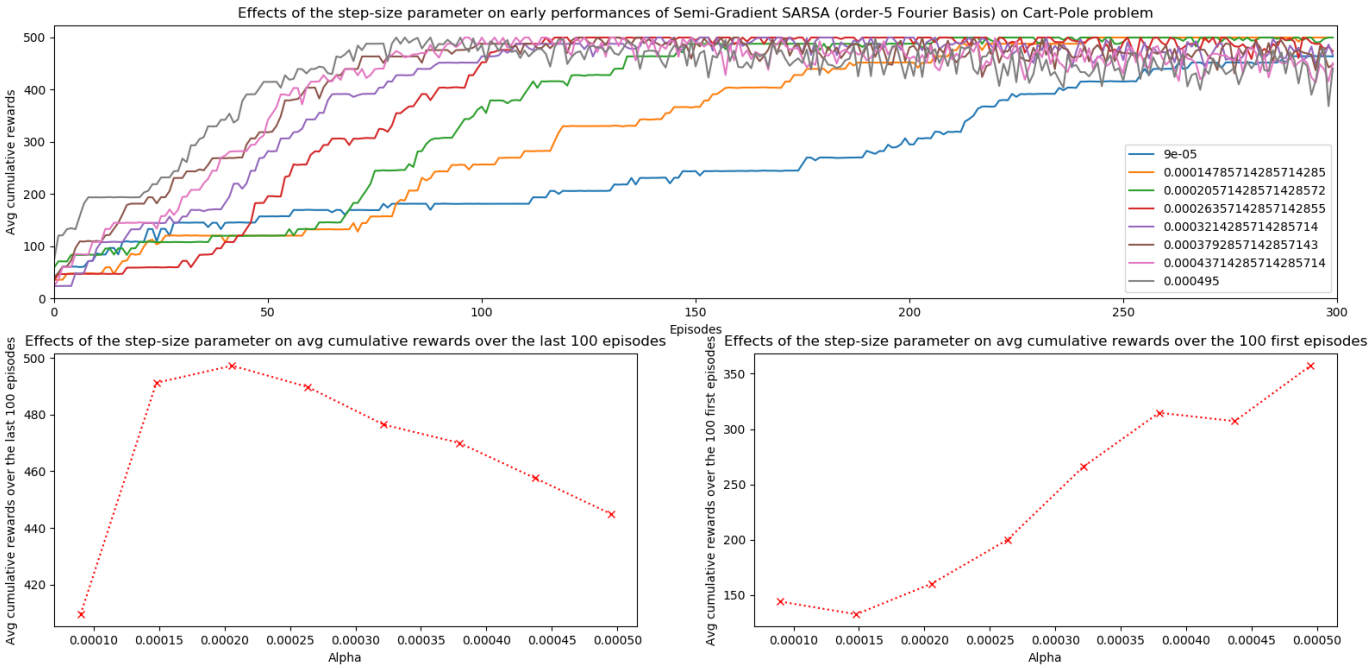


Figure 3.6: Training performance of Semi-Gradient SARSA combined with a order-5 Fourier Basis (over 40 trainings) for various values of the step-size parameter α

slower convergence but a stronger guarantee to converge to the optimal solution. Thereby, the choice of α appears as a trade-off between speed and efficiency.

Comparing the results for $m = 3$ with those for $m = 5$, Figure 3.6 shows us that the order of the Fourier Basis has an influence on the step size parameter α. Indeed, the higher is the order of the Fourier Basis, smaller the step-size parameter α is. The best step-size parameter α found for the order 5 Fourier Basis is $\alpha = 0.00025$. Besides, in any case, the convergence is slower with order 5 than with order 3. It is explained by a greater number of features with order 5 implying more weights to learn.

It can be interesting, to check the impact of another method like the **Semi-Gradient Expected SARSA** [6] on the performance. Semi-Gradient Expected SARSA is a method close to the Semi-Gradient SARSA with the target:

$$\widehat{U}_t = R(s_{t+1}, a_{t+1}) + \gamma \sum_{a \in \Gamma(s)} \pi(a|s_{t+1}) \widehat{q}(s_{t+1}, a, \mathbf{w_t})$$

Figure 3.7 depicts the results of a Grid-Search on the Semi-Gradient Expected SARSA. Training performance are similar to those with Semi-Gradient SARSA. Moreover, the order of magnitude of best step-size parameters α is identical. Furthermore, Expected SARSA is not significantly better or worse than Semi-Gradient SARSA.

The study has been extended to other derivatives of Semi-Gradient SARSA methods as the Semi-Gradient SARSA(λ). The **Semi-Gradient SARSA(λ)** is an improvement of the Semi-Gradient n-step SARSA where the target $\widehat{U}_t$ is a weighted average of the target of the Semi-Gradient SARSA n-step for different values n. For a trace-decay λ close to 0 the target takes more into account the target of Semi-Gradient n-step for small n, whereas it is the opposite for

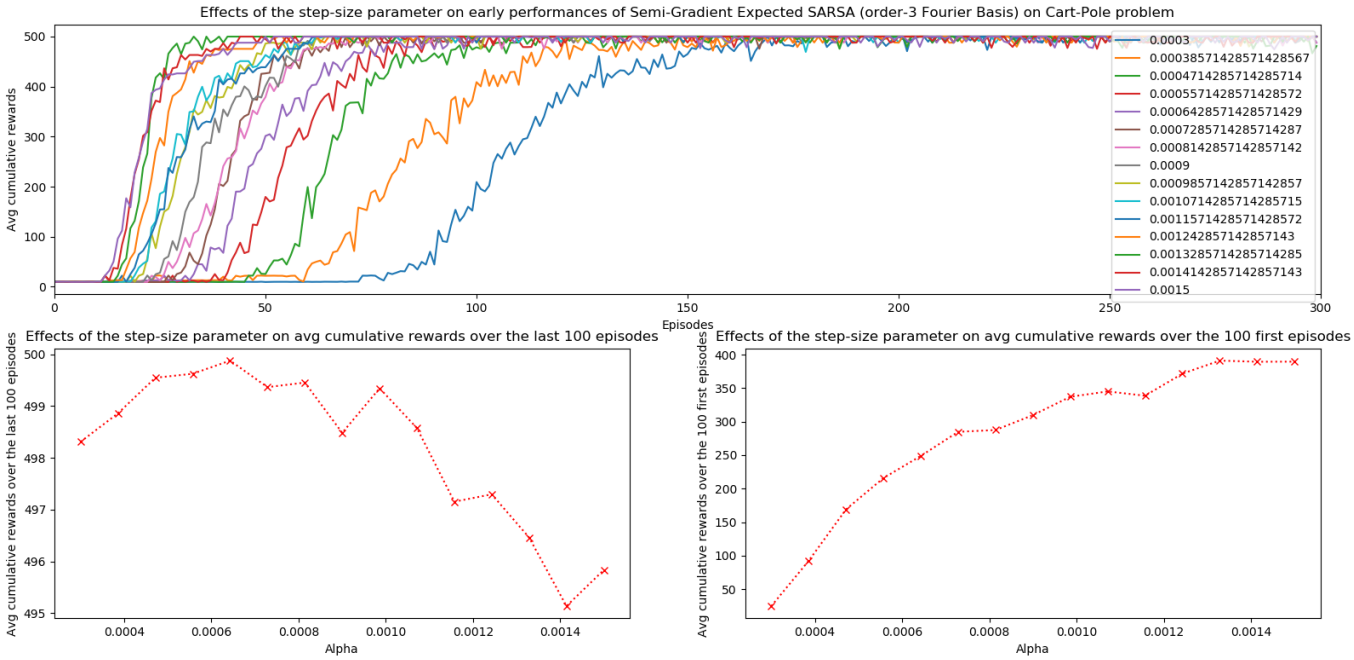


Figure 3.7: Training perfomance of Semi-Gradient Expected SARSA with a order-3 Fourier Basis (over 40 trainings) for various values of the step-size parameter α

λ close to 1.
Figure 3.8 shows performance for the **Semi-Gradient SARSA(λ)** [6] combined with a order-3 Fourier Basis for different values of λ and α values. All curves have the same behavior suggesting that the step-size parameter α has the same impact on all Semi-Gradient SARSA methods and that all Semi-Gradient methods have the same performance. Moreover, all Semi-Gradient SARSA(λ) share the same optimal parameter $\alpha = 0.0005$. Increasing the parameter λ seems to increase the α sensitivity of Semi-Gradient SARSA(λ). In other words higher is λ, smaller is α. $\lambda = 0.3$ is more robust to the variations of the step-size parameter α than other derivatives.

Figure 3.9 is a comparative of all Approximate Solution Methods implemented with the best hyper-parameters found. It is clear that increasing the Fourier order decreases the speed of the learning. Besides after convergence, there are more oscillations than with methods combined with order-3. Semi-Gradient Tree-Backup(λ) performs better than Semi-Gradient SARSA for the the first episodes but is caught up before convergence. Finally, Semi-Gradient SARSA combined with order-3 Fourier Basis is the best method. Surprisingly, derivatives using n-step do not perform better for both features encoding.

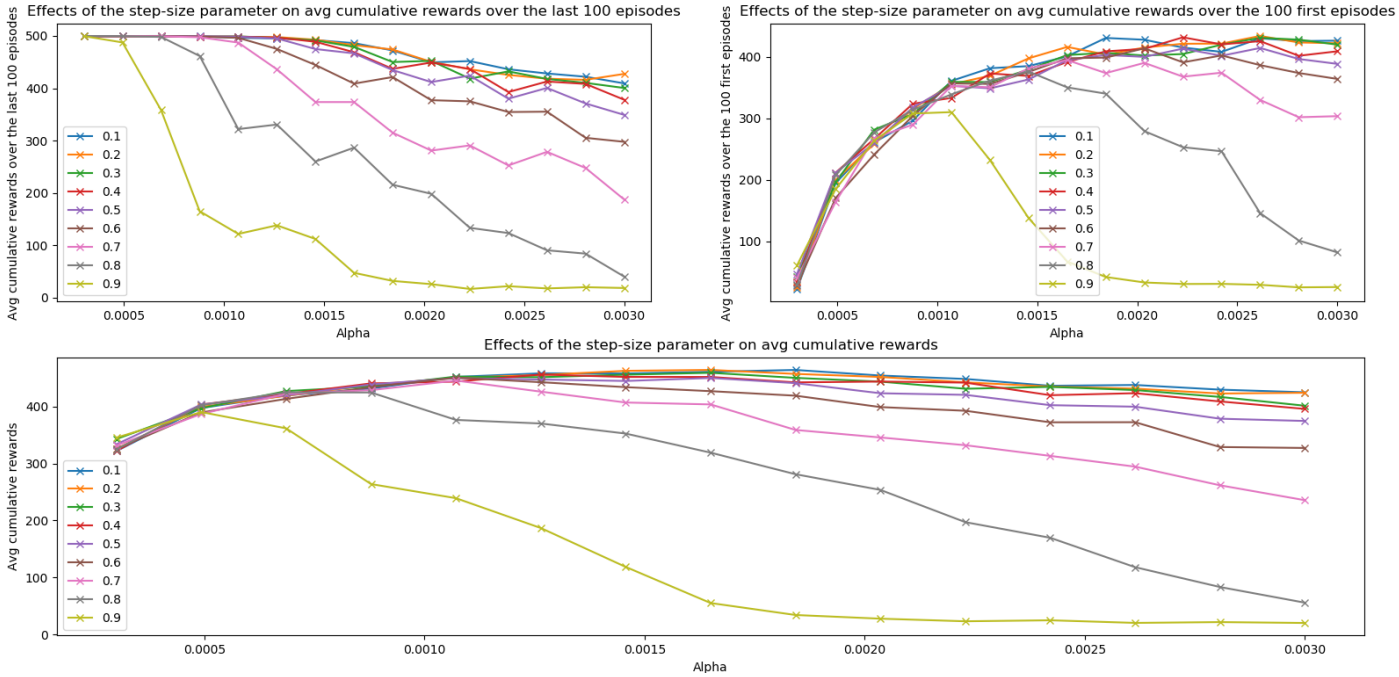


Figure 3.8: Training performance of Semi-Gradient SARSA(λ) combined with a order-3 Fourier Basis (over 20 trainings) for various values of the step-size parameter α

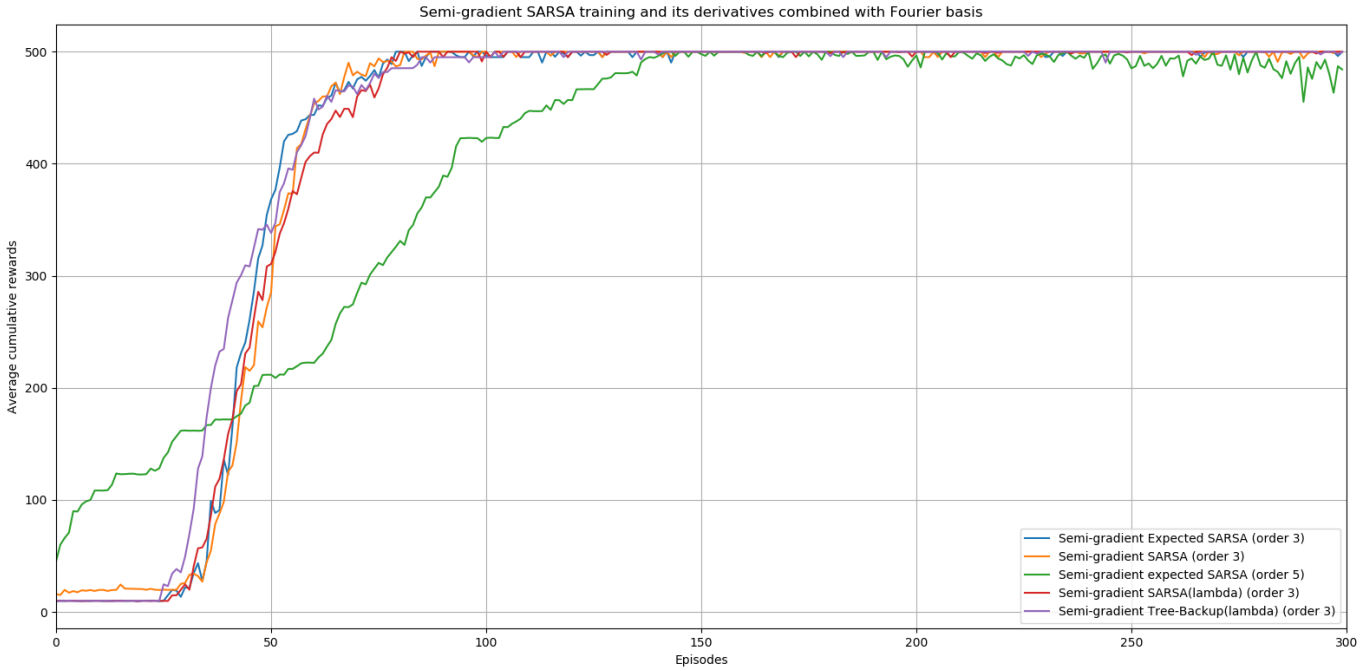


Figure 3.9: Comparative of the training performance of Semi-Gradient SARSA and its derivatives combined with Fourier Basis (over 100 trainings)

Chapter 4

Introduction To Deep Reinforcement Learning

Since 2010, great progress has been made in the use of neural networks in Machine Learning to such an extent that a new family of Machine Learning methods based on neural networks called **Deep Learning** has emerged. However, the use of neural networks in Reinforcement Learning is more recent and challenging. Indeed in RL, contrary to the Supervised Learning and the Unsupervised Learning, there is no training set with independent and identically distributed (i.i.d) samples. The use of Deep Learning methods in Reinforcement Learning is named **Deep Reinforcement Learning**.
The purpose of this chapter is to propose a state of the art of the best known Deep RL methods.

4.1 Deep Q-Network (DQN)

Deep Q-Learning (DQN) [8] was introduced for the first time in 2014 to solve Atari Games. As an action-value method, DQN learns a parameterized estimation $\widehat{q}(s, a, \mathbf{w})$ of the real action-value function $Q_\pi(s, a)$ using a deep neural network and selects actions according to a ϵ-greedy strategy with respects to $\widehat{q}$. This method only treats tasks with a finite action space $\mathcal{A}$ where $\Gamma(s) = \mathcal{A}, \forall s \in \mathcal{S}$. The neural network takes states as inputs and and has as outputs the estimates of the action-value function. The last layer of the DQN architecture is a fully-connected layer. The parameters of the network are trained with a gradient descent to minimize some suitable loss function. Indeed, the gradient of the loss is back-propagated into the parameters $\mathbf{w}$. The same error function defined on On-Policy Control approximation, $\overline{VE}$ (in section 2.3.1), is used. The state-value function is replaced by the action-value function and the target is replaced by the Q-Learning target $\widehat{U}_t = R(s_t, a_t) + \gamma \max_{a \in \Gamma(s)} \widehat{q}(s_{t+1}, a, \mathbf{w}_t)$ [6, 9]:

$$\mathcal{L}(\mathbf{w}_t) = \mathbb{E}_\pi \left[\left(R(s_t, a_t) + \gamma \max_{a \in \Gamma(s)} \widehat{q}(s_{t+1}, a, \mathbf{w}_t) - \widehat{q}(s_t, a_t, \mathbf{w}_t) \right)^2 \right] \tag{4.1}$$

Nevertheless, the use of neural networks in RL is not so easy, two issue need to be addressed before the neural network converges towards a solution:

- In the same way the use of bootstrapping was problematic for On-Policy Approximation Value Methods (section 2.3.2), it is an issue for DQN. Contrary to Supervised Learning problems, the target $\widehat{U}_t = R(s_t, a_t) + \gamma \max_{a \in \Gamma(s)} \widehat{q}(s_{t+1}, a, \mathbf{w}_t)$ is not stable and is changing with $\mathbf{w}_t$. Since, the same parameters $\mathbf{w}_t$ are used to estimate targets and predictions, it means that at every step of the training target value shifts as much as estimate $\widehat{q}$ values. It is like chasing a moving target or chasing its own tail.

  A solution is to introduce **periods of supervised learning** [8]. The idea is to create two deep neural networks with parameters $\mathbf{w}^-$ and $\mathbf{w}$. The network with parameters $\mathbf{w}^-$ called the **target network** is used to retrieve Q_π estimates while the **online network** with parameters $\mathbf{w}$ includes all updates in the training. After a predefined number of iterations K, $\mathbf{w}^-$ is updated to $\mathbf{w}$. The purpose of the target network is to fix the Q_π estimates temporarily for the target.

- In most Learning Methods, samples are assumed **independently and identically distributed** (i.i.d). Obviously, it is not the case in Reinforcement Learning because there is a correlation between samples. Indeed, each sample $(s_t, a_t, R(s_t, a_t), s_{t+1})$ depends on what was made in the past. Recall that actions that are taken by the agent affect the future path of the state variable. Another problem is that DQN tends to forget the previous experiences as it overwrites with new experiences.

  Putting samples into a buffer called **Replay Buffer** (Lin, 1993), and applying updates on batch of experiences randomly chosen in the Replay Buffer addresses these problems. It makes the data distribution more stationary, independent of each other and closer to the i.i.d. ideal.

The combination of these both concepts (Replay Buffer and Target Network) make the use of neural networks possible. An example of Pseudo-Code for the DQN is given by the Algorithm 2.

For a better and robust convergence, authors [8] advice the use of the RMSprop optimization algorithm to update weights and the Huber Loss which is more robust to outliers.

Appendix B.1 introduces some DQN extensions used in this internship to improve the DQN performance.

4.2 Advantage Actor-Critic Methods

This section introduces a new strategy for RL methods other than the one proposed with action-value methods.

4.2.1 Introduction To Actor-Critic Methods

This subsection introduces the classic Actor-Critic methods used in Reinforcement Learning developped in the Sutton's book [6].

Algorithm 2 Deep Q-Learning $\pi \approx \pi^*$ (taken from [8])

Initialize :

Initialize replay memory $\mathcal{D}$ to capacity $\mathcal{N}$

Initialize action-value function $\widehat{q}$ with random weights

loop

Initialize $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for each step of episode $t = 1, \cdots, T$ **do**

With probability ϵ select a random action $a_t \in \Gamma(s_t)$ otherwise select $a_t = \arg\max_a \widehat{q}(\phi(s_t), a, \mathbf{w})$

Execute action a_t in emulator and observe reward $r_t = R(s_t, a_t)$

Preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \widehat{q}(\phi_{j+1}, a', \mathbf{w}^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $(y_j - \widehat{q}(\phi_j, a_j, \mathbf{w}))^2)$ with respect to the network parameters $\mathbf{w}$

Every K steps, $\mathbf{w}^- \leftarrow \mathbf{w}$

end for

end loop

4.2.1.1 Policy Gradient Methods

So far only action-value methods were considered. Instead of learning value functions to know which action to select, the new class of methods introduced in this section parameterizes the policy π with a weight vector $\boldsymbol{\theta} \in \mathbb{R}^d$. A value function may still be used but only to help to learn the policy parameters $\boldsymbol{\theta}$. From now on, $\pi(a|s, \boldsymbol{\theta})$ depicts the probability to take an action $a \in \Gamma(s)$ if the state is $s \in \mathcal{S}$ and if the vector parameter is $\boldsymbol{\theta}$. The policy can be parametrized as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters.

A **Stochastic Gradient Ascent** method adjusts the weight vector $\boldsymbol{\theta} \in \mathbb{R}^d$ at each step t by a small amount in the direction that would most maximize the scalar performance measure $J(\boldsymbol{\theta})$. The update is given by :

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{4.2}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^d$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$ and α is the step size parameter. Such methods using the update rule 4.2 are called **Policy Gradient Methods** [6].

Contrary to action-value methods, Policy Gradient methods can work with infinite action space, can find an optimal stochastic policy as well as an optimal deterministic policy (not an ϵ-greedy policy). Besides, the action probability distribution changes smoothly with Policy Gradient methods.

4.2.1.2 Policy Gradient Theorem

For episodic tasks, the performance $J(\boldsymbol{\theta})$ is defined as :

$$J(\boldsymbol{\theta}) = V_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{4.3}$$

where $V_{\pi_{\boldsymbol{\theta}}}$ the true value function for policy $\pi_{\boldsymbol{\theta}}$ parameterized by vector $\boldsymbol{\theta}$ and $s_0 \in \mathcal{S}_0$ is the initial state drawn from the distribution ψ. Changing the policy parameters $\boldsymbol{\theta}$ in a way that ensures improvement is not easy. It is difficult because performance $J(\boldsymbol{\theta})$ depends on both the action selection (determined directly by $\pi_{\boldsymbol{\theta}}$) and the stationary distribution of states following the target selection behavior (determined indirectly by $\pi_{\boldsymbol{\theta}}$). The effect of the policy on the state distribution is a function of the environment and it is typically unknown. However, the **Policy Gradient Theorem** [6] offers a solution that does not involve the derivative of the state distribution. Indeed,

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \Gamma(s)} Q_\pi(s,a) \nabla \pi(a|s, \boldsymbol{\theta}) \tag{4.4}$$

$$= \mathbb{E}_\pi[\sum_{a \in \Gamma(s)} Q_\pi(s,a) \nabla \pi(a|s, \boldsymbol{\theta})] \tag{4.5}$$

where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, π denotes the policy parameterized by the weight vector $\boldsymbol{\theta}$ and $\mu \in \mathscr{P}(\mathcal{S})$ is the state distribution under the policy π. A proof for the Policy Gradient Theorem is given in the Sutton's book [6].

4.2.1.3 REINFORCE Algorithm

From the Policy Gradient Theorem (4.5):

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \left[\sum_a Q_\pi(s_t, a) \pi(a|s_t, \boldsymbol{\theta}) \frac{\nabla \pi(a|s_t, \boldsymbol{\theta})}{\pi(a|s_t, \boldsymbol{\theta})} \right] \tag{4.6}$$

$$\propto \mathbb{E}_\pi \left[Q_\pi(s_t, a_t) \frac{\nabla \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})} \right] \tag{4.7}$$

$$\propto \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})} \right] \text{ because } \mathbb{E}(G_t|s_t, a_t) = Q_\pi(s_t, a_t) \tag{4.8}$$

Knowing that and using the stochastic gradient ascent rule 4.2, we have the update :

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(a_t|s_t, \boldsymbol{\theta})}{\pi(a_t|s_t, \boldsymbol{\theta})} = \boldsymbol{\theta}_t + \alpha G_t \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \tag{4.9}$$

The update (4.9) defines the **REINFORCE method**. The term G_t makes sense because the parameters move to most in the direction that favors actions that yield the highest return. Actions that are selected frequently are at an advantage and might win out even if they do not yield the highest return. For this reason, the $\pi(a_t|s_t, \boldsymbol{\theta})$ term in (4.9) is used to avoid this issue. $\nabla \ln(\pi(A_t|S_t, \boldsymbol{\theta}))$ vector is also known as the **eligibility vector**.

Because the use of G_t, the REINFORCE method has to wait until the end of the episode, it is considered as a Monte Carlo method. Such as any Monte Carlo methods, REINFORCE has a high variance and a slow convergence.

4.2.1.4 REINFORCE Algorithm With A Baseline

The Policy Gradient Theorem (4.5) can be generalized to include a comparison of the action-value to an arbitrary **baseline** $b(s)$ (for $s \in \mathcal{S}$) :

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a (Q_\pi(s,a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta}) \tag{4.10}$$

The baseline $b(s)$ can be any function as long as it does not vary with action a. Equation (4.10) is still valid because :

$$\sum_a b(s) \nabla \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0$$

The update rule of the REINFORCE method with baseline [6] is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha (G_t - b(s_t)) \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta}))$$

It is a generalization of the REINFORCE method. The use of a baseline reduces the variance and speeds up the convergence. For example if in some states all actions have high value, we need a high baseline to differentiate the higher valued actions from the less highly valued ones. A natural choice is to choose as baseline the estimate of the state value function $\widehat{v}(s_t, \mathbf{w})$ with $\mathbf{w} \in \mathbb{R}^{d'}$ as weight vector.

4.2.1.5 Actor-Critic Methods

Actor-Critic methods [6] are a combination of both Policy Gradient methods and Action-Value methods. The **Actor** term refers to the learned policy and the **Critic** term refers to the learned value function. The critic observes agent's actions and provides feedback. Learning from this feedback, the agent updates its policy and becomes better at playing that game. For its part, Critic also updates its parameters to provide feedback to become better for the next time. In other words, the critic measures how good a selected action is and the actor controls how the agent behaves.

In the same way as for the Semi-Gradient SARSA method, the idea is to replace the expected return G_t with the target $R(s_t, a_t) + \gamma \widehat{v}(s_{t+1}, \mathbf{w_t})$ where $\widehat{v}(s_t, \mathbf{w}_t)$ is the critic parameterized function estimating the state value function with $\mathbf{w}_t \in \mathbb{R}^{d'}$ as weight vector. The update rule becomes thus:

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha (R(s_t, a_t) + \gamma \widehat{v}(s_{t+1}, \mathbf{w}_t) - \widehat{v}(s_t, \mathbf{w}_t)) \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \\ &= \boldsymbol{\theta}_t + \alpha \delta_t \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \end{aligned}$$

where $\delta_t = R(s_t, a_t) + \gamma \widehat{v}(s_{t+1}, \mathbf{w}_t) - \widehat{v}(s_t, \mathbf{w}_t)$ is called the **TD error**.

Contrary to the REINFORCE methods, Actor-Critic methods update their weights at every time step of an episode. An example of a Pseudo-Code for Actor-Critic method is given by Algorithm 3.

Algorithm 3 Example of Actor-Critic for estimating $\pi_{\boldsymbol{\theta}} \approx \pi^*$ (taken from [6])

Require: a differentiable policy parametrization $\pi(a|s, \boldsymbol{\theta})$
Require: a differentiable state-value function parameterizaton $\widehat{v}(s, \mathbf{w})$
Require: step-size $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{w} > 0$
Initialize :
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ and state-value heights $\mathbf{w} \in \mathbb{R}^{d'}$(e.g to $\mathbf{0}$)
loop
Initialize s (first state of episode)
$I \leftarrow 1$
while s is not terminal **do**
$a \sim \pi(.|s, \boldsymbol{\theta})$
Take action a, observe s' and $R(s, a)$
$\delta \leftarrow R(s, a) + \gamma \widehat{v}(s', \mathbf{w}) - \widehat{v}(s, \mathbf{w})$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \widehat{v}(s, \mathbf{w})$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln(\pi(a|s, \boldsymbol{\theta}))$
$I \leftarrow \gamma I$
$s \leftarrow s'$
end while
end loop

4.2.2 A2C In Deep RL

This sections deals again with Deep Reinforcement Learning methods using neural networks. In the previous section, Actor-Critic methods used an estimate of the state value function as critic. However, state-value functions have a high variability [6]. A solution is to use **advantage functions** instead of value functions (see the advantage function definition in appendix B.1.1). Advantage function captures how better is an action compared to the others at a given state. Indeed, $A_\pi(s, a)$ gives the extra reward earned if the action $a \in \Gamma(s)$ is taken in the state $s \in \mathcal{S}$. if $A_\pi(s_t, a_t) > 0$ the gradient is pushed in that direction otherwise the gradient is pushed in the opposite direction. Actor-Critic methods using advantage function as critic are called **Advantage Actor-Critic methods** (A2C) [10].

By definition, estimating the advantage function requires both an estimate of the action-value function Q_π and of the state-value function V_π. Fortunately, there exists a trick avoiding the implementation of another neural network estimating Q_π. Indeed, the advantage function is defined as:

$$\begin{aligned} A_\pi(s_t, a_t) &= Q_\pi(s_t, a_t) - V_\pi(s_t) && (4.11)\\ &= \mathbb{E}_\pi \left[R(s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \right] && (4.12)\\ &\approx \mathbb{E}_\pi \left[R(s_t, a_t) + \gamma \widehat{v}(s_{t+1}, \mathbf{w}_t) - \widehat{v}(s_t, \mathbf{w}_t) \right] = \mathbb{E}_\pi \left[\delta_t \right] && (4.13) \end{aligned}$$

where $\widehat{v}(s_t, \mathbf{w}_t)$ is the critic parameterized function estimating the state value function with $\mathbf{w}_t \in \mathbb{R}^{d'}$ as weight vector. The TD error δ_t is thus an estimate of the advantage function.

According to 4.8, the Advantage Actor-Critic loss is defined as :

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \mathbb{E}_\pi \left[A_\pi(s_t, a_t) \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \right] \\ &\approx \mathbb{E}_\pi \left[\delta_t \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \right] \end{aligned}$$

Usually, A2C methods have a single network with two outputs: one softmax output for the policy prediction $\pi(a_t|s_t, \boldsymbol{\theta})$) and another one for the the state value function prediction $\widehat{v}(s_t, \mathbf{w}_t)$. Both predictions use parameters defined in the first layers [10].

Contrary to most of action-value methods, Advantage Actor-Critic methods do not explore with ϵ-greedy policy and implement another strategy named **entropy** [11]. Entropy is used in many scientific fields and it was introduced for the first time in physics to denote the lack of order within a system. In RL, entropy relates directly to the unpredictability of the actions which an agent takes in a given policy. The greater the entropy, the more random the actions an agent takes. While learning, entropy of the action selection policy decreases and the policy becomes more deterministic. To keep exploration and encourage the agent to take actions more unpredictably, entropy is added to the loss function :

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[A_\pi(s_t, a_t) \nabla \ln(\pi(a_t|s_t, \boldsymbol{\theta})) \right] + \beta \nabla H(\pi(.|\boldsymbol{\theta}))$$

where β controls the strength of the entropy regularization term and H is the entropy defined as :

$$H(\pi(.|\boldsymbol{\theta}) = -\mathbb{E}_\pi[\pi(a_t|s_t, \boldsymbol{\theta}) \ln(\pi(a_t|s_t, \boldsymbol{\theta}))] \tag{4.14}$$

Entropy is a good way for learning alternative ways of accomplishing the task. Moreover, it gives a better **generalization** for the learning process. Generalization is useful under a number of circumstances, all of which related to changes in the agent's knowledge of the environment, or changes in the environment itself over time.

4.3 Trust Region Policy Optimization And Its derivatives

It is very to difficult to choose a good step-size parameter for Advantage Actor-Critic method because: input data is non-stationary due to changing policy and observation and reward distributions change all the time. Besides, the choice of a bad step-size parameter affects the state distribution and then hurts very badly performance. With a bad policy, the methods is going to explore another part of the state space that is not necessarily interesting and learn from that. The performance will collapse and it will take a long time if ever, to recover. Consequently, because the step-size parameter is not tuned correctly Advantage Actor-Critic does not perform well most of the time. **Trust Region Policy Optimization** (TRPO) [2] is an Advantage Actor-Critic method which solves this issue by constraining the policy so the policy does not make aggressive and hurtful moves.

4.3.1 Natural Policy Gradient And Trust Region Policy Optimization

Let π denote a stochastic policy and $J(\pi)$ denoting its expected return :

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right] \tag{4.15}$$

The expected return of another policy $\tilde{\pi}$ can be expressed in terms of the advantage over π, accumulated over timesteps as shown in [2]:

$$J(\tilde{\pi}) = J(\pi) + \mathbb{E}_{\tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t) \right] \tag{4.16}$$

Let ρ_π be the discounted visitation frequencies :

$$\rho_\pi(s) = \sum_{t=0}^{\infty} \gamma^t Pr(s_t = s|\pi) = \psi(s) + \sum_{t=1}^{\infty} \gamma^t \mu(s)$$

where $s \in \mathcal{S}$, $\psi \in \mathscr{P}(\mathcal{S})$ is the MDP state distribution for initial states and $\mu \in \mathscr{P}(\mathcal{S})$ is the state distribution given by the policy π in a specific MDP. Equation (4.16) can be written as a sum over states instead of time steps:

$$J(\tilde{\pi}) = J(\pi) + \sum_{t=0}^{\infty} \sum_{s} Pr(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) \gamma^t A_\pi(s, a) \tag{4.17}$$

$$= J(\pi) + \sum_{s} \sum_{t=0}^{\infty} \gamma^t Pr(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) A_\pi(s, a) \tag{4.18}$$

$$= J(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s, a) \tag{4.19}$$

If there is an update $\pi \to \tilde{\pi}$ where $\sum_a \tilde{\pi}(a|s) A_\pi(s, a) \geq 0$, there is a guarantee of improvement of performance J, or at least a constant performance in the case of equality. It is a reformulation of the Policy Improvement Theorem conditions 2.9. The dependency in Equation 4.19 between $\tilde{\pi}$ and $\rho_{\tilde{\pi}}$ makes difficult an optimization. For this reason, we define:

$$L_\pi(\tilde{\pi}) = J(\pi) + \sum_{s} \rho_\pi(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s, a) \tag{4.20}$$

L_π ignores changes in state visitation density due to changes in the policy. Kakade and Langford [12] demonstrate that L_π approximates locally J for the first order:

$$L_{\pi_{\theta_0}}(\pi_{\theta_0}) = J(\pi_{\theta_0}), \tag{4.21}$$

$$\nabla L_{\pi_{\theta_0}}(\pi_{\theta_0})|_{\theta=\theta_0} = \nabla J(\pi_\theta)|_{\theta=\theta_0} \tag{4.22}$$

Therefore, a sufficiently small step $\pi_{\theta_0} \to \tilde{\pi}$ that improves $L_{\pi_{\theta_0}}$ will also improve J, but does not give us any guidance on how big of a step to take. Besides, L_π is a lower bound function approximating J locally at the current policy:

$$J(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi}) \tag{4.23}$$

where :

$$C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$$

$$\epsilon = \max_{s,a} |A_\pi(s, a)|$$

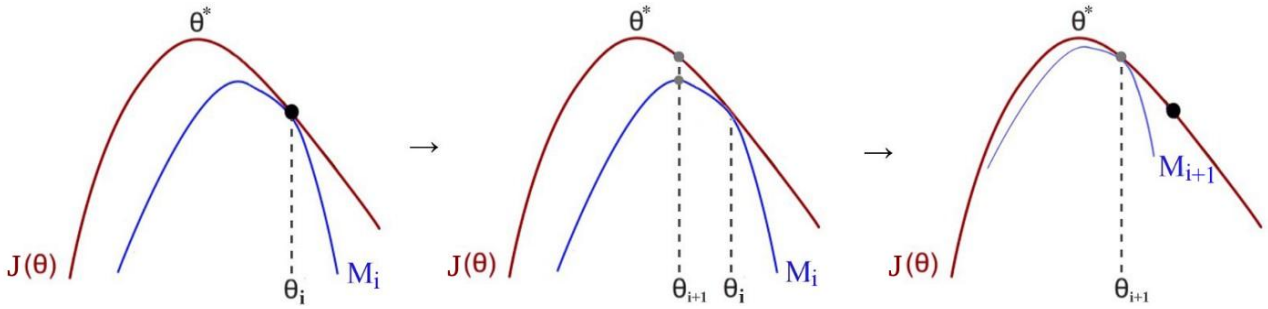


Figure 4.1: Principle of the MM algorithm (adapted from `https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12`)

The proof of this inequality is given in [2].

With $M_i(\pi) = L_{\pi_i}(\pi) - CD_{KL}^{\max}(\pi_i, \pi)$, any improvement in $M_i(\pi_{i+1})$ over $M_i(\pi_i)$ will also punch $J(\pi_{i+1})$ up at least the same amount.

$$\begin{aligned} J(\pi_{i+1}) &\geq M_i(\pi_{i+1}) \text{ by Equation 4.23,} \\ J(\pi_i) &= M_i(\pi_i) \text{ by definition} \\ J(\pi_{i+1}) - J(\pi_i) &\geq M_i(\pi_{i+1}) - M_i(\pi_i) \end{aligned}$$

Such algorithm maximizing M_i to improve J is a **Minorization-Maximization** algorithm (MM algorithm). MM algorithm treats maximization problem by finding an approximated lower bound of the original objective as the surrogate objective and then maximizes the approximated lower bound so as to optimize the original objective. The principle of an MM algorithm is summarized in Figure 4.1.

From now on, we are going to consider again the case where the policy π is parameterized with a vector $\boldsymbol{\theta}$. From now on, notations will be adapted to parameterization : $J(\boldsymbol{\theta}) := J(\pi_{\boldsymbol{\theta}})$, $L_{\boldsymbol{\theta}}(\boldsymbol{\theta}) := L_{\pi_{\boldsymbol{\theta}}}(\pi_{\boldsymbol{\theta}})$, $\rho_{\pi_{\boldsymbol{\theta}}} := \rho_{\boldsymbol{\theta}}$ and $D_{KL}(\boldsymbol{\theta} \parallel \tilde{\boldsymbol{\theta}}) := D_{KL}(\pi_{\boldsymbol{\theta}} \parallel \pi_{\tilde{\boldsymbol{\theta}}})$. Besides, $\boldsymbol{\theta}_{\text{old}}$ denotes the previous policy parameters that we want to improve upon According to the MM algorithm and the inequality 4.23, to improve the policy π at each step, the following maximization need to be performed :

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \quad L_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{\theta}) - CD_{KL}^{\max}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) \tag{4.24}$$

However, in practice the use of the coefficient C implies a small step size parameter. One way to take larger steps in a robust way is to use a constraint δ on the KL divergence between the new policy and the old policy. Such constraint defines a **trust region**. With the Lagrangian Duality, the objective is mathematically the same as the following using a trust region constraint.

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{maximize}} \quad & L_{\boldsymbol{\theta}_{\text{old}}}(\boldsymbol{\theta}) \\ \text{subject to} \quad & D_{KL}^{\max}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) \leq \delta \end{aligned} \tag{4.25}$$

In practice, the trust region parameter δ is much easier to tune than the parameter C. The problem is that finding the maximum KL divergence (among all policies) is intractable. The requirement is relaxed and the mean of the KL divergence is used instead of the maximum.

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \sum_s \rho_{\boldsymbol{\theta}_{\text{old}}}(s) \sum_a \pi_{\boldsymbol{\theta}(a|s)} A_{\boldsymbol{\theta}_{\text{old}}}(s,a) \text{ definition 4.20} \\ \text{subject to} \quad & \overline{D}_{KL}^{\rho_{\boldsymbol{\theta}_{\text{old}}}}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) \leq \delta \end{aligned} \tag{4.26}$$

where $\overline{D}^{\rho}_{KL}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \mathbb{E}_{s\sim\rho}\left[D_{KL}(\pi_{\boldsymbol{\theta}_1}(.|s) \parallel \pi_{\boldsymbol{\theta}_2}(.|s))\right]$ In practice, it is easier to use data from old policies:

$$\begin{aligned}
&\sum_s \rho_{\boldsymbol{\theta}_{\text{old}}}(s) \sum_a \pi_{\boldsymbol{\theta}}(a|s) A_{\boldsymbol{\theta}_{\text{old}}}(s,a) \\
&= \mathbb{E}_{s\sim\rho_{\boldsymbol{\theta}_{\text{old}}}, a\sim\pi_{\boldsymbol{\theta}}}\left[A_{\boldsymbol{\theta}_{\text{old}}}(s,a)\right] \\
&= \mathbb{E}_{s\sim\rho_{\boldsymbol{\theta}_{\text{old}}}, a\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a,s)} A_{\boldsymbol{\theta}_{\text{old}}}(s,a)\right]
\end{aligned}$$

Therefore, the problem becomes:

$$\begin{aligned}
\underset{\boldsymbol{\theta}}{\text{maximize}} \quad & \mathbb{E}_{s\sim\rho_{\boldsymbol{\theta}_{\text{old}}}, a\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}(a|s)}}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a,s)} A_{\boldsymbol{\theta}_{\text{old}}}(s,a)\right] \\
\text{subject to} \quad & \overline{D}^{\rho_{\boldsymbol{\theta}_{\text{old}}}}_{KL}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) \leq \delta
\end{aligned} \tag{4.27}$$

or equivalently:

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \quad \mathbb{E}_{s\sim\rho_{\pi_{\text{old}}}, a\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}(a|s)}}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a,s)} A_{\boldsymbol{\theta}_{\text{old}}}(s,a)\right] - C\overline{D}^{\rho_{\boldsymbol{\theta}_{\text{old}}}}_{KL}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) \tag{4.28}$$

The problem is solved by using the first order Taylor approximation for the objective and the second order for the KL divergence :

$$\begin{aligned}
L_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(\boldsymbol{\theta}) &\approx L_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(\boldsymbol{\theta}_{\text{old}}) + \mathbf{g}^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) \quad \text{with } \mathbf{g} = \nabla L_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_{\text{old}}} \\
\overline{D}^{\rho_{\boldsymbol{\theta}_{\text{old}}}}_{KL}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta}) &\approx \tfrac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}})^\top \mathbf{F}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) \qquad \text{with } \mathbf{F} = \nabla^2 \overline{D}^{\rho_{\boldsymbol{\theta}_{\text{old}}}}_{KL}(\boldsymbol{\theta}_{\text{old}}, \boldsymbol{\theta})|_{\boldsymbol{\theta}_{\text{old}}}
\end{aligned} \tag{4.29}$$

The matrix $\mathbf{F}$ is called the **Fisher Information Matrix** [2]. The quadratic part of $L_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(\boldsymbol{\theta})$ is negligible compared to the KL divergence quadratic term. The problem becomes easier:

$$\begin{aligned}
\underset{\boldsymbol{\theta}}{\text{maximize}} \quad & \mathbf{g}^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) \\
\text{subject to} \quad & \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}})^\top \mathbf{F}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) \leq \delta
\end{aligned} \tag{4.30}$$

or :

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \quad g^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) - C\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}})^\top \mathbf{F}(\boldsymbol{\theta} - \boldsymbol{\theta}_{\text{old}}) \tag{4.31}$$

It can be solved analytically with the solution:

$$\boldsymbol{\theta} = \boldsymbol{\theta}_{\text{old}} + \sqrt{\frac{2\delta}{\mathbf{g}^\top \mathbf{F}^{-1} \mathbf{g}}} \mathbf{F}^{-1}\mathbf{g} = \boldsymbol{\theta}_{\text{old}} + \alpha \mathbf{F}^{-1}\mathbf{g} \tag{4.32}$$

where $\alpha = \sqrt{\frac{2\delta}{\mathbf{g}^\top \mathbf{F}^{-1} \mathbf{g}}}$

A proof of this result is given in the course [13].

$\tilde{\nabla} J(\boldsymbol{\theta}) = \mathbf{F}^{-1}\mathbf{g}$ is called the **natural gradient** of $J(\boldsymbol{\theta})$. The gradient $\nabla J(\boldsymbol{\theta})$ gives the steepest direction to maximize rewards in the parameter space whereas $\tilde{\nabla} J(\boldsymbol{\theta})$ is the steepest direction in the policy space into the corresponding parameter space. However, computing the natural policy gradient is very expensive because the computation of the Fisher Matrix $\mathbf{F}$ and its inverse. For this reason, **Trust Region Policy Optimization** (TRPO) [2] uses the **conjugate gradient method** combined with a **Line Search** method [14] to optimize the computation of the natural policy gradient.

4.3.2 Proximal Policy Optimization (PPO)

Even if TRPO performs better than classic A2C methods, there are several limitations because it is hard to use with architectures with multiple outputs (one for computing action probabilities and another one the state-value prediction), it empirically performs poorly on tasks requiring deep CNNs or RNNs and it is difficult to implement.

Proximal Policy Optimization (PPO) [15] is a method performing better than TRPO in many tasks. It makes a first order derivative solution closer to the second-order derivative solution by adding soft constraints. There is still a chance of having a bad policy decision once a while with the first-order solutions as the stochastic gradient descent. But soft constraints are added into the objective function so the optimization has better insurance to optimize within a trust region, the chance of bad decision becomes smaller. To optimize a policy π, PPO alternates between sampling data from the policy π and performing several epochs of optimization on the sampled data.
Let $r_t(\boldsymbol{\theta})$ be the ratio :

$$r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)} \tag{4.33}$$

TRPO seeks to maximize the following surrogate function (given by equation 4.27) :

$$L^{\text{TRPO}}(\boldsymbol{\theta}) = \mathbb{E}_\pi \left[\frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)} \widehat{a}_t(s_t, a_t) \right] = \mathbb{E}_\pi \left[r_t(\boldsymbol{\theta}) \widehat{a}_t(s_t, a_t) \right] \tag{4.34}$$

where $\widehat{a}_t$ is an estimate of the advantage function A_π at time t. Because the current policy is improved at each epoch, the difference between the current and the old policy is getting larger. The variance of the estimation increases and bad decision can be made because of the inaccuracy. Objective is thus changed in PPO, to penalize changes to the policy that move $r_t(\boldsymbol{\theta})$ away from 1 :

$$L^{\text{PPO}}(\boldsymbol{\theta}) = \mathbb{E}_t \left[\min(r_t(\boldsymbol{\theta})\widehat{a}_t(s_t, a_t), \text{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\widehat{a}_t(s_t, a_t)) \right] \tag{4.35}$$

where ϵ is a hyperparameter (authors suggest to set $\epsilon = 2$ [3]).
$L^{\text{PPO}}(\boldsymbol{\theta}) = L^{\text{TRPO}}(\boldsymbol{\theta})$ around $\boldsymbol{\theta}_{\text{old}}$ but becomes different as long as $\boldsymbol{\theta}$ moves way from $\boldsymbol{\theta}_{\text{old}}$ Clipping prevents policy from having incentive to go far away from the old policy. Indeed, if the probability ratio between the new policy and the old policy falls outside the range $(1-\epsilon)$ and $(1+\epsilon)$, the advantage function will be clipped. By taking the minimum of the clipped and unclipped objective, the final objective becomes a lower bound. Figure 4.2 resumes the effect of clip on the advantage function. An example of Pseudo-Code for the PPO algorithm is given by the Algorithm 4.

Authors [3] strongly suggest the use of the **Generalized Advantage Estimation** [16] (GAE) to estimate the advantage function. Equation (4.13) shows that the TD error estimates the advantage function. A new class of advantage function estimators can be defined by generalizing the TD error on more successor states:

$$\begin{aligned}
\widehat{a}_t^{(1)} &= R(s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \\
\widehat{a}_t^{(2)} &= R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \gamma^2 V_\pi(s_{t+1}) - V_\pi(s_t) \\
\cdots &= \cdots \\
\widehat{a}_t^{(\infty)} &= R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \gamma^2 R(s_{t+2}, a_{t+2}) + \cdots - V_\pi(s_t)
\end{aligned}$$

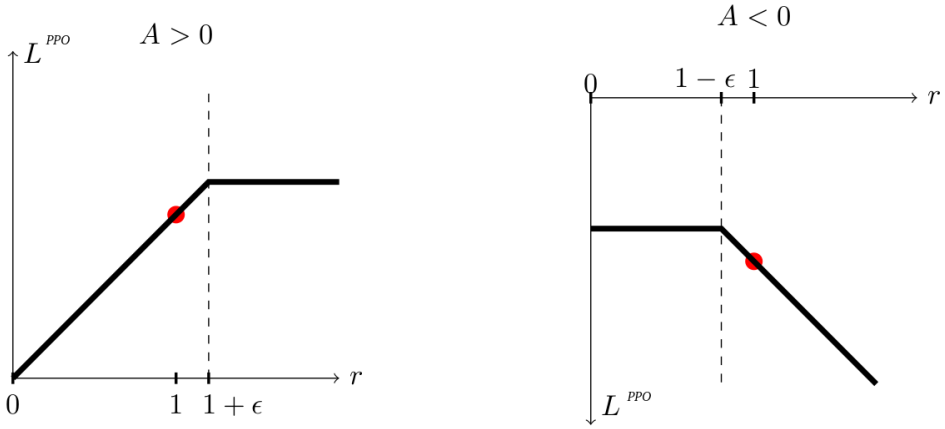


Figure 4.2: Plots showing one term (i.e., a single timestep) of the surrogate function L^{PPO} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{PPO} sums many of these terms. (taken from [15])

Algorithm 4 PPO (taken from [14])

Require: Initial input policy parameters $\boldsymbol{\theta}_0$, clipping the threshold ϵ
  for $k = 0, 1, 2, \cdots$ **do**
    Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_{\boldsymbol{\theta}_k}$
    Estimate advantages $\widehat{a}_t^{\pi_{\boldsymbol{\theta}_k}}$ using any advantage estimation algorithm
    Compute the policy update $\boldsymbol{\theta}_{k+1} = \arg\max_\theta L_{\boldsymbol{\theta}_k}^{\text{PPO}}(\boldsymbol{\theta})$ by taking K steps of minibatch SGD (via Adam).
  end for

Estimators $\widehat{a}_t^{(k)}$ with small k have low variance but high bias, whereas those with large k have low bias but high variance because the $\gamma^k V_\pi(s_t)$ becomes smaller as long as $k \to \infty$.
GAE approximates the advantage function by averaging all of these estimates :

$$\widehat{a}_t^{\text{GAE}(\gamma,\lambda)} = (1-\lambda)(\widehat{a}_t^{(1)} + \lambda \widehat{a}_t^{(2)} + \lambda^2 \widehat{a}_t^{(3)} + \cdots) \tag{4.36}$$

$$= (1-\lambda)(\delta_t + \lambda(\delta_t + \delta_{t+1}) + \lambda^2(\delta_t + \delta_{t+1} + \delta_{t+2}) + \cdots) \tag{4.37}$$

$$= (1-\lambda)\left(\delta_t \frac{1}{1-\lambda} + \delta_{t+1}\frac{\lambda}{1-\lambda} + \cdots\right) \tag{4.38}$$

$$= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{4.39}$$

where $\delta_t = R(s_t, a_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$ is the TD error at time step t and $\lambda \in [0, 1]$ is an hyperparameter adjusting the bias-variance tradeoff.

4.3.3 Actor-Critic Using Kronecker-Factored Trust Region (ACKTR)

As well as PPO, **Actor-Critic using Kronecker-Factored Trust Region** (ACKTR) [17] is an improvement of TRPO. In TRPO, the computation of the natural gradient $\tilde{\nabla} J(\boldsymbol{\theta}) = \mathbf{F}^{-1}\mathbf{g}$ is expensive in computation despite the use of conjugate gradient. Contrary to PPO, ACKTR is a second order optimization. ACKTR approximates the natural gradient layer per layer with the **Kronecker-factored Approximate Curvature** (K-FAC) [18] introduced in Appendix B.2.

Critic can also be optimized by applying natural gradient. Usually, learning the critic can be thought of as a least-squares function approximation problem. In the setting of least-squares function approximation, the second-order algorithm of choice is commonly **Gauss-Newton**, which approximates the curvature as the Gauss-Newton matrix $G = \mathbb{E}[\mathbf{J}^\top \mathbf{J}]$, where $\mathbf{J}$ is the Jacobian of the mapping from parameters to outputs. The Gauss-Newton matrix is equivalent to the Fisher matrix for a Gaussian observation model; this equivalence allows us to apply K-FAC to the critic as well. Because there is equivalence, we assume the critic $\widehat{v}$ is defined to be a Gaussian distribution $p(\widehat{v}|s_t) \sim \mathcal{N}(\widehat{v}; V_\pi(s_t), \sigma^2)$. The Fisher matrix for the critic is defined with respect to the Gaussian output distribution. In practice, we can simply set $\sigma = 1$, which is equivalent to the vanilla Gauss-Newton method.

For the layers shared by actor and critic networks, there is an Independence assumption of the joint distribution of the policy and the value distribution : $Pr(a_t, \widehat{v}_t|s_t) = p(a_t|s_t)p(\widehat{v}_t|s_t)$ then K-FAC is applied to :

$$\mathbf{F} = \mathbb{E}\left[\frac{d\log(p(a_t, \widehat{v}_t|s_t, \boldsymbol{\theta}))}{d\boldsymbol{\theta}} \frac{d\log(p(a_t, \widehat{v}_t|s_t, \boldsymbol{\theta}))^\top}{d\boldsymbol{\theta}}\right]$$

to perform updates simultaneously.

4.4 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) [4] is an off-policy Actor-Critic method incorporating the entropy measure [11] (see entropy in section 4.2.2) of the policy into the reward to encourage exploration. SAC is used only on continuous action-space $\mathcal{A}$. The SAC objective is to learn a policy that acts as randomly as possible while it is still able to succeed at the task. SAC was designed for real-world robotic applications. Contrary to the previous Actor-Critic methods, SAC learns three distinct neural networks for the policy (with parameters $\boldsymbol{\theta}$), for the soft Q-value estimates (with parameters $\mathbf{w}$) and for the soft state estimates (with parameters $\boldsymbol{\psi}$). The policy is trained with the objective to maximize the expected return and the entropy at the same time:

$$J(\boldsymbol{\theta}) = \sum_{t=0}^{\infty} \mathbb{E}_\pi \left[\gamma^t R(s_t, a_t) + \alpha \mathcal{H}(\pi_{\boldsymbol{\theta}}(.|s_t)\right]$$

where where $\mathcal{H}$ is the entropy measure and α is the **temperature parameter** controlling how important the entropy term is. The entropy maximization leads to policies that can explore more and if there exist multiple options that seem to be equally good, the policy should assign each with an equal probability to be chosen.

Authors defined **soft Q-value** and **soft state-value** as:

$$Q_\pi(s_t, a_t) = R(s_t, a_t) + \gamma \mathbb{E}_\pi \left[V_\pi(s_{t+1}) \right]]$$

where $V_\pi(s_t) = \mathbb{E}_\pi \left[Q_\pi(s_t, a_t) - \alpha \log(\pi(a_t|s_t)) \right]$ [4].

The **soft state value function** $\widehat{v}$ (with parameters $\boldsymbol{\psi}$) is trained to minimize the Mean Squared Error (MSE):

$$J_{\widehat{v}}(\boldsymbol{\psi}) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\frac{1}{2} \left(\widehat{v}(s_t) - \mathbb{E}_\pi \left[\widehat{q}(s_t, a_t) - \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \right] \right)^2 \right]$$

with gradient:

$$\nabla_{\boldsymbol{\psi}} J_{\widehat{v}}(\boldsymbol{\psi}) = \nabla_{\boldsymbol{\psi}} \widehat{v}(s_t) (\widehat{v}(s_t) - \widehat{q}(s_t, a_t) + \log \pi_{\boldsymbol{\theta}}(a_t|s_t))$$

where $\mathcal{D}$ is the replay buffer.
The **soft action-value function** $\widehat{q}$ is trained to minimize the **soft Bellman residual** (see [4]):

$$J_{\widehat{q}}(\mathbf{w}) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \left[\frac{1}{2} (\widehat{q}(s_t, a_t) - \left(R(s_t, a_t) + \gamma \mathbb{E}_\pi \left[\widehat{v}_{\bar{\boldsymbol{\psi}}}(s_{t+1}) \right] \right) \right)^2 \right]$$

with gradient:

$$\nabla_{\mathbf{w}} J_{\widehat{q}}(\mathbf{w}) = \nabla_{\mathbf{w}} \widehat{q}(s_t, a_t) (\widehat{q}(s_t, a_t) - R(s_t, a_t) - \gamma \widehat{v}_{\bar{\boldsymbol{\psi}}}(s_{t+1}))$$

where $\bar{\boldsymbol{\psi}}$ is the target value function which is the exponential moving average, just like how the parameter of the target Q network is treated in DQN to stabilize the training.
SAC updates the policy to minimize the KL-divergence [4]:

$$\begin{aligned} \pi_{\text{new}} &= \arg\min_{\pi'} \mathrm{D}_{KL} \left(\pi(.|s_t) || \frac{\exp(Q_{\pi_{\text{old}}}(s_t, .)}{Z_{\pi_{\text{old}}}(s_t)} \right) \\ &= \arg\min_{\pi} \mathrm{D}_{KL} \left(\pi(.|s_t) || \exp(Q_{\pi_{\text{old}}}(s_t, .) - \log Z_{\pi_{\text{old}}}(s_t)) \right) \end{aligned}$$

where $Z_{\pi_{\text{old}}}(s_t)$ is the partition function to normalize the distribution. It is usually intractable but does not contribute to the gradient.
The objective is thus:

$$\begin{aligned} J_\pi(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathrm{D}_{KL} \left(\pi_{\boldsymbol{\theta}}(.|s_t) || \exp(\widehat{q}(s_t, .) - \log Z_{\mathbf{w}}(s_t)) \right) \\ &= \mathbb{E}_\pi \left[\log \pi_{\boldsymbol{\theta}}(a_t|s_t) - \widehat{q}(s_t, a_t) + \log Z_{\mathbf{w}}(s_t) \right] \end{aligned}$$

This update guarantees that $Q_{\pi_{\text{new}}}(s_t, a_t) \geq Q_{\pi_{\text{old}}}(s_t, a_t)$. The proof of this result is given in the paper [4]. The Pseudo-Code of SAC is given by Algorithm 5.

The classic SAC method was defined here. However, there exits some extensions that uses only the action-value function and disposes of the state-value function. There is also an automatic discovery of the temperature parameter α. You can check these extensions on this paper [4]. These extensions were used for the experiments in the next chapters.

Algorithm 5 SAC (taken from [4])

Require: The learning rates $\lambda_\pi, \lambda_{\widehat{q}}, \lambda_{\widehat{v}}$ for function $\pi_{\boldsymbol{\theta}}, \widehat{q}, \widehat{v}$ respectively; The weight factor τ for exponential moving average.

Initialize parameters $\boldsymbol{\theta}, \mathbf{w}, \boldsymbol{\psi}, \bar{\boldsymbol{\psi}}$

for each iteration **do**

  for each environment step t **do**

    $a_t \sim \pi_{\boldsymbol{\theta}}(.|s_t)$

    Take action a_t, observe $s_{t+1}, R(s_t, a_t)$

    Store transition $(s_t, a_t, R(s_t, a_t), s_{t+1})$ in $\mathcal{D}$

  end for

  for each gradient step **do**

    $\boldsymbol{\psi} \leftarrow \boldsymbol{\psi} - \lambda_{\widehat{v}} \nabla_{\boldsymbol{\psi}} J_{\widehat{v}}(\boldsymbol{\psi})$

    $\mathbf{w} \leftarrow \mathbf{w} - \lambda_{\widehat{q}} \nabla_{\mathbf{w}} J_{\widehat{q}}(\mathbf{w})$

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda_{\pi_{\boldsymbol{\theta}}} \nabla_{\boldsymbol{\theta}} J_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\theta})$

    $\bar{\boldsymbol{\psi}} \leftarrow \tau \boldsymbol{\psi} + (1 - \tau) \bar{\boldsymbol{\psi}}$

  end for

end for

Chapter 5

Deep Reinforcement Learning Methods Applied To The Cart-Pole Environment

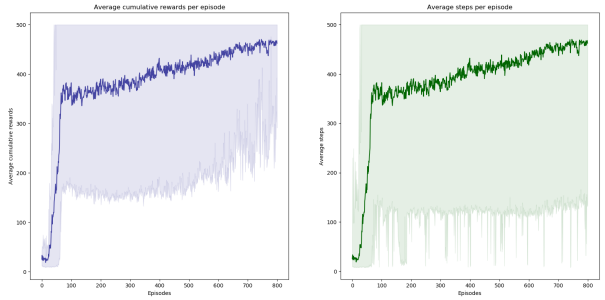
The objective of this chapter is to study the behavior of Deep Reinforcement Learning methods (introduced in Chapter 4) for the Cart-Pole task (introduced in section 3.1).
The following Deep RL methods were implemented from scratch with PyTorch:

- **Action-Value Methods**: DQN (section 4.1) with its extensions:
  - PER (Appendix B.1.2)
  - NoisyNets (Appendix B.1.3)
  - DDQN (Appendix B.1.1)
  - Duel-DQN (Appendix B.1.4)
- **Policy-Gradient Methods**
  - REINFORCE using neural networks (section 4.2.1.3)
- **Advantage Actor-Critic methods**:
  - A2C (section 4.2.2)
  - PPO (section 4.3.2)
  - ACKTR (section 4.3.3)
  - SAC

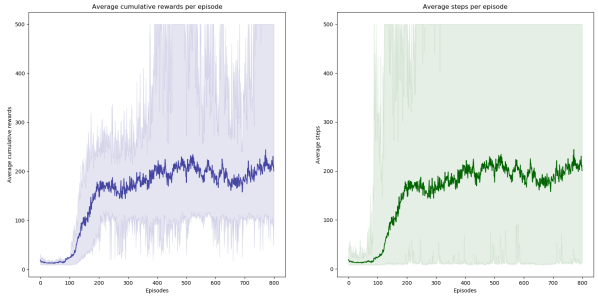
The project is available on GitHub: `https://github.com/DavidBrellmann/DeepRL`.

5.1 Fourier Basis Improvement

Fourier Basis was introduced for the fist time as features encoding for approximating the state-value function in classic RL methods [5, 6]. Efficiency of Fourier Basis on classic RL methods for the Cart-Pole environment is highlighted in Chapter 3. Nevertheless, to the best of my knowledge, no study has been done on the impact of Fourier Basis as features encoding on

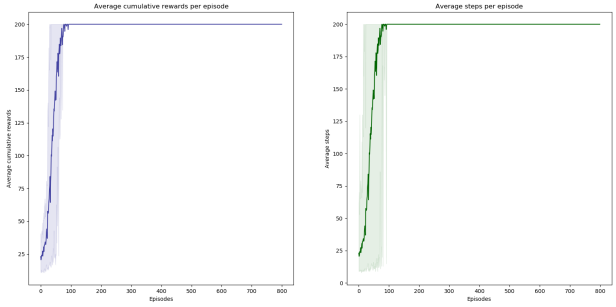


(a) With order-1 Fourier Basis

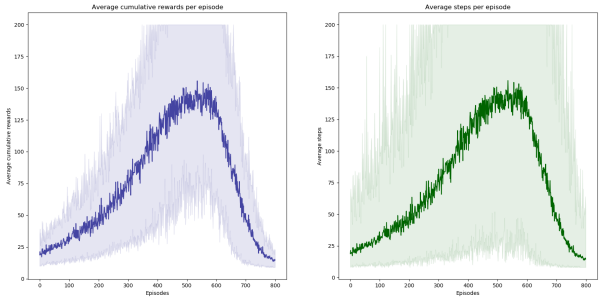


(b) Without order-1 Fourier Basis

Figure 5.1: Training performance of Duel-DQN combined with and without order-1 Fourier Basis over 60 trainings (maximum episode length: 500 steps)



(a) With order-1 Fourier Basis

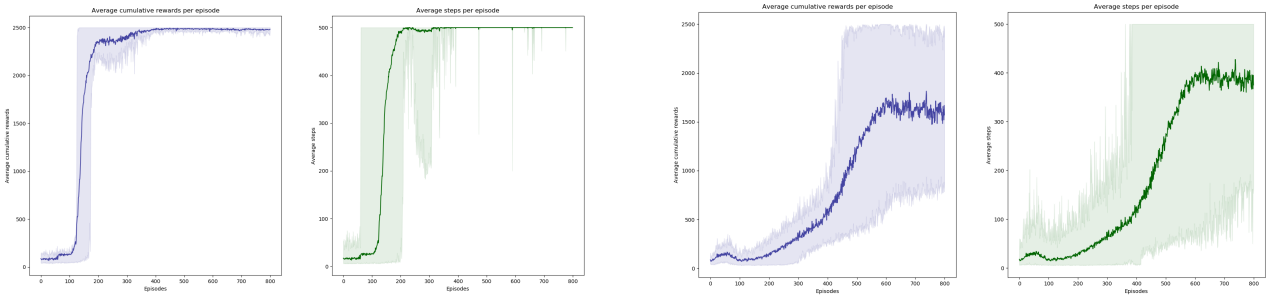


(b) Without order-1 Fourier Basis

Figure 5.2: Training performance of PPO combined with and without order-1 Fourier Basis over 60 trainings (maximum episode length: 200 steps)

Deep Reinforcement Learning methods. The use of Fourier Basis requires the same preprocessing f applied in section 2.4.

The same Cart-Pole environment defined in section 3.1 was used for experiments. Fourier Basis is used to encode the observations returned by the environment and the Deep RL methods take as inputs the features returned by the Fourier Basis.
Figures 5.1 and Figure 5.2 respectively describe the training performance of Duel-DQN and PPO combined with and without order-1 Fourier Basis as features encoding. Hyperparameters for both methods were found with a Grid-Search (see Appendix C.1 for their values). The green curve represents the number of steps per episode during training representing the ability of the Cart-Pole to respect hard constraints (see conditions of termination in section 3.1). The blue curve depicts cumulative rewards per episode during the training. Since we get the same reward at each step, both curves are similar.
There is no doubt that combining Deep RL methods with order-1 Fourier Basis improves performance very significantly. Regardless of the strategy adopted by RL methods, there is an improvement for both Actor-Critic methods (PPO) and value function methods (Duel-DQN). Indeed, without Fourier Basis as features encoding, the training is very slow and the methods fail to converge toward an optimal policy within 800 episodes. It is even observed for PPO that without the use of Fourier Basis as features encoding, the method unlearns from the 500^{th}



(a) With order-1 Fourier Basis

(b) Without order-1 Fourier Basis

Figure 5.3: Training performance of SAC combined with and without order-1 Fourier Basis over 60 trainings (maximum episode length: 500 steps)

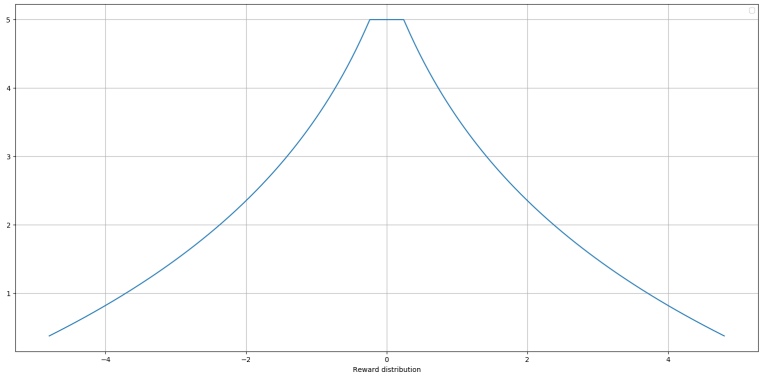


Figure 5.4: Reward Distribution according to the Cart position

episode and converges toward a policy with poor performance. The same method with order-1 Fourier Basis converges on average towards an optimal policy in less than 100 episodes. Note that for PPO without features encoding, no better hyperparameters were found. During the hyperparameter research, it was observed that there are far more sets of hyperparameters converging towards an optimal policy for Deep RL methods combined with Fourier Basis than without features encoding. Such observations suggest that Deep RL methods combined with Fourier Basis are more resistant to the variation of hyperparameters.

In addition, during the research of hyperparamters it seemed that increasing the Fourier Basis order does not improve performance. Indeed, even if performance is still better than without Fourier Basis, it is not as good as with 1-order Fourier Basis. Such observations can be explained because increasing the Fourier order exponentially increases the amount of features [5] and make the learning more difficult.

A similar comparison was made for SAC as illustrated in Figure 5.3. As a reminder, SAC only works on a continuous action space. Thereby, the Gym Cart-Pole environment was customized for a continuous the action space $\mathcal{A} = [0, 1]$. Moreover, the Cart-Pole environment used with SAC has another reward distribution depicted in Figure 5.4. The shape of this reward function will be discussed in more details in section 5.3. Just like PPO and DQN, Fourier Basis significantly improves SAC performance.

Such good performance can be explained because when Fourier Basis is combined with neural network, the first layer of the neural network determines coefficients of an m^{th} Fourier expansion and not approximate without any knowledge a function. It becomes then easier for the layer to approximate the function and the training is thus faster. Moreover, Cart-Pole has few inputs with only four observations therefore the number of features generated by Fourier Basis is not huge (only 16 features). However, as the number of features increases exponentially with the number of inputs, combining Deep RL methods with Fourier Basis on environments with more observations would not necessarily give better performance. Further research are conducted on more environments to answer this question.

From now on, all Deep RL methods are combined with order-1 Fourier Basis.

5.2 Comparison Of Training Performance

The purpose of this section is to compare training performance of Deep RL methods with each other on both deterministic and non-deterministic versions of the Cart-Pole environment. There exists few papers comparing RL methods with each other on the Cart-Pole environment [19] but no papers were found to compare the use of Deep RL methods on the Cart-Pole environment.

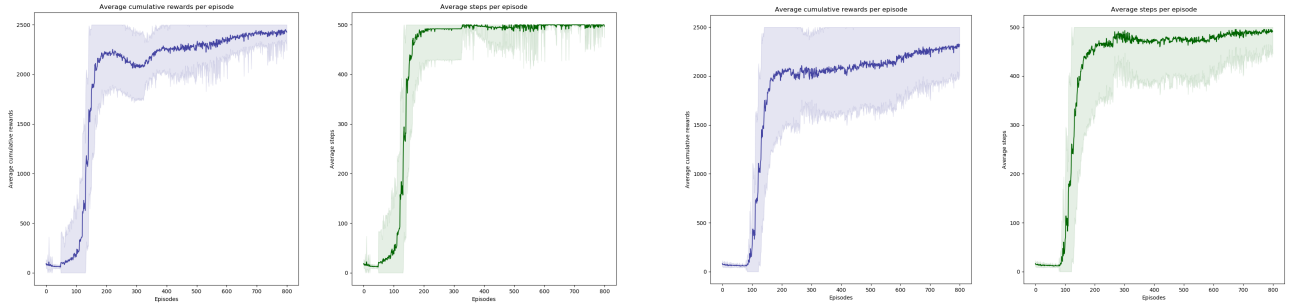
In this section, the Gym Cart-Pole environment (introduced in section 3.1) is customized. From now on, all episodes have a maximal length of 500 steps and the reward distribution is changed with the distribution given by the Figure 5.4. This new reward distribution encourages the agent to stay in position within the zone defined by the plateau in Figure 5.4. The problem becomes thus more difficult since it forces the Cart-Pole to respect the same constraints as before in addition to forcing it to stay in a smaller zone in position. The shape of this reward function will be discussed in more details in section 5.3. In addition, the initial position of the Cart-Pole is no longer randomly chosen between $[-0.05, 0.05]$ but in the zone corresponding to the plateau ($[-0.25, 0.25]$).

5.2.1 Comparison On A Deterministic Environment

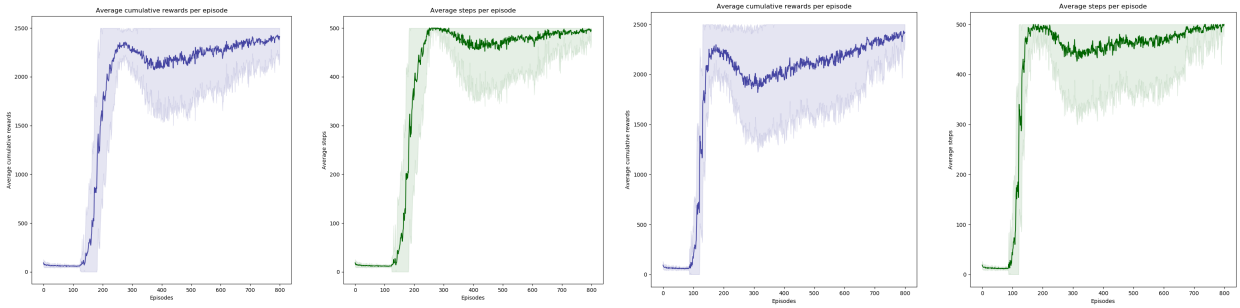
The Gym Cart-Pole environment is by default deterministic because no external disturbances are applied (noise, frictions, ...). In this section, training performance of Deep RL methods are compared in this perfect simulated deterministic environment with no external disturbance. As usual, hyperparameters for Deep RL methods were found with a Grid-Search.

5.2.1.1 DQN And Its Extensions

For DQN and its extensions, there exists an alternative to the ϵ-greedy strategy for exploration called NoisyNets [20] (see Appendix B.1.3). In practice when NoisyNets are combined with DQN, the learning speed is very different from one training to another. Sometimes DQN converges towards an optimal policy within 800 episodes, sometimes it does not. Whereas with an ϵ-greedy strategy the variance during training is lower. For this reason, an ϵ-greedy



(a) DQN (b) DDQN



(c) Duel-DDQN (d) Duel-DQN

Figure 5.5: Training performance of DQN and its extensions combined with order-1 Fourier Basis on the deterministic Cart-Pole task over 60 trainings (maximum episode length of 500 steps)

policy strategy was preferred. To speed up the training, a widely known trick is the use of an ϵ-decreasing exploration over steps. The training starts with a high $\epsilon_{\text{start}} \approx 1$ and ends to a small $\epsilon_{\text{end}} \approx 0$. In this way, the exploration is encouraged at the beginning of the training. The ϵ-decreasing exploration considered for the Cart-Pole task is given by:

$$\epsilon(t) = \max(\epsilon_{\text{end}}, \epsilon_{\text{start}} - \log(t+1)\epsilon_{\text{decay}}) \tag{5.1}$$

where $\epsilon_{\text{decay}} \in \mathbb{R}$ is the ϵ decay and t is the step.
Performance of DQN are better with the PER extension (Appendix B.1.2). However, it introduces two new hyper-parameters: α which determines how much prioritization is used and β which corresponds to the amount of importance-sampling correction. Authors suggest the use of a variable β over time [15]. Indeed, at the beginning of the training, it is recommended to have a high bias and a low variance to speed up the learning. For training, the bias is corrected by decreasing the bias (and then increasing β) to converge toward an optimal policy. A linear decrease of β was chosen:

$$\beta(t) = \min(1, t \times \beta_{\text{decay}} + \beta_{\text{start}}) \tag{5.2}$$

where $\beta_{\text{start}} \in [0, 1]$ is the β threshold start value and $\beta_{\text{decay}} \in \mathbb{R}$ is the β decay. Authors suggest $\beta_{\text{start}} = 0.4$.
All other hyperparameters chosen for DQN and its extensions can be found in Table C.4.

Figure 5.5 depicts training performance of DQN and its extensions on the deterministic version of the Cart-Pole environment. The average cumulative rewards per episode (blue curve) give us information about the ability of the agent to stay in the plateau. Whereas, the average

steps per episode (green curve) represent its ability to balance the pole. The pole is correctly balanced when the number of steps per episode is equal to 500 steps and the Cart stays in the plateau for the whole episode when the average cumulative rewards value per episode is equal to 2500.

It is obvious that DQN and its extensions learn and converge toward an optimal policy. However, DQN and its extensions do not reach exactly the maximum cumulative rewards on average. Besides, there is no guarantee after the training that the agent balances the pole correctly during 500 steps because the standard deviation is not equal to zero for the 800^{th} episode. Consequently, learning these methods within 800 episodes is not enough, they need more episodes. Besides, there is a short period (over 100 episodes) starting from the 200^{th} where methods unlearn. A wrong setting of the parameter β and the variation of β may explain this phenomenon. Maybe if β_{decay} is increased and thus the bias is decreased, the problem may be fixed.

In addition, Figure 5.5 shows that these methods do not simultaneously learn to stay in the plateau and to balance the pole, in other words the angle and position control is not simultaneous. Indeed the green curve and the blue curve are similar for all methods. Such leaning may seems to be counter-intuitive since if the agent fails to maintain the pole, the episode ends and therefore it no longer earns rewards. Balancing the pole should have an higher priority than the position control during the training but it is not the case in practice. These results suggest that this prioritization was not clear enough in the reward distribution.

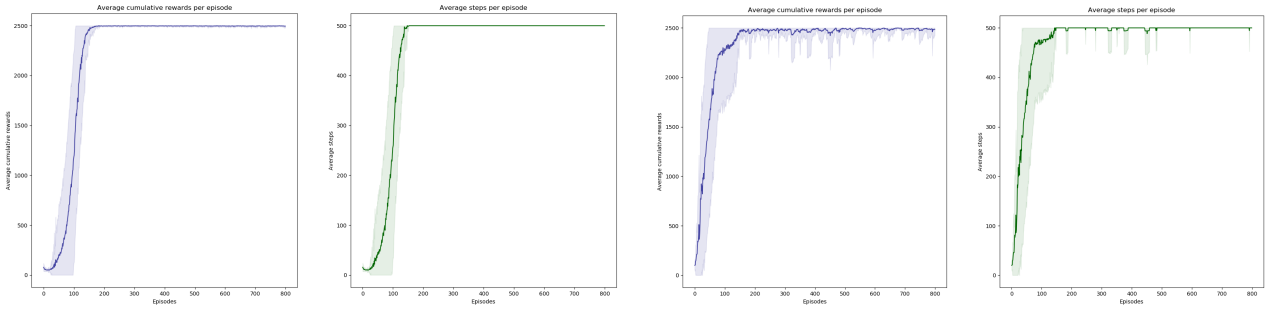
Finally, DQN extensions do not perform better than DQN, it is even worse. The use of DDQN appears to increase the variance because DDQN and Duel-DDQN have a higher standard deviation than DQN and Duel-DQN, respectively. Likewise, a duel architecture is no more efficient. Maybe another hyper-parameter set for the duel-architecture would improve performances.

5.2.1.2 Advantage Actor-Critic Methods

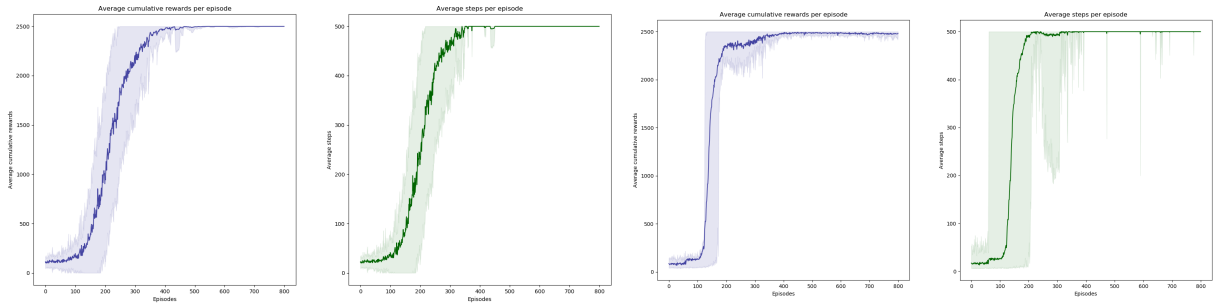
REINFORCE method was the first Policy-Gradient method implemented but results with the constant reward signal were not convincing compared to those of the DQN. For this reason, no experiences were done with the new reward distribution.

This subsection studies the training performance of the Advantage Actor-Critic methods A2C, PPO, ACKTR and SAC. All hyperparameters were found with a Grid-Search ans can be found in Appendix C.2. A value loss coefficient hyperparameter for critic is added for Advantage Actor-Critic methods. Since it is usual for Actor and Critic to share the same first layers in the neural architecture, it is common to use the same learning rate η to update both Actor and Critic layers. However, it was observed that having a slightly different learning rate can improve performance. For this reason, a value loss coefficient is introduced to modify the learning rate of the critic by multiplying it by the coefficient. Besides another hyperparameter called the trace-decay for the critic is added to speed up the learning. This trace-decay is used to estimate the "true" state value and is similar to the trace decay used by the TD(λ) method [6].

Figure 5.6 depicts training performance of A2C, PPO, ACKTR and SAC methods on the deterministic version of the Cart-Pole environment. It is clear that Advantage Actor-Critic methods learn faster than DQN methods. Moreover, the variance is lower for Advantage-Actor



(a) A2C (b) PPO



(c) ACKTR (d) SAC

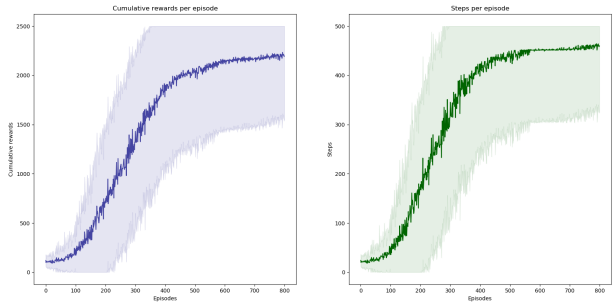
Figure 5.6: Training performance of Actor-Critic methods combined with order-1 Fourier Basis on the deterministic Cart-Pole task over 60 trainings (maximum episode length of 500 steps)

Critic methods. Recall a near-zero standard deviation over a long period indicates that the learning is nearly finished. At the end of the training, for all Actor-Critic methods, there is a zero standard deviation for the green curve suggesting that methods succeed in learning to respect the hard constraints (defined by the condition of terminations) of the Cart-Pole task. However, this is not the case for the blue curves meaning that the learning of the position control is not completely finished. As for DQN, Actor-Critic methods do not set an higher priority to balance the pole than to control the position because the blue and green curves are almost similar during the training.

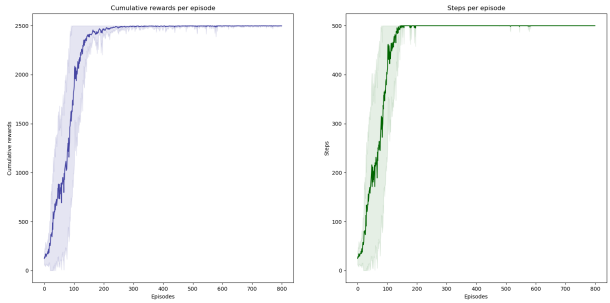
A2C is the fastest learning method with the fastest convergence and near-zero variance at the end of learning. ACKTR is the slowest method to converge at the beginning of learning, but it is one of the only methods to have a standard deviation close to zero at the end of learning. It is the opposite for PPO with a very fast convergence at the beginning of the training with a slightly larger standard deviation at the end of the training. Nevertheless, it seems that all methods learned to respect hard constraints at the end of the training for a finite horizon time. In other words they were able to learn what they were asked to learn.

5.2.2 Comparison On A Non-Deterministic Environment

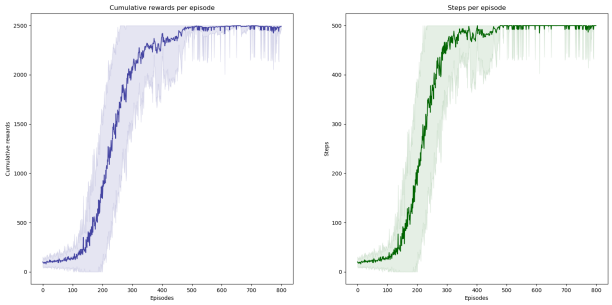
So far Deep RL methods were only applied on a deterministic version of the Cart-Pole environment with no external disturbance. However, in the real world the environment is not necessarily deterministic and external disturbances (noise, delays) can make the environment non-deterministic. For closing the simulation to real gap, it is important that the methods



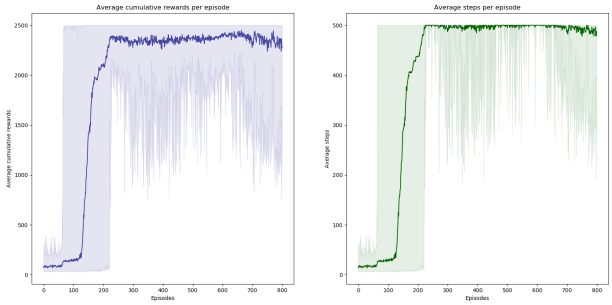
(a) A2C



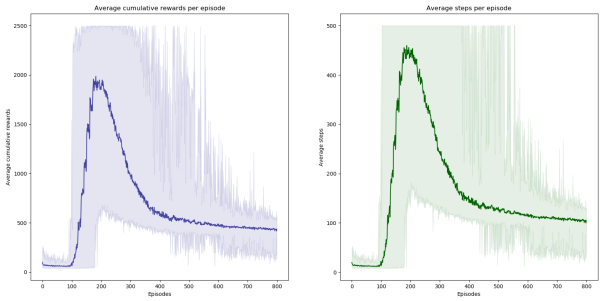
(b) PPO



(c) ACKTR



(d) SAC

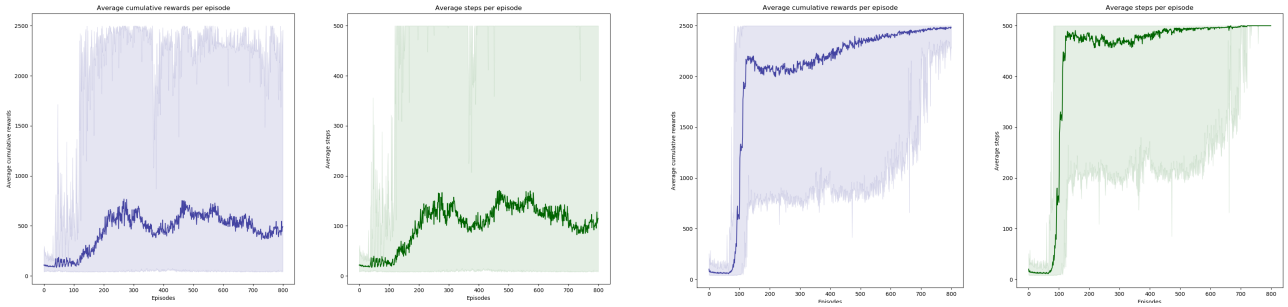


(e) Duel-DQN

Figure 5.7: Training performance of several Deep-RL methods combined with order-1 Fourier Basis on a non-deterministic version of the Cart-Pole environment over 60 trainings (maximum episode length of 500 steps)

are able to learn in non-deterministic environments. To have a more representative real-world environment, **Gaussian noise** is added to the observations. The environment becomes thus non-deterministic for the agent. In this subsection, only Gaussian noise of zero mean is considered as noise with standard deviation of 0.0025, 0.01, 0.013, 0.01 applied to the position, the velocity, the angle, and the angular velocity respectively. Note that standard deviations were chosen arbitrarily. The non-deterministic environment uses the reward distribution depicted in Figure 5.4. A Grid-Search was performed to find the best hyperparameters for each method. Almost the same hyperparameters as those used in the previous sub-section were found, except for the batch size. It seems that increasing the batch size improves performance when external perturbations are added.

Figure 5.7 depicts the behavior of Deep RL methods on the non-deterministic version of the Cart-Pole environment. Unlike the deterministic version, Duel-DQN does not converge toward



(a) With a delay (b) Without a delay

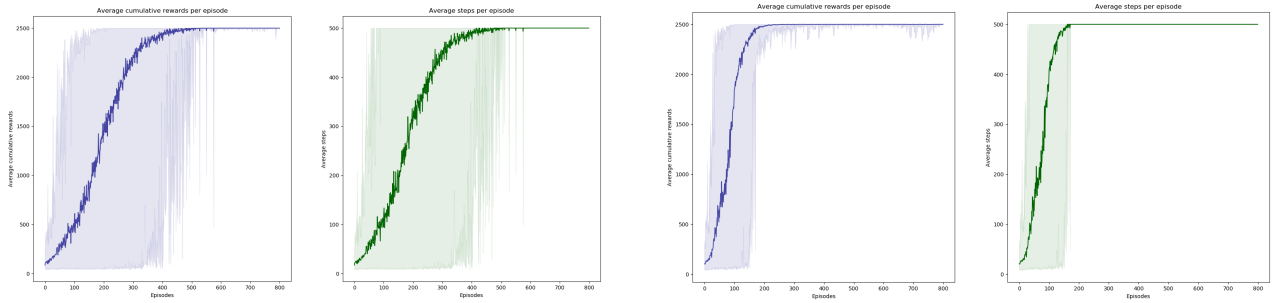
Figure 5.8: Training performance of Duel-DQN combined with order-1 Fourier Basis with/without delay over 60 trainings (maximum episode length of 500 steps)

an optimal policy during the training. At the beginning of the training, the method seems to learn then starting from the 200$^{\text{th}}$ it unlearns and converge towards a policy with poor performance. The same phenomenon of unlearning was observed on a deterministic environment but after a while the method was able to relearn and converge towards an optimal solution.
In contrast, Actor-Critic methods still converge towards an optimal policy. The standard deviation for the training is just higher on non-deterministic environment than on deterministic environment. It can be concluded that learning takes more time in non-deterministic environments than in deterministic environments to converge towards an optimal policy for Actor-Critic methods. PPO and ACKTR have training performances similar to those obtained in a deterministic environment, except that the standard deviation is slightly higher. Unfortunately, this is not the case for A2C. The addition of noise significantly slows down the learning of A2C with a higher standard deviation and a slower convergence on average. The fact that PPO and ACKTR outperform A2C is perhaps not so surprising since they were initially tested and developed for robotic applications [3, 17].

5.2.3 Deep RL Methods On A Delayed Environment

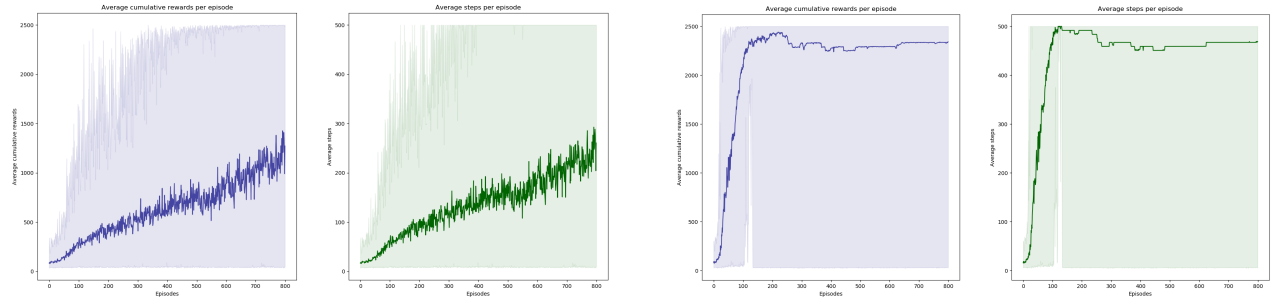
Noise is not the only external disturbance that can be found in the real world, the agent can also be subject to delays. For example, there may be a delay when the environment sends information to the agent and the agent processes it. Such delays are really common in robotics. It is therefore very useful to study the training performance of Deep RL methods on a delayed version of the Cart-Pole environment. The same reward distribution depicted in Figure 5.4 was on this new version of the Cart-Pole environment. In the Gym Cart-Pole environment, the state is measured every 20 ms and the information is sent directly to the agent. For this study, a delay of 60 ms is introduced between the time the state is measured and the time it is sent to the agent. Same hyperparameters found in the previous subsection and defined in Appendix C.1 are used.

Figure 5.8 describes the training performance of the Duel-DQN with and without the addition of delay. It is obvious that adding a delay affects the training of Duel-DQN. Duel-DQN is no longer able to converge towards an optimal policy in 800 episodes. Even worse, Duel-DQN seems to converge on a sub-optimal policy with poor performance and the variance is signifi-



(a) With a delay (b) Without a delay

Figure 5.9: Training performance of PPO combined with order-1 Fourier Basis with/without delay over 60 trainings (maximum episode length of 500 steps)

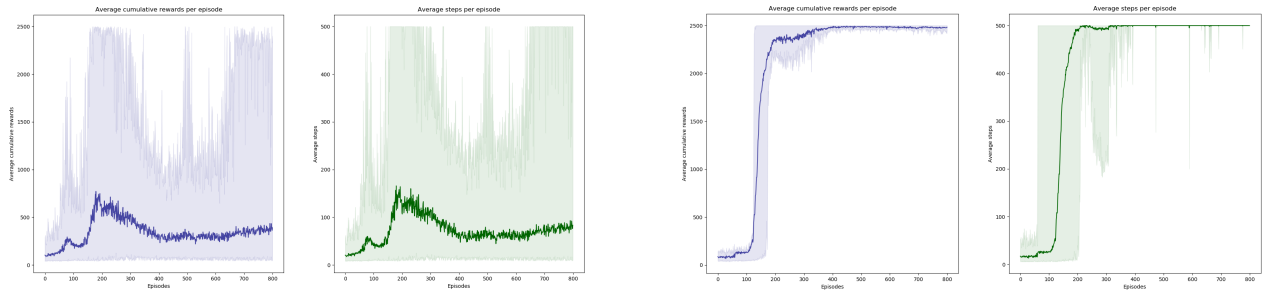


(a) With a delay (b) Without a delay

Figure 5.10: Training performance of PPO combined with order-1 Fourier Basis with/without delay over 60 trainings on a version of the Cart-Pole environment with a continuous action space (maximum episode length of 500 steps)

cant. Such high variance means that training is very different from one simulation to another and that it is very difficult to know the behavior of the method during the training.

The delay has less impact on PPO as shown in Figure 5.9. Indeed, although PPO takes longer to converge toward a policy, it still converges toward an optimal policy in less than 800 episodes. The convergence is almost twice longer with a delay. It can therefore be said that the PPO continues to learn effectively even though the training is much longer with the addition of noise. Until now in this subsection, the action space $\mathcal{A}$ was discrete but Actor-Critic methods like PPO can be adapted for continuous action space [6]. Therefore, PPO was also tested with the same hyperparameters on a Cart-Pole environment with a continuous action space. Results are shown in Figure 5.10. For PPO, with a continuous space of action learning is more difficult with a longer learning and a greater variance than in an environment with a discrete action space. However, even if, PPO does not converge towards an optimal policy within 800 episodes, it is clear that the method learns unlike Duel-DQN. Perhaps with a longer training, the method would converge towards an optimal policy. ACKTR behaves similarly to PPO with longer learning on a delayed environment than on the perfect environment. However, not all Actor-Critic methods react in the same way to delay, as shown in Figure 5.11 for SAC. Similar Duel-DQN, SAC converges towards a policy with poor performance when a delay is added.



(a) With a delay

(b) Without a delay

Figure 5.11: Training performance of SAC combined with order-1 Fourier Basis with/without delay over 60 trainings on a version of the Cart-Pole environment with a continuous action space (maximum episode length of 500 steps)

These experiments show that adding a delay degrades the learning of Deep RL methods more than noise. Indeed, some methods such as DQN or SAC are no longer able to converge towards an optimal policy. While for others, such as PPO or ACKTR, the learning takes much more time. Such poort results can be explained by a bad tuning of hyperpramaters, no special Grid-Search was performed on the delayed version of the Cart-Pole environment. The general behavior of Deep RL methods on delayed environments is not known although there exists some papers dealing with the case where rewards are delayed [21, 22].

5.3 Performance After Training

We saw in the previous sections that Deep RL methods learn on both deterministic and non-deterministic versions of the Cart-Pole environment, but it is important to ensure that the learned controller is effective. The objective of this section is to study the post-training performance of the methods on the Cart-Pole environment and to compare them with each other and with more conventional methods.

From now on, the training was made on finite horizon time but it would be nice after the training to have an agent that can be used for an infinite horizon time. It was observed that with a constant reward signal, the policy was unsafe for episodes longer than the training horizon. Indeed, for example, after training the agent on finite horizon time of 200 steps on the classic gym Cart-Pole environment, it was common to see the cart getting out of the rail after the 200 steps. There are RL methods for learning on continuous tasks, however most methods are developed for episodic tasks. To fix this problem, one idea is to encourage the agent during the training to return in the set of its initial states by changing the reward distribution. This property is called the Inductive Invariance property and will be discussed in more details in Chapter 6. For the Cart-Pole environment, it was decided to use the reward distribution shown in Figure 5.4 to incite the Cart-Pole to return in the set of its initial positions. The plateau that we will note $\mathcal{P}$ in the reward distribution corresponds to the set of the initial position of the Cart-Pole. Note that with this reward distribution, no penalty is assigned to the other state variables.

5.3.1 Metrics

In order to measure the ability of the agent to return to the set of its initial states, metrics must to be defined. The list below summarizes metrics used in this subsection to evaluate the post-training performance of the Deep RL methods:

- the **success rate** which indicates the percentage of complete episodes. An episode is complete if its duration corresponds to the predefined maximum number of steps (= 501 steps in this section). A rate equal to one guarantees the agent respects hard constraints for the training horizon time,
- the **average length of an episode** which is the average number of steps per episode. The maximum number of steps is 501 in this section,
- the **average number of outliers in position** which is the average number of steps over the last 150 steps of each complete episode where the Cart-Pole is not in the set of its initial positions $\mathcal{P} = [-0.25, 0.25]$,
- the **statistics on distances in position** which is the distance between the Cart's position and the set of its initial positions $\mathcal{P}$: $d_{\text{pos}}(x, \mathcal{P}) = \inf_{p \in \mathcal{P}}\{|x - p|\}$ where x is the position of the Cart. This distance is only computed over the last 150 steps of each complete episode. A distance greater than zero means that the step is an outlier in position and quantifies how far the outlier is from $\mathcal{P}$,
- the **average number of outliers in angle** which is the average number of steps over the last 150 steps of each complete episode where the Cart-Pole is not in the set of its initial angles $\mathcal{I} = [-2.86°, 2.86°]$,
- the **statistics on distances in angle** which is the distance between the Pole's angle and the set of its initial angles $\mathcal{I}$: $d_{\text{angle}}(x, \mathcal{I}) = \inf_{\theta' \in \mathcal{I}}\{|\theta - \theta'|\}$ where θ is the angle of the Pole. This distance is only computed over the last 150 steps of each complete episode. A distance greater than zero means that the step is an outlier in angle and quantifies how far the outlier is from $\mathcal{I}$,

5.3.2 Performance On A Deterministic Environment

In this sub-section we compare the Deep RL methods by comparing the ability of the controllers resulting from these methods to return to the set of initial states $\mathcal{P}$. Deep RL methods are compared with each other but also with a conventional method (PID). The PID used is in fact the combination of two PIDs: one that regulates the position of the cart and another that regulates the angle of the pole. The two outputs are added to give a single output. As the action space is discrete, the sign of the sum is studied to discretize which the actions to be taken. The coefficients were chosen manually.
The post-training performance of Deep RL methods were computed on 60 different controllers learned by these methods. Metrics were computed on the data of states of 800 episodes for each controller learned. In order to study the behavior of a controller, it is important to test it on as many different configurations as possible. For this reason, $\mathcal{P}$ is divided into 800 disjoint segments corresponding to the 800 episodes. For which segment, a position is randomly chosen

Methods	Sucess Rate	Average Length of Episode
PID	1.0	501
DQN	0.9922	499.46
DDQN	0.9599	492.20
Duel-DQN	0.9787	498.12
Duel-DDQN	0.9762	498.05
A2C	1.0	501
PPO	1.0	501
ACKTR	1.0	501

Table 5.1: Performance Metrics

Metrics	**PID**	**DQN**	**DDQN**	**Duel-DQN**	**Duel-DDQN**	**A2C**	**PPO**	**ACKTR**
Number Of Steps In $\mathcal{P}$	151	117,02	95,97	110,54	109,54	148,99	148.12	**151**
Average	0	2.06e-01	3.50e-01	2.50e-01	2.41e-01	4.50e-02	9.31-02	**0**
Median	0	1.13e-01	2.35e-01	1.62e-01	1.59e-01	3.12e-01	9.88e-02	**0**
Q1	0	4.33e-02	8.66e-02	5.39e-02	6.05e-02	3.18e-02	3.02e-02	**0**
Q3	0	3.01e-01	5.36e-01	3.54e-01	3.34e-01	5.73e-02	1.35e-01	**0**
Variance	0	9.98e-03	2.44e-02	2.10e-02	1.32e-02	4.32e-04	4.28e-03	**0**
Worst Distance	0	2.18	2.19	2.16	2.17	1.31e-01	4.65e-01	**0**

Table 5.2: Statistics on distances in position. Distances in position are computed for the last 150 steps of each episode. The worst distance corresponds to the maximum distance observed. For each episode, distances are averaged. Average, median, Q1, Q3 and variance are computed on these average distances.

to be the initial position of the Cart.

Table 5.1, Table 5.2, and Table 5.3 resume metrics computed for some Deep RL methods. We can see that the Actor-Critical Advantage methods succeed in the Cart-Pole task over the training time horizon with a success rate equal to 1 (Table 5.1). This is not true for the Action-Value methods even though DQN is very close to this rate. These results are consistent with what was observed during training. Since the Actor-Critical methods are faster for learning, it is normal that they perform better after training. Besides, it confirms that the learning to respect hard constraints was over for Actor-Critic methods.

In addition, Actor-Critic methods perform better than Action-Value methods to return into $\mathcal{P}$ over the training time horizon (Table 5.2). The number of outliers is significantly lower as well as the distance of its outliers from $\mathcal{P}$. The worst distance is almost 20 times lower with Advantage Actor-Critic methods than with Action-Value methods. ACKTR is the only method

Metrics	**PID**	**DQN**	**DDQN**	**Duel-DQN**	**Duel-DDQN**	**A2C**	**PPO**	**ACKTR**
Number Of Steps In $\mathcal{I}$	151	150,26	149,37	147,22	148,18	**151**	**151**	**151**
Average	0	1.03e-02	1.48e-02	1.14e-02	1.29e-02	**0**	**0**	**0**
Median	0	5.61e-03	9.12e-03	6.74e-03	7.59e-03	**0**	**0**	**0**
Q1	0	3.10e-03	3.36e-03	3.26e-03	3.00e-03	**0**	**0**	**0**
Q3	0	1.36e-02	2.23e-02	1.68e-02	1.88e-02	**0**	**0**	**0**
Variance	0	6.67e-05	1.35e-04	7.18e-05	9.26e-05	**0**	**0**	**0**
Worst Distance	0	1.84e-01	1.60e-01	1.61e-01	1.56e-01	**0**	**0**	**0**

Table 5.3: Statistics on distances in angle. Distances in angle are computed for the last 150 steps of each episode. The worst distance corresponds to the maximum distance observed. For each episode, distances are averaged. Average, median, Q1, Q3 and variance are computed on these average distances.

at the end of the training that provides similar performance to the PID controller even if A2C and PPO are close. Such results suggest that a training in 800 episodes is not enough to learn to return in $\mathcal{P}$. This is also likely the case for DQN and its extensions. For longer training, all methods would achieve the same performance as PID.

Same results are observed for the angle control, Advantage Actor-Critic methods perform better than action-value methods with no outliers observed (Table 5.3) over the training time horizon. Even if there are some outliers for DQN and its extensions, they are very close to $\mathcal{I}$. Nevertheless, these metrics on angle do not make sense because rewards are assigned according to the position of the cart and not according to the angle of the pole. The agent is not explicitly encouraged with rewards to return to $\mathcal{I}$, but it seems to return there naturally. It is possible than with the current reward distribution and the condition of termination, the agent may have an incentive to return to $\mathcal{I}$.

Not so surprisingly, these results are consistent with what was observed in section 5.2 during the training. Advantage-Actor Critic methods give better results than Action-Value methods after the training because they learn faster and the learning was almost finished. ACKTR is the only method close to the performance got with the PID controller. With a longer training, there is a good chance of for learned controller to achieve performance similar to the PID controller. There is a good chance that a controller learned with ACKTR can be used over an infinite horizon time. Even if no outliers are observed in angle and position, to verify the inductive invariance property it is necessary to test all the trajectories generated by ACKTR and calculate metrics for speed and angular velocity. In chapter 6, there is a discussion about how to efficiently measure the safety of a controller.

Methods	Sucess Rate	Average Length of Episode
PID	1.0	501
A2C	0,9165	461,08
PPO	**1.0**	**501**
ACKTR	0,99944	500,76

Table 5.4: Performance metrics on non-determinsitic Cart-Pole

Metrics	**PID**	**A2C**	**PPO**	**ACKTR**
Number Of Steps In $\mathcal{P}$	151	96,09	**150,46**	141,41
Average	0	1.4964e-01	**3.1537e-02**	1.8298e-01
Median	0	1.0951e-01	**1.5803e-02**	1.7781e-01
Q1	0	5.0546e-02	**5.8220e-03**	6.2138e-02
Q3	0	1.9734e-01	**4.0948e-02**	2.7826e-01
Variance	0	1.0984e-02	**6.7018e-04**	1.1417e-03
Worst Distance	0	2.0243	**4.9563e-01**	1.0624

Table 5.5: Statistics on distances in position on non-deterministic Cart-Pole. Distances in position are computed for the last 150 steps of each episode. The worst distance corresponds to the maximum distance observed. For each episode, distances are averaged. Average, median, Q1, Q3 and variance are computed on these average distances.

5.3.3 Performance On A Non-Deterministic Environment

The same strategy and the same metrics defined in the previous section were used for measuring the post-training performance of Actor-Critic methods on a non-deterministic version of the Cart-Pole environment introduced in Section 5.2.2. Results can be found in Table 5.4, 5.5 and Table 5.5. Unlike the deterministic version of the environment, most Actor-Critic methods do not respect hard constraints of the Cart-Pole environment (Table 5.4). PPO is the only method that generates controllers that still meet these strict constraints. Even if, the rate of ACKTR is very close to one, failures were observed. It even worse for A2C with very poor performance. These failures can be explained by the fact that training takes more time in a non-deterministic environment, as explained in section 5.2.2. PPO and ACKTR were better than A2C during the training and these results are refound after the training. Similar results are observed in Table 5.5 and Table 5.5. Unlike the deterministic version of the environment, no method achieves the performance obtained with PID. PPO has surprisingly better performance on a non-deterministic version of the environment than on the deterministic version of environment and it seems to be the most robust Deep RL method studied to external perturbations for the Cart-Pole environment. However, according to the learning curves in Figure 5.7, there is a good chance that with longer training, most of the Actor-Critic methods would perform as well as the PID controller.

Metrics	**PID**	**A2C**	**PPO**	**ACKTR**
Number Of Steps In $\mathcal{I}$	151	144,03	150,85	**150,99**
Average	0	7.3247e-03	4.4065e-03	**3.0075e-03**
Median	0	5.2064e-03	3.2649e-03	**1.8893e-03**
Q1	0	2.6471e-03	1.8853e-03	**8.4088e-04**
Q3	0	9.3344e-03	5.9479e-03	**3.8694e-03**
Variance	0	3.8314e-05	1.1547e-05	**1.7800e-06**
Worst Distance	0	1.3738e-01	5.1630e-02	**1.3592e-02**

Table 5.6: Statistics on distances in angle on non-deterministic Cart-Pole. Distances in angle are computed for the last 150 steps of each episode. The worst distance corresponds to the maximum distance observed. For each episode, distances are averaged. Average, median, Q1, Q3 and variance are computed on these average distances.

Chapter 6

Measurements Of Safety Requirements

Recent advances in Artificial Intelligence (AI) have contributed to a proliferation of autonomous agents such as vehicles, drones, and inspection robots. Autonomous agents must respect some minimal **performance** (rise time, learning time, overshoot...) as well as **safety requirements** based on state constraints. The **safety requirements** are essential for the proper functioning of the controller. Indeed, a violation of these requirements can lead to a deterioration of the system. Often safety requirements are expressed as constraints on state variables. For example, for the Cart-Pole task we do not want that the cart moves out of the rail, so constraints on its position are defined.

We define proper metrics measuring the safety requirements of a controller. These metrics give us a quantitative information to reject a controller as well as to compare controllers between them. In that way, we can favor the safest controller. Two type of measures can be distinguished :

- the **absolute metrics** : they have a formal interpretation over behaviors and they accurately depict a property that we want to check.
- the **ranking** : it is more suited to compare methods. The values only make sense if you have several of them to provide a ranking. Formally speaking, a measure used for ranking only needs to be a pre-order, it is not a metric in the mathematical sense.

The aim of this chapter is to define a set of metrics measuring the safety requirements (based on state constraints) of a controller. These metrics are computed from data generated by the controller. The controller can be either a conventional controller or learned by a RL method as long as it is **deterministic**. Besides, we consider throughout this chapter that the n state variables s_i of the system are **normalized**. Indeed, the safety requirements on each state variable must be treated equally.

6.1 Safety Of Dynamical System

6.1.1 Constraint Zone

This report considers the state space $\mathcal{S}$ of the system as a subspace of $\mathbb{R}^n$ (with n the number of state variables of the system) therefore a metric space can be easily defined. The **Chebyshev distance** ($\delta_{\text{Chebychev}}(s, s') = \max_i(|s_i - s'_i|)$ for $s, s' \in \mathcal{S}$) associated to the state space $\mathcal{S}$ defines for example a metric space $(\mathcal{S}, \delta_{\text{Chebychev}})$ of the state space.

The system has a set of given **forbidden or critical states** $\bar{\mathcal{P}} \subset \mathcal{S}$ that need to be avoided. These states deteriorate the system, may lead to its self-destruction or its malfunction. The **constraint zone** $\mathcal{P} := \mathcal{S} \setminus \bar{\mathcal{P}} \subseteq \mathcal{S}$ is the complementary of the set of forbidden states.

States from which the controller can ensure that the system never enters the forbidden states are called **safe**. The states that are not safe are called **unsafe**; from these, the controller may not be able to keep the system away from the forbidden states. A priori (i.e., without further analysis), we only know which states are forbidden, not which states are safe or unsafe. For example, we know the maximum allowed position of the cart, but we don't know the maximum speed just before that position for which the cart can still slow down in time.

6.1.2 Dynamical System

The behavior of the system is mathematically described by a **dynamical system** on a discrete time. A dynamical system [7] is a pair $(\mathcal{S}, h)$ where $\mathcal{S} = (\mathcal{S}, \delta)$ is an arbitrary metric space and h is a map from $\mathcal{S}$ into itself. Here, the metric space $\mathcal{S} = (\mathcal{S}, \delta_{\text{Chebychev}})$ is a metric space of the state space $\mathcal{S}$. h denotes the **behavior of the system** and gives the next state of the system from the current state. If s_t is the current state, the state at the time $t+1$ is given by $h(s_{t+1})$. Noise can be added into h. For example, we can define h as $h(s_t) = g(s_t) + W_t$ where s_t is the current state, g is a map from $\mathcal{S}$ into itself describing the behavior of the controller and $(W_t)_{t\geq 1} \overset{\text{i.i.d}}{\sim} \phi \in \mathcal{D}(Z)$ the sequence describing the noise given by the probability distribution of the noise ϕ applied at each time t.

6.1.3 Trajectory

By the ***n-th* iterate** of $s \in \mathcal{S}$ under h we mean $h^n(s)$. Conventionally, $h^0(s) = s$. A **trajectory** $\mathcal{T}$ of $s_0 \in \mathcal{S}_0$ is defined as the sequence $(h^t(s_0))_{t\geq 0}$ and s_0 is the **initial state** of the trajectory $\mathcal{T}$. The subset $\mathcal{S}_0 \subseteq \mathcal{P}$ denotes the set of all initial states of any trajectory generated by the controller. In the case of a deterministic environment for a deterministic controller and with no noise applied, h is deterministic. Then, there is just one trajectory generated by the controller for each initial state $s_0 \in \mathcal{S}_0$. Otherwise, we can have multiple trajectories different from each other starting from one of the initial states.

A controller is said **safe** if for any of trajectory $\mathcal{T}$ generated, $\mathcal{T} \subseteq \mathcal{P}$. In other words, no forbidden states can be reached from any initial state in $\mathcal{S}_0$ following the policy of the controller.

6.2 Directed Hausdorff Metric

A first idea to measure the safety requirements is to define a distance metric to measure how far a trajectory $\mathcal{T}$ generated by a controller is from the constraint zone $\mathcal{P}$.

The **directed Hausdorff distance** is a general dissimilarity measure for two sets of points in a metric space. It is a well known distance measure in the field of computational geometry and image processing. Given two sets of points $A \subseteq \mathbb{R}^n$ and $B \subseteq \mathbb{R}^n$, the directed Hausdorff distance $\overrightarrow{\mathrm{d}}(A, B)$ from A to B corresponds to the maximum distance from a point $a \in A$ to the closest point of B, where distance between points is measured by the metric $\delta(a, b)$ of the metric space $(\mathbb{R}^n, \delta)$

$$\overrightarrow{\mathrm{d}}(A, B) = \sup_{x \in A} \inf_{y \in B} \delta(x, y) \tag{6.1}$$

The directed Hausdorff distance is not a true distance since it is not symmetric; the reverse Hausdorff distance $\overrightarrow{\mathrm{d}}(B, A)$ is in general different from $\overrightarrow{\mathrm{d}}(A, B)$. However, the directed Hausdorff distance has the interesting property that $\overrightarrow{\mathrm{d}}(A, B) = 0$ if $A \subseteq B$.

Therefore, considering the metric space of the state space $(\mathcal{S}, \delta_{\text{Chebychev}})$, the directed Hausdorff distance $\overrightarrow{\mathrm{d}}(\mathcal{T}, \mathcal{P})$ measures how far is the trajectory from the constraint zone. If we have $\overrightarrow{\mathrm{d}}(\mathcal{T}, \mathcal{P}) = 0$ then the system evolves for the trajectory $\mathcal{T}$ inside the constraint zone $\mathcal{P}$. In this way, we are able know to determine and quantify safety requirements for a given trajectory. If for any trajectory $\mathcal{T}$ generated by the controller we have $\overrightarrow{\mathrm{d}}(\mathcal{T}, \mathcal{P}) = 0$ then the controller is safe. Besides, we know how far the trajectory is far from $\mathcal{P}$ when $\overrightarrow{\mathrm{d}}(\mathcal{T}, \mathcal{P}) > 0$.

Even if we have a way to measure the distance between a trajectory and the constraint zone, we do not know how far the trajectory is from the forbidden states set $\bar{\mathcal{P}}$ if $\overrightarrow{\mathrm{d}}(\mathcal{T}, \mathcal{P}) = 0$. Indeed, the controller may evolve close to $\bar{\mathcal{P}}$. Consequently, with this metric, we do not favor the controller furthest away from $\bar{\mathcal{P}}$. For this reason, we need a more accurate metric.

6.3 Quantitative Robustness Estimate Metric

A boolean answer to know if the safety requirements are respected provides only a partial information and can be augmented with quantitative information about the satisfaction to provide a better basis for decision making. Therefore, **Signal Temporal Logic** can be an interesting framework to determine a new metric measuring the safety requirements. **Signal Temporal Logic** (STL) is a convenient and powerful formalism for continuous and hybrid systems for computing the robustness degree in which a piecewise-continuous signal satisfies or violates an STL formula [23]. Appendix D.1 recall the STL formalism necessary to understand this section.

To define a metric measuring the safety requirements, we can consider the states variables of a trajectory as signals. Moreover, the safety requirements are expressed as state constraints thus the constraint zone is defined from constraints on state variables. A **constraint on a state variable** s_i is defined as $\psi_{i,j} : a_{i,j} < x_i \leq b_{i,j}$ where $a_{i,j} < b_{i,j}$ and $a_{i,j}, b_{i,j} \in \mathbb{R} \cup \{+\infty\} \cup \{-\infty\}$ are the constraints corresponding to $\psi_{i,j}$.

Let φ_i be the conjunction of the m constraints corresponding to all the constraints defined on the state variable s_i : $\varphi_i = \wedge_{j=1,\ldots,m}\psi_{i,j} = \psi_{i,1} \wedge \ldots \wedge \psi_{i,m}$.

Let w denote the execution trace of real-valued signals of the state variables of the system $\{x_1^w, \ldots, x_n^w\}$ of a trajectory $\mathcal{T}$ generated by the controller. The time domain D is here the length of the trajectory.

The assertions

$$w, t \models \wedge_{i=1,\ldots,n}\varphi_i$$

check if the the state $x = (x_1^w, \ldots, x_n^w)$ from the trajectory $\mathcal{T}$ at time t is in the constraint zone. From now on, we only consider **finite trajectories**. Therefore, if T is the length of the trajectory $\mathcal{T}$, the assertions :

$$w, t \models \wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i$$

check if the the the trajectory $\mathcal{T}$ is in the constraint zone.

The **robustness estimate metric** for checking the safety requirement is defined as :

$$\begin{aligned}
\rho(\wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i, w, 0) &= \min_i \inf_{t\in[0,T]} \rho(\varphi_i, w, t) \\
&= \min_i \inf_{t\in[0,T]} \rho(\wedge_{j=1,\ldots,m}\psi_{i,j}, w, t) \\
&= \min_i \inf_{t\in[0,T]} \min\{\rho(\psi_{i,j}, w, t) | i \in [1,n], j \in [1,m]\}
\end{aligned}$$

If $\rho(\wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i, w, 0) > 0$ then the the trajectory $\mathcal{T}$ is in the constraint zone. Its magnitude gives information about how far the state variable x is far from $\bar{\mathcal{P}}$. $\rho(\wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i, w, 0)$ is a metric to measure the safety requirements based on state constraints. This metric evaluates for a given trajectory the furthest point from the constraint zone $\mathcal{P}$ if $\rho(\wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i, w, 0) < 0$ or the closer one from $\bar{\mathcal{P}}$ if $\rho(\wedge_{i=1,\ldots,n}\Box_{[0,T]}\varphi_i, w, 0) > 0$. The robustness estimate metric actually measures the worst constraints violation on the trajectory $\mathcal{T}$. If we have a sequence of m finite trajectories $(\mathcal{T}_i)_{i=1,\ldots,m}$ we take the minimal robustness metric.

6.4 Metrics Based On A Probabilistic Approach

Until now, only distances were used to define metrics. In this section, we will consider a probabilistic approach to define metrics for the safety requirements.

6.4.1 A Probabilistic Approach Of The Safety Requirements

We can consider at each time step t the **cumulative distribution** (Appendix D.3.1) F_i^t associated to the state variable x_i ($i \in [1,n]$). $F_i^t(x)$ represents the probability for the state variable x_i to be below a quantity x at the time step t :

$$F_i^t(x) = Pr(x_i(t) \leq x)$$

Given a constraint $\psi_{i,j}^t$ on the state variable x_i at time step t (see the definition of $\psi_{i,j}$ in the section 6.3), the probability :

$$\begin{aligned} 1 - Pr(\psi_{i,j}^t) &= 1 - (F_i^t(b_{i,j}) - F_i^t(a_{i,j})) \\ &= 1 - F_i^t(b_{i,j}) + F_i^t(a_{i,j}) \end{aligned}$$

represents the the probability that the state variable x_i does not respect the constraint $\psi_{i,j}^t$ at the time step t where $a_{i,j}, b_{i,j}$ are the constraints associated to $\psi_{i,j}^t$.

The bijection const is defined as $\text{const}(\psi_{i,j}) = [a_{i,j}, b_{i,j}]$ where $a_{i,j}, b_{i,j}$ are the constraints associated to $\psi_{i,j}^t$ (see the definition of $\psi_{i,j}$ in the section 6.3). Two constraints $\psi_{i,j}^t$ and $\psi_{i,k}^t$ are said **disjoint** if $\text{const}(\psi_{i,j}) \cap \text{const}(\psi_{i,k}) = \emptyset$

Let φ_i be the set of all disjoint constraints $\psi_{i,j}$ on the state variable x_i. All constraints on the state variable can be expressed as a set of all disjoint constraints. Indeed, if two constraints $\psi_{i,j}^t$ and $\psi_{i,k}^t$ are not disjoint then you build the constraint $\psi_{i,l}^t$ such that $\text{const}(\psi_{i,l}) = \text{const}(\psi_{i,j}) \cup \text{const}(\psi_{i,k})$. We have then:

$$\begin{aligned} Pr(\bar{\varphi}_i^t) &= 1 - Pr(\varphi_i^t) = 1 - Pr(\cup_{k=1}^p \psi_{i,k}^t) \\ &= 1 - \sum_{k=1}^{p} \left(F_i^t(b_{i,k}) - F_i^t(a_{i,k})\right) \end{aligned}$$

where $Pr(\bar{\varphi}_i^t)$ represents the probability that the state variable x_i does not respect its constraints at time step t and p is the number of constraints defined in $\varphi_i(t)$. And

$$\begin{aligned} Pr(\bar{\varphi}_i^{0:T}) = Pr(\cup_{t=0}^T \bar{\varphi}_i^t) &= \sum_{t=0}^{T} (1 - Pr(\varphi_i^t)) \\ &= \sum_{t=0}^{T} \left(1 - \sum_{k=1}^{p} \left(F_i^t(b_{i,k}) - F_i^t(a_{i,k})\right)\right) \end{aligned}$$

is the probability that the state variable x_i does not respect the constraints on the the period $[0, T]$ with the assumption that $\forall t, t' \in [0, T]$ with $t \neq t'$, the events $\bar{\varphi}_i^t$ and $\bar{\varphi}_i^{t'}$ are independent. For the time horizon T, the approximation of the probability $Pr(\bar{\varphi}_i^{0:T})$ defines metrics to measure the safety requirements based on states constraints. Contrary to the robustness estimate metrics which was only focused on the worst constraint violation, the probabilistic approach considers equally all the constraints violations. This metric is not better than the robustness estimate metrics but gives other information. For example, consider the case where we have two controllers : a controller C_1 with few constraint violations and another controller C_2 with more violations. Besides, we assume that the controller C_1 has a worst constraint violation than the controller C_2. Therefore, the robustness estimate metric says that the controller C_1 is worst than the controller C_2 but the probabilistic metric may say the opposite. Thus, which metric has right ? The answer is that it depends on the tasks. There are no universal metrics giving always the best controller for any task.

6.4.2 Empirical Metric Based On The Beta Distribution

We assume that we have collected from a controller a set of m trajectories $\{\mathcal{T}_i\}_{i=1,\dots,m}$. The set $\mathcal{D}_i^t = \{x_i(t) \in \mathcal{T}_j | j \in [1,m]\}$ is a set of m i.i.d samples given by the cumulative distribution F_i^t of the state variable x_i at time step t. With the **Beta distribution generator method** [24] we are able to compute the confidence intervals $F_{i,l}^t$ and $F_{i,u}^t$ at a confidence level c of the cumulative distribution F_i^t (see Appendix D.3.1 for further explanations). Given a constraint $\psi_{i,k}$ where $\text{const}(\psi_{i,k}) = [a_{i,k}, b_{i,k}]$, we have:

$$F_{i,l}^t(b_{i,k}) - F_{i,u}^t(a_{i,k}) \leq F_i^t(b_{i,k}) - F_i^t(a_{i,k}) \leq F_{i,u}^t(b_{i,k}) - F_{i,l}^t(a_{i,k})$$

We can then set an interval with a confidence level c for $Pr(\bar{\varphi}_i^{0:T})$ where T is the length of all trajectories:

$$\underbrace{\sum_{t=0}^{T}\left(1 - \sum_{k=1}^{p}\left(F_{i,l}^t(b_{i,k}) - F_{i,u}^t(a_{i,k})\right)\right)}_{=A_i} \leq Pr(\bar{\varphi}_i^{0:T}) \leq \underbrace{\sum_{t=0}^{T}\left(1 - \sum_{k=1}^{p}\left(F_{i,u}^t(b_{i,k}) - F_{i,l}^t(a_{i,k})\right)\right)}_{=B_i}$$

Besides with the **Empirical Distribution Function** [25] (Appendix D.2), we can have an approximate $\hat{F}_i^t$ of the cumulative distribution function F_i^t and thus an approximate of $Pr(\bar{\varphi}_i^{0:T})$:

$$\mathbb{P}(\bar{\varphi}_i^{0:T}) \approx \underbrace{\sum_{t=0}^{T}\left(1 - \sum_{k=1}^{p}\left(\hat{F}_i^t(b_{i,k}) - \hat{F}_i^t(a_{i,k})\right)\right)}_{=\hat{P}_i}$$

The interval $[A_i, B_i]$ and $\hat{P}_i$ give us information on the probability that the controller does not respect the constraints on the state variable x_i for a time horizon T. Both of them are relevant. Indeed, if for example we have two controller for which we have $[A_i^1, B_i^1]$ and $\hat{P}_i^1$ and $[A_i^2, B_i^2]$ and $\hat{P}_i^2$, respectively, such that:

$$[A_i^2, B_i^2] \subseteq [A_i^1, B_i^1] \tag{6.2}$$

and

$$\hat{P}_i^1 \leq \hat{P}_i^2$$

It is not so easy to know which controller is the best. It depends on the task. For this reason, sometimes it can be more interesting to keep either the interval $[A_i, B_i]$ or $\hat{P}_i$

6.5 On Infinite Horizon Time

Until now, we only use metrics on a finite horizon time T to check if a controller respects the safety requirements. However, most tasks have an infinite horizon time and we do not know its behavior after this time horizon T. Thereby, with the current metrics for tasks on a finite horizon time T we have no guarantee on the safety of the controller.
Moreover, even if we have a finite horizon time T, we have to check all the trajectories generated by the controller to be sure that the controller is safe. The number of trajectories generated by the controller may be infinite if $\mathcal{S}_0$ is infinite. In practice, it is not possible. For this reason, we need to properly define a property that the controller has to check to make sure it is safe.

$(\mathcal{S}, h)$ is the dynamical system given by the controller. A deterministic controller means that there exists a bijection between the initial states set $\mathcal{S}_0$ and the set Σ of trajectories generated by the controller. If for every $x_0 \in \mathcal{S}_0$, there exists t' such that $h^{t'}(x_0) \in \mathcal{S}_0$ and that the finite sequence $\mathcal{T}^{0:t'} = \{x_0, h^1(x_0), \ldots, h^{t'}(x_0)\} \subseteq \mathcal{P}$ then the controller is safe. It is the **Inductive Invariance**. Every controller which satisfies this property is safe but every safe controller does not satisfy this property. However, in that way we can find some controllers which are safe for an infinite horizon time. The Inductive Invariance can be checked with the metrics previously defined between the trajectory generated by the controller and the set $\mathcal{S}_0$.

In practice, it is very difficult to determine exactly each element of the subset $\mathcal{S}_0$. Indeed, when the system reboots, we are not exactly sure that the systems reboots exactly at a specific initial state x_0 but more in a neighboring state of x_0. Therefore, $\mathcal{S}_0$ is more like an infinite subset defined by constraints than a finite subset. As $\mathcal{S}_0$ is infinite, we have an infinite number of trajectories and consequently we can not in practice check the Inductive Invariance. Thereby, we can just test for as many different trajectories as possible the conditions of the Inductive Invariance with the defined metrics to compare methods or to get an idea of the controller behavior. But we are no longer able to say with certainty that the controller is safe.

Chapter 7

Conclusions

The first part of this report attempts to introduce the basics of Reinforcement Learning and Deep Reinforcement Learning in a compact yet simple form for anyone who would like to get started in Reinforcement Learning.
The experiments highlight the importance of the choice of features encoding for Deep RL methods and more particularly reveal the efficiency of Fourier Basis encoding. Indeed, with Fourier Basis, the Deep RL methods learn faster, converge to better policies and are easier to tune.
A comparison of the training performance of several Deep RL methods on the Cart-Pole environment shows that not all Deep RL methods react in the same way, in particular in a non-deterministic environment with external disturbances (noise, delays). For some methods such as DQN, the learning becomes impossible with the addition of external disturbances. In contrast, most Actor-Critic methods are more robust but the training requires generally more time and the variance is higher. Surprisingly, the Proximal Policy Optimization (PPO) is the exception: it has the same training performance in a non-deterministic environment with even a lower variance than with a deterministic version of the environment.
Basic metrics were introduced to determine the safety of Deep RL methods after the training. It was observed that with a constant reward signal, the policy was unsafe for episodes longer than the training horizon. By changing the distribution of rewards to encourage the agent to respect the Inductive Invariance (its ability to return to its set of initial states), it is possible to improve the safety over an infinite horizon time even if the training was made on an episodic task.
Finally, this report defines the safety of a given controller as its ability to stay outside of forbidden states given by a set of constraints. Safe states are states from which the controller can ensure that the system never enters forbidden states deteriorating the system. Three types of metrics are suggested to correctly quantify the safety of a controller: a distance to the forbidden states, a quantitative interpretation of the Signal Temporal Logic and an approximation of the confidence interval for the probability distribution.

For future work, the next steps will be:

- to test Fourier Basis encoding on more environments to prove its efficiency,
- to further compare Deep RL methods with each other and with conventional methods on simulated experiences and real-world experiences,

- to study in more detail the effect of external disturbances such as noise on Deep RL methods (more environments) and try to understand why some fail,
- to study the choice of different reward distributions on the safety of Deep RL methods,

It would be equally interesting to try to define new methods that impose constraints (by using a pattern close to the TRPO method) during the learning in order to guarantee the safety of the controller after the training or methods robust to external disturbances using unsupervised learning on their replay buffer.

All the implementations made during this project are available at `https://github.com/DavidBrellmann/DeepRL`.

.

Appendix A

Appendix To Introduction To Reinforcement Learning

Fundamental concepts useful for the definition of Markov Decision Process are defined in this appendix. The following definitions are taken from the Stachurski's Book [7].

A.1 Measurable Space

Let $\mathcal{S}$ be any nonempty set. A family of sets $\mathscr{S} \subseteq \mathfrak{B}(S)$ ($\mathfrak{B}(S)$ denotes the set of all subsets of $\mathcal{S}$) is called a **σ-algebra** if:

- $\mathcal{S} \in \mathscr{S}$
- $A \in \mathscr{S}$ implies $A^c \in \mathscr{S}$
- if $(A_n)_{n \geq 1}$ is a sequence with A_n in $\mathscr{S}$ for all n, then $\cup_n A_n \in \mathcal{S}$

The pair $(\mathcal{S}, \mathscr{S})$ is called a **measurable space** and elements of $\mathscr{S}$ are called **measurable sets**.

Let $\mathscr{S}$ be any metric space. The **Borel sets** on $\mathcal{S}$ denoted $\mathscr{B}(\mathcal{S})$ is the smallest σ-algebra on $\mathcal{S}$ that contains $\mathscr{O}$, the open subsets of $\mathcal{S}$. The smallest σ-algebra on $\mathcal{S}$ that contains $\mathscr{O}$ is the intersection of all σ-algebra on $\mathcal{S}$ that contains $\mathscr{O}$.

A.2 Probability Measure

For a given measurable set $(\mathcal{S}, \mathscr{S})$, a **probability measure** μ is a function from $\mathscr{S}$ to $[0, 1]$ such that:

- $\mu(\emptyset) = 0$
- μ is countably additive: if $(A_n) \in \mathscr{S}$ is disjoint then $\mu(\cup_n A_n) = \sum_n \mu(A_n)$

- $\mu(S) = 1$

The triple $(\mathcal{S}, \mathscr{S}, \mu)$ is called a **probability space**.

A.3 Stochastic Process

Let $(S, \mathscr{S})$ be a measurable space. An $\mathcal{S}$-valued **stochastic process** is a tuple:

$$(\Omega, \mathscr{F}, \mathbb{P}, (X_t)_{t \in \mathbb{T}})$$

where $(\Omega, \mathscr{F}, \mathbb{P})$ is a probability space, $\mathbb{T}$ is an index set such as $\mathbb{N}$ or $\mathbb{Z}$ and X_t is an $\mathcal{S}$-valued random variable on $(\Omega, \mathscr{F}, \mathbb{P})$ for all $t \in \mathbb{T}$

The idea is that at the beginning, a state x_0 is selected from the set Ω according to the probability law $\mathbb{P}$ and then $X_t(x)$ reports the time t outcome for the variable of interest as a function of that realization.

A.4 Stochastic Kernel

Let $\mathcal{S}$ be a Borel subset of $\mathbb{R}^n$. A **stochastic kernel** on $\mathcal{S}$ is a family of probability measures.

$$P(x, dy) \in \mathscr{P}(\mathcal{S}) \ (x \in \mathcal{S})$$

with $\mathscr{P}(\mathcal{S})$ the set of all probability measures on $(\mathcal{S}, \mathscr{B}(\mathcal{S}))$.

Each finite kernel p on the finite set $\mathcal{S}$ defines a general kernel P on $\mathcal{S}$ by

$$P(x, B) = \sum_{y \in B} p(x, y) \ (x \in \mathcal{S}, B \subset \mathcal{S})$$

Each density kernel p on a Borel set $\mathcal{S} \subset \mathbb{R}^n$ defines a general kernel P on $\mathcal{S}$ by

$$P(x, B) = \int_B p(x, y) dy \ (x \in \mathcal{S}, B \in \mathscr{B}(\mathcal{S}))$$

Appendix B

Appendix To Deep Reinforcement Learning

B.1 Extensions To DQN

This appendix proposes a selection of the most popular extensions of DQN.

B.1.1 Double DQN

The max operator in the target of the DQN results in the propagation of over-estimations with a positive bias toward the $\widehat{q}$ estimations. For further details or explanations to understand this positive bias, see [6], [26].

Double Deep Q-learning Network (DDQN) [27], addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation.
Indeed, one way to avoid maximization bias is to divide the plays into two sets and use them to determine two independent estimates $\widehat{q}_1(s, a, \mathbf{w})$ and $\widehat{q}_2(s, a, \mathbf{w}^-)$ of the true action-value estimate $Q_\pi(s, a)$. One estimate is used to determine the maximization action $a^* = \arg\max_{a \in \Gamma(s)} \widehat{q}_1(s, a, \mathbf{w})$ and the other one, $\widehat{q}_2(s, a, \mathbf{w}^-)$, to provide its action-value function. The process can be repeated inverting the two estimates.
There are already two networks, the target network and the online network, predicting Q_π. The decoupling is made thus by the use of the target network and the network for predictions. From the DQN loss 4.1, the loss for the DDQN is :

$$\mathcal{L}(\mathbf{w}_t) = \mathbb{E}_\pi \left[(R(s_t, a_t) + \gamma \widehat{q}(s_{t+1}, \arg \max_{a' \in \Gamma(s_{t+1})} \widehat{q}(s_{t+1}, a', \mathbf{w}_t)), \mathbf{w}_t^-) - \widehat{q}(s_t, a_t, \mathbf{w}_t))^2 \right] \quad \text{(B.1)}$$

The convergence may be twice faster with this minor change.

B.1.2 Prioritized Experience Replay

In classic DQN, transitions are uniformly sampled from the replay buffer but replaying all transitions with equal probability is suboptimal. Ideally, we want to sample more frequently those transitions from which there is much to learn.

Prioritized Experience Replay [15] (PER) solves this problem considering that some experiences may be more interesting than others for our training, but might occur less frequently. Indeed, samples with a greater **TD error** improve the critic faster so in PER they have a higher probability of being selected.
For the DQN method, the **TD error** δ_t defined as:

$$\delta_t = R(s_t, a_t) + \gamma \max_{a \in \Gamma(s)} \widehat{q}(s_{t+1}, a, \mathbf{w}_t^-) - \widehat{q}(s_t, a_t, \mathbf{w}_t) \tag{B.2}$$

Two ways of getting priorities may be considered:

- a **proportional prioritization**: $p_i = |\delta_i| + \epsilon$, where ϵ is a small constant ensuring that the sample has some non-zero probability of being drawn.
- a **rank based prioritization**: $p_t = \frac{1}{\text{rank}(t)}$ which sorts the items according to $|\delta_t|$ to get the rank.

Both distributions are monotonic in δ_t, but the latter is likely to be more robust, as it is insensitive to outliers. During exploration, p_i terms are not known for brand-new samples because those have not been evaluated with the networks to get a TD error term. For this reason, PER initializes p_i according to the maximum priority and then favoring those terms during sampling later.

From priority p_i, the probability of sampling transition i is:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{B.3}$$

with $\alpha \geq 0$. The α term determines how much prioritization is used (if $\alpha \to 0$ we are getting close to the uniform case, and if $\alpha \to 1$ there is full prioritization).

The Buffer size N can be quite large and authors suggest a special data structure, the **sum-tree** [15], to reduce the time complexity to find samples.

In the loss 4.1, the samples $(s_t, a_t, R(s_t, a_t), s_{t+1})$ are drawn by the state distribution $\mu \in \mathscr{P}(\mathcal{S})$ corresponding to the policy π. With the classic Replay Buffer combined with uniformly sampling, samples are still drawn from μ. However, PER introduces a bias because samples are not drawn anymore from μ because some samples are drawn more frequently than with the state distribution μ.
This bias can be corrected by using **importance-sampling weights**:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta \tag{B.4}$$

where N is the size of the PER and β controls how much prioritization is applied.
The importance-sampling weights compensate for the non-uniform probability $P(i)$ if $\beta = 1$.
Each minibatch is further scaled such that $\max_i w_i = 1$ for stability reasons.
The weight w_i is proportional to the the probability sample $P(i)$: if $P(i) \to 1$ the weight gets smaller, with an extreme down-weighting of the sample's impact. Otherwise, if $P(i) \to 0$, the weight gets larger. If $P(i) = \frac{1}{N}, \forall i$, there is then uniform sampling with weights equal to one.

An example of Pseudo-Code for DDQN combined with PER is given by Algorithm 6.

Algorithm 6 DDQN with PER (taken from [15])

Require: Minibatch of k elements, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize :
Initialize the Replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
loop
Observe s_0
With probability ϵ select a random action $a_0 \in \Gamma(s_0)$ otherwise select $a_0 = \arg\max_a \widehat{q}(\phi(s_0), a, \mathbf{w})$
for each step of episode $t = 1, \cdots, T$ **do**
Observe $s_t, r_t = R(s_t, a_t), \gamma_t$
Store transition $(s_{t-1}, a_{t-1}, r_t, \gamma_t, s_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
if $t \equiv 0 \mod K$ **then**
for $j = 1$ to k **do**
Sample transition $j \sim P(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$
Compute importance-sampling weight $w_j = (N.P(j))^{-\beta} / \max_i w_i$
Compute TD-error $\delta_j = r_j + \gamma_j \widehat{q}(s_j, \arg\max_{a \in \Gamma(s_j)} \widehat{q}(s_j, a, \mathbf{w}), \mathbf{w}^-) - \widehat{q}(s_{j-1}, a_{j-1}, \mathbf{w})$
Update transition priority $p_j \leftarrow |\delta_j|$
Accumulate weight-change $\Delta \leftarrow \Delta + w_j.\delta_j.\nabla_{\mathbf{w}} \widehat{q}(s_{j-1}, a_{j-1}, \mathbf{w})$
end for
Update weights $\mathbf{w} \leftarrow \mathbf{w} + \eta.\Delta$, reset $\Delta = 0$
From time to time copy weights into target network $\mathbf{w}^- \leftarrow \mathbf{w}$
end if
With probability ϵ select a random action $a_t \in \Gamma(s_t)$ otherwise select $a_t = \arg\max_a \widehat{q}(\phi(s_t), a, \mathbf{w})$
end for
end loop

B.1.3 NoisyNets

Classic DQN uses an ϵ-greedy strategy to select actions. The limitations of exploring with an ϵ-greedy policy may be clear in environments where many actions must be executed to collect the first reward.

NoisyNets [20] proposes another strategy. Methods treating with NoisyNets use a greedy

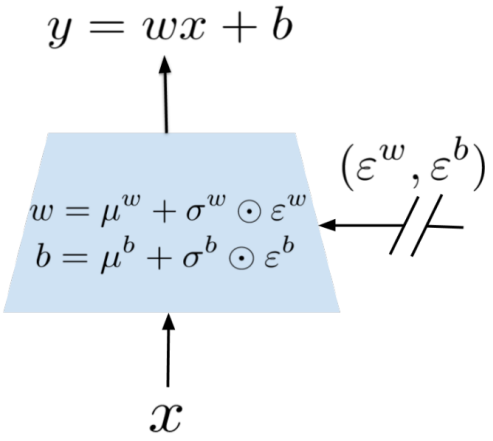


Figure B.1: Example of Noisy Layer (taken from [20])

strategy to select actions from the the estimate $\widehat{q}$. To maintain exploration, trainable parameterized noise are added to the last fully-connected layer. Adding such noise to a deep network is equivalent or better than using an ϵ-greedy strategy.

Let be a linear layer of the neural network with p inputs and q outputs, represented by :

$$\mathbf{y} = \mathbf{wx}+\mathbf{b} \tag{B.5}$$

where $\mathbf{x} \in \mathbb{R}^p$ is the layer input, $\mathbf{w} \in \mathbb{R}^{q \times p}$ the weight matrix and $\mathbf{b} \in \mathbb{R}^q$ the bias.
The corresponding **noisy linear layer** is defined as :

$$\mathbf{y} = (\boldsymbol{\mu}^w + \boldsymbol{\sigma}^w \odot \epsilon^w)\mathbf{x} + (\boldsymbol{\mu}^b + \boldsymbol{\sigma}^b \odot \epsilon^b) \tag{B.6}$$

where $\epsilon^b \in \mathbb{R}^q$ and $\epsilon^w \in \mathbb{R}^{q \times p}$ are noise random variables, $\boldsymbol{\mu}^w \in \mathbb{R}^{q \times p}, \boldsymbol{\mu}^b \in \mathbb{R}^q, \boldsymbol{\sigma}^w \in \mathbb{R}^{q \times p}, \boldsymbol{\sigma}^b \in \mathbb{R}^q$ are learnable and $\odot$ denotes the element-wise product. Such noisy layers are depicted in the Figure B.1.

Authors propose two noise distributions for linear layers in a noisy network :

- **Independent Gaussian noise**, which uses an Independent Gaussian noise entry per weight
- **Factorised Gaussian noise**, which uses an independent noise per each output and another independent noise per each input. By factorising the entry $\epsilon^w_{i,j}$ of the matrix ϵ^w, p unit Gaussian variables ϵ_i and q unit Gaussian variables ϵ_j for noise of the outputs can be used. Each $\epsilon^w_{i,j}$ and ϵ^b_j (corresponding to the entry of the matrix ϵ^b) can be defined as :

$$\epsilon^w_{i,j} = f(\epsilon_i)f(\epsilon_j) \tag{B.7}$$
$$\epsilon^b_j = f(\epsilon_j) \tag{B.8}$$

  where f is a real-valued function. Authors used $f(x) = \text{sgn}(x)\sqrt{|x|}$.

Factorised Gaussian noise is more interesting because there is less random numbers generation and thus the computation time is better.

Over time, the network can learn to ignore the noise, but will do so at different rates in differ-

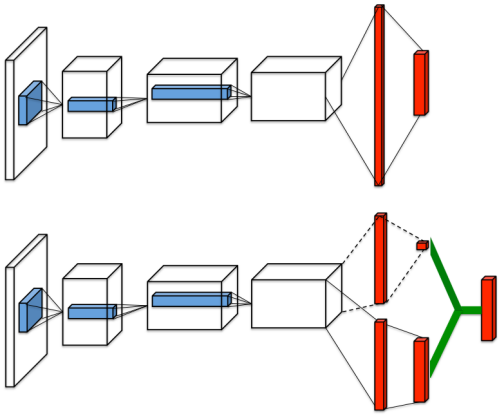


Figure B.2: Single Stream Q-Network (top) and dueling Q-network (bottom). First layers are convolutionals as the original DQN) (taken from [28])

ent parts of the state space, allowing state-conditional exploration with a form of self-annealing.

It is strongly advised to use different noise for the target network and the online network. According to the authors, the convergence may be better and faster with NoisyNets.

B.1.4 Duel DQN

This section proposes a new neural network architecture for DQN methods : the **dueling architecture** [28].

As action-value method, DQN learns the action-value function Q_π. By definition, the action-value function $Q_\pi(s_t, a_t)$ represents how it is good to select an action a_t in a particular state s_t. The idea is to decompose the Q_π function into a sum of two functions :

$$Q_\pi(s_t, a_t) = V_\pi(s_t) + A_\pi(s_t, a_t) \tag{B.9}$$

where $V_\pi(s_t)$ is the state-value function and $A_\pi(s_t, a_t) = q_\pi(s_t, a_t) - V_\pi(s_t)$ is the **advantage function**.
The state-value function $V_\pi(s_t)$ represents how it is good to be in a particular state s_t whereas the advantage function $A_\pi(s_t, a_t)$ gives a relative measure of the importance of each action.

The dueling architecture [28] is composed of two streams that represent the state value and the advantage function, while sharing a common feature learning module (first layers).
Both streams are combined via a special **aggregating layer** to produce the estimate $\widehat{q}$ of the action-value function Q_π.
Figure B.2 resumes this dueling architecture.

The reason behind this separation is that it is easier for the dueling architecture which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful for states where their actions do not affect the environment in a relevant way. In this case, it is unnecessary to calculate the value of each action thanks to the computation of the state-value function.

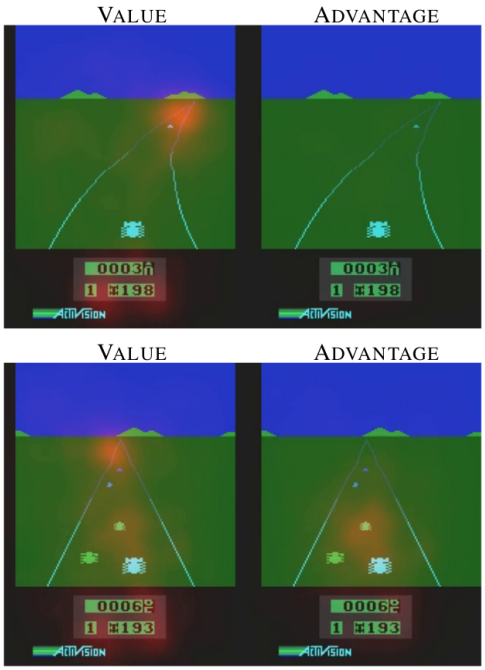


Figure B.3: See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions. (taken from [28])

Authors [28] give an example with the Atari game Enduro (Figure B.3). Indeed, moving right or left only matters if there is a risk of collision. And, in most states, the choice of the action has no effect on what happens (both images at the top of Figure B.3). In these states, the advantage function does not pay attention to the visual input. However, if there is a risk of collision (both images at the below of Figure B.3), advantage function pays attention to the danger to compute the best possible action.

Considering the dueling architecture introduced by Figure B.2, there are one stream of fully-connected layers $\widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta})$ estimating the state value function, and another stream output $\widehat{a}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha})$ evaluating the advantage value, where $\mathbf{w}$ denotes the parameters of the convolutional layers and $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ are the parameters of the two streams of fully-connected layers.
From the definition B.9, the aggregating layer may be defined as :

$$\widehat{q}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta}) + \widehat{a}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}) \tag{B.10}$$

Unfortunately, there is an **issue of identifiability**.
Indeed, given $\widehat{q}$ there are no way to refind $\widehat{v}$ and $\widehat{a}$. To improve understanding, adding a constant to $\widehat{v}$ and subtracting this same constant to $\widehat{a}$ implies the same value for $\widehat{q}$; thus there is an infinity pairs of $\widehat{v}$ and $\widehat{a}$ giving $\widehat{q}$.

Not being able to find $\widehat{v}$ and $\widehat{a}$ given $\widehat{q}$ is a major problem for the neural network's back propagation. A solution would be forcing advantage function estimator to have zero advantage at the chosen action :

$$\widehat{q}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta}) + \left(\widehat{a}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}) - \max_{a \in \Gamma(s_t)} \widehat{a}(s_t, a, \mathbf{w}, \boldsymbol{\alpha}) \right) \tag{B.11}$$

Consequently, for $a^* = \arg\max_{a\in\Gamma(s_t)} \widehat{q}(s_t, a, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \arg\max_{a\in\Gamma(s_t)} \widehat{a}(s_t, a, \mathbf{w}, \boldsymbol{\alpha})$, $\widehat{q}(s_t, a^*, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta})$.
With such aggregation, $\widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta})$ provides an estimate of the value function, while the other stream produces a correct estimate of the advantage function.

In practice another way is used to improve the stability:

$$\widehat{q}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \widehat{v}(s_t, \mathbf{w}, \boldsymbol{\beta}) + \left(\widehat{a}(s_t, a_t, \mathbf{w}, \boldsymbol{\alpha}) - \frac{1}{|\Gamma(s_t)|} \sum_{a\in\Gamma(s_t)} \widehat{a}(s_t, a, \mathbf{w}, \boldsymbol{\alpha}) \right) \tag{B.12}$$

Actually, it makes sense because $A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$ and $V_\pi(s_t) = \mathbb{E}_{a\sim\pi(s_t)}[Q_\pi(s_t, a)]$ so $\mathbb{E}_{a\sim\pi(s_t)}[A_\pi(s_t, a)] = 0$. However, the original semantics of $\widehat{v}$ and $\widehat{a}$ are lost because they are now off-target by a constant.

B.2 Introduction To Kronecker-factored Approximate Curvature

Let $\mathbf{W}_l \in \mathbb{R}^{C^{\text{out}}\times C^{\text{in}}}$ be the weight matrix in the l^{th} layer, where C^{out} and C^{in} are the number of output/input neurons of the layer.
We denote by $\mathbf{p}_l = \mathbf{W}_l \mathbf{o}_{l-1}$ the weighted sum for the l^{th} layer, and by $\mathbf{o}_l = \phi(\mathbf{p}_l) \in \mathbb{R}^{C^{\text{in}}}$ the vector of unit outputs for the l^{th} layer where ϕ is the activation function of the layer.
Let $\boldsymbol{\theta}$ to be the vector consisting of all of the network's parameters concatenated together, i.e. $[\text{vec}(\mathbf{W}_1)^\top, \text{vec}(\mathbf{W}_2)^\top \cdots \text{vec}(\mathbf{W}_l)^\top]^\top$, where vec is the operator which vectorizes matrices by stacking their columns together.

The Fisher Matrix $\mathbf{F}$ can be defined as [18]:

$$\mathbf{F} = \mathbb{E}\left[\frac{d\log(\pi(a_t|s_t, \boldsymbol{\theta}))}{d\boldsymbol{\theta}} \frac{d\log(\pi(a_t|s_t, \boldsymbol{\theta}))^\top}{d\boldsymbol{\theta}} \right] = \mathbb{E}\left[\mathcal{D}\boldsymbol{\theta} \mathcal{D}\boldsymbol{\theta}^\top \right]$$

where the operator $\mathcal{D}$ is defined as:

$$\mathcal{D}\mathbf{v} = \frac{d\log(\pi(a_t|s_t, \mathbf{v}))}{d\mathbf{v}}$$

Note that $\mathcal{D}\boldsymbol{\theta} = [\mathbf{d}_1^\top \mathbf{d}_2^\top \cdots \mathbf{d}_l^\top]$ where $\mathbf{d}_i = \text{vec}(\mathcal{D}\mathbf{W}_i)$ and so $\mathbf{F} = \mathbb{E}\left[\mathcal{D}\boldsymbol{\theta}\mathcal{D}\boldsymbol{\theta}^\top\right]$ can be viewed as an l by l block with the (i, j)-th block $F_{i,j}$ given by $F_{i,j} = \mathbb{E}[\mathbf{d}_i \mathbf{d}_j^\top]$.
Noting that $\mathcal{D}\mathbf{W}_i = \mathbf{g}_i \mathbf{o}_{i-1}^\top$ with $\mathbf{g}_i = \mathcal{D}\mathbf{p}_i$ and that $\text{vec}(\mathbf{u}\mathbf{v}^\top) = \mathbf{v} \otimes \mathbf{u}$ we have $\mathbf{d}_i = \text{vec}(\mathbf{g}_i \mathbf{o}_{i-1}^\top) = \mathbf{o}_{i-1} \otimes \mathbf{g}_i$, with $\otimes$ the **Kronecker product**. Thus we can rewrite the (i, j)-th block $F_{i,j}$ as :

$$\begin{aligned} F_{i,j} &= \mathbb{E}\left[\mathbf{d}_i \mathbf{d}_j^\top\right] \\ &= \mathbb{E}\left[(\mathbf{o}_{i-1} \otimes \mathbf{g}_i)(\mathbf{o}_{j-1} \otimes \mathbf{g}_j)^\top\right] \\ &= \mathbb{E}\left[(\mathbf{o}_{i-1} \otimes \mathbf{g}_i)(\mathbf{o}_{j-1}^\top \otimes \mathbf{g}_j^\top)\right] \\ &= \mathbb{E}\left[(\mathbf{o}_{i-1}\mathbf{o}_{j-1}^\top \otimes \mathbf{g}_i \mathbf{g}_j^\top)\right] \end{aligned}$$

The K-FAC approximation $\widehat{\mathbf{F}}$ of the Fisher information matrix $\mathbf{F}$ is derived by applying two statistical assumptions :

- Statistics of activation and pre-activation gradients are independent across layers, so that $\widehat{F}_{i,j} = 0$ for $i \neq j$ making $\widehat{\mathbf{F}}$ a block-diagonal matrix.
- Activation and pre-activation gradients are independent, then :

$$\begin{aligned}\widehat{F}_{k,k} &= \mathbb{E}\left[(\mathbf{o}_{k-1}\mathbf{o}_{k-1}^\top \otimes \mathbf{g}_k\mathbf{g}_k^\top)\right] \\ &= \mathbb{E}\left[\mathbf{o}_{k-1}\mathbf{o}_{k-1}^\top\right] \otimes \mathbb{E}\left[\mathbf{g}_k\mathbf{g}_k^\top\right] \\ &= O_{k-1} \otimes G_k\end{aligned}$$

  where $O_k = \mathbb{E}\left[\mathbf{o}_k\mathbf{o}_k^\top\right]$ and $G_k = \mathbb{E}\left[\mathbf{g}_k\mathbf{g}_k^\top\right]$

$\widehat{\mathbf{F}}$ is a block-diagonal matrix approximation of the Fisher matrix $\mathbf{F}$.
Knowing that for $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$:

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{B.13}$$

$$(\mathbf{A} \otimes \mathbf{B})\,\mathrm{vec}(\mathbf{C}) = \mathrm{vec}(\mathbf{A}\mathbf{C}\mathbf{B}^\top) \tag{B.14}$$

The natural gradient according to $\mathbf{W}_l$ is defined as:

$$\mathrm{vec}(\tilde{\nabla}_{\mathbf{W}_l} J) = \widehat{\mathbf{F}}_{l,l}^{-1}\,\mathrm{vec}(\nabla_{\mathbf{W}_l} J) = \mathrm{vec}(\mathbf{O}_l^{-1}\nabla_{\mathbf{W}_l} J \mathbf{G}_l^{-1}) \tag{B.15}$$

The update of the layers is done layer per layer. Each computation involves nodes in each layer only. Consider the number of nodes in each layer is in thousand or hundred ranges, rather than millions for the whole network. This is much manageable and K-FAC significantly reduces the computational complexity.

Appendix C

Appendix To Deep Reinforcement Learning Methods Applied To The Cart-Pole Environment

C.1 Hyperparameters For Fourier Basis Improvement

Hyperparameters	Values (with Fourier)	Values (without Fourier)
ϵ_{start} (exploration)	0.9	0.9
ϵ_{end} (exploration)	0.05	0.05
ϵ_{decay} (exploration)	0.077	0.077
α (PER)	0.6	0.6
β (PER)	0.4	0.4
β_{decay} (PER)	1/390000	1/390000
Learning rate η	5e-4	5e-4
Neural Architecture	[16]	[16, 16]
Batch size n	32	32
Target update	10	10
Discount factor γ	0.99	0.99
Replay Buffer Size	5000	5000

Table C.1: Duel-DQN Hyperparameters combined with and without order-1 Fourier Basis

Hyperparameters	Values (with Fourier)	Values (without Fourier)
Learning rate η	5e-4	7e-5
Value loss coefficient	0.85	0.5
Entropy coefficient	0.01	0.01
Neural Architecture	[256]	[256]
Batch size n	256	256
Number of epoch	25	15
ϵ_{clip}	0.2	0.2
Trace-decay (GAE)	0.96	0.96
Trace-decay (critic)	0.96	0.96
Advantage Normalization	True	True
Discount factor γ	0.99	0.99

Table C.2: PPO Hyperparameters combined with and without order-1 Fourier Basis

Hyperparameters	Values
Learning rate η	3e-4
Neural architecture for Actor	[256]
Neural architecture for Critic	[256]
Batch size n	64
Discount factor γ	0.99
Target update	1
Target update smooth	5e-3
Replay Buffer Size	1000000

Table C.3: SAC Hyperparameters combined with and without order-1 Fourier Basis

C.2 Hyperparameters For Comparison Of Deep RL Methods

Hyperparameters	Values
ϵ_{start} (exploration)	0.9
ϵ_{end} (exploration)	0.05
ϵ_{decay} (exploration)	0.077
α (PER)	0.6
β (PER)	0.4
β_{decay} (PER)	1/390000
Learning rate η	5e-4
Neural Architecture	[16]
Batch size n	32
Target update	10
Discount factor γ	0.99
Replay Buffer Size	10000

Table C.4: DQN and its extensions hyperparameters combined with order-1 Fourier Basis

Hyperparameters	Values (with Fourier)
Learning rate η	5e-4
Value loss coefficient	0.9
Entropy coefficient	0.01
Neural Architecture	[256]
Batch size n	1
Advantage Normalization	False
Discount factor γ	0.99

Table C.5: A2C Hyperparameters combined with order-1 Fourier Basis

Hyperparameters	Values (with Fourier)
Learning rate η	5e-4
Value loss coefficient	0.9
Entropy coefficient	0.01
Neural Architecture	[256]
Batch size n	256
Number of epoch	25
ϵ_{clip}	0.2
Trace-decay (GAE)	0.96
Trace-decay (critic)	0.96
Advantage Normalization	True
Discount factor γ	0.99

Table C.6: PPO Hyperparameters combined with order-1 Fourier Basis

Hyperparameters	Values (with Fourier)
Learning rate η	2.5e-1
Value loss coefficient	0.9
Entropy coefficient	0.01
Neural Architecture	[256]
Batch size n	25
Advantage Normalization	False
Discount factor γ	0.99

Table C.7: ACKTR Hyperparameters combined with order-1 Fourier Basis

Hyperparameters	Values
Learning rate η	3e-4
Neural architecture for Actor	[256]
Neural architecture for Critic	[256]
Batch size n	64
Discount factor γ	0.99
Target update	1
Target update smooth	5e-3
Replay Buffer Size	1000000

Table C.8: SAC Hyperparameters combined with order-1 Fourier Basis

Appendix D

Appendix To Measurements Of Safety Requirements

D.1 Signal Temporal Logic

D.1.1 STL Formalism

Let consider the set $\mathbb{B} := \{\bot, \top\}$ of **boolean values** (with $\bot < \top, \top = \bot$ and $\bot = \top$) and $\bar{\mathbb{R}} = \mathbb{R} \cup \mathbb{B}$ the totally ordered set of real numbers with as smallest element $\bot$ and greatest element $\top$.

A **signal** is a function $D \to E$, with D an interval of $\mathbb{R}^+$ and $E \subseteq \bar{\mathbb{R}}$. Signals with $E = \mathbb{R}$ are called **real-valued signals**.

An **execution trace** [23] w is a set of real-valued signals $\{x_1^w, x_2^w, \cdots, x_k^w\}$ defined over some interval D of $\mathbb{R}$ where D is called the **time domain** of w.
Such a trace can be "booleanized" through a set of threshold predicates of the form $x_i \geq 0$. Signal Temporal Logic is then a simple extension of Metric Temporal Logic where real-valued variables $(x_i)_{i\in\mathbb{N}}$ are transformed into Boolean values via these predicates.

Let w be a trace of time domain D. The **STL formula** [23] φ is said to be defined over a time interval $\mathrm{dom}(\varphi, w)$ given by the following rules: $\mathrm{dom}(true, w) = \mathrm{dom}(x_i \geq 0, w) = D$, $\mathrm{dom}(\neg\varphi, w) = \mathrm{dom}(\varphi, w), \mathrm{dom}(\varphi \wedge \psi, w) = \mathrm{dom}(\varphi, w) \cap \mathrm{dom}(\psi, w), \mathrm{dom}(\varphi \mathbf{U_I} \psi, w) = \{t \in \mathbb{R} | t + [0, inf(I)] \subseteq \mathrm{dom}(\varphi, w)$ and $t + inf(I) \in \mathrm{dom}(\psi, w)\}$ where I is a closed, non-singular interval of $\mathbb{R}^+$ (includes bounded intervals $[a, b]$ and unbounded intervals $[a, +\infty)$ for any $0 \leq a < b$).

D.1.2 Boolean Semantics

For a trace w, the **validity** [23] of an STL formula φ at a given time $t \in \text{dom}(\varphi, w)$ is set according to the following inductive definition.

$w, t \models true$
$w, t \models x_i \geq 0$ iff $x_i^w(t) \geq 0$
$w, t \models \neg\varphi$ iff $w, t \not\models \varphi$
$w, t \models \varphi \wedge \psi$ iff $w, t \models \varphi$ and $w, t \models \psi$
$w, t \models \varphi \mathbf{U_I} \psi$ iff exists $t' \in t + I$ s.t. $w, t' \models \psi$ and for all $t'' \in [t, t'], w, t'' \models \varphi$ (*until* operator)

We can redefine other usual operators as syntactic abbreviations:

$$\begin{aligned} false &:= \neg true \\ \Diamond_I \varphi &:= true \mathbf{U_I} \varphi \ (\textit{eventually operator}) \\ \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi) \\ \Box_I \varphi &:= \neg\Diamond_I \neg\varphi \ (\textit{always operator}) \end{aligned}$$

The *always operator* $\Box_{[a,b]}\varphi$ means that $\forall t' \in [t + a, t + b], (w, t') \models \varphi$. In other words, the property φ is respected on the time interval $[t + a, t + b]$.

D.1.3 Quantitative Semantics

Given a formula φ, trace w, and time $t \in \text{dom}(\varphi, w)$, we can also define the **quantitative semantics** $\rho(\varphi, w, t)$ [23] by induction as follows:

$$\begin{aligned} \rho(true, w, t) &= \top \\ \rho(x_i \geq 0, w, t) &= x_i^w(t) \\ \rho(\neg\varphi, w, t) &= -\rho(\varphi, w, t) \\ \rho(\varphi \wedge \psi, w, t) &= \min\{\rho(\varphi, w, t), \rho(\psi, w, t)\} \\ \rho(\varphi \mathbf{U_I} \psi, w, t) &= \sup_{t' \in t+I} \{\min\{\rho(\psi, w, t'), \inf_{t'' \in [t,t']} \rho(\varphi, w, t'')\}\} \end{aligned}$$

Therefore, by definition of the *eventually operator* and the *always operator*, we can also define ρ as :

$$\begin{aligned} \rho(\Diamond_I \varphi, w, t) &= \sup_{t' \in t+I} \rho(\varphi, w, t') \\ \rho(\Box_I \varphi, w, t) &= \inf_{t' \in t+I} \rho(\varphi, w, t') \end{aligned}$$

The quantitative semantics of STL have many interesting properties. $\rho(\varphi, w, t)$ quantifies the degree of satisfiability.
A large positive value indicates that the formula φ is robustly satisfied by the trace w at time t, a positive value close to zero suggests that w satisfies φ but it is close to violating φ, and a negative value indicates that the formula φ is violated by w.

D.2 Empirical Distribution Function

The **empirical distribution function** is a well-known non-parametric estimator $\hat{F}_n$ of the cumulative distribution F of a random variable X with independent, identically distributed (i.i.d) sample $(X_i)_{i=1,\dots,n}$ from an unknown distribution function $F(x) = P(X \leq x)$ [25]. The empirical cumulative distribution function is defined as:

$$\hat{F}_n(x) = \frac{1}{n} \sum_{i=1}^{n} 1\{X_i \leq x\} \ (x \in \mathbb{R})$$

Simply said, $F_n(x)$ is just the fraction of sample that falls bellow x. With the Law of Large Numbers theorem we know that if X has a cumulative distribution F, then $F_n(x) \to F(x)$ with probability one for each x as $n \to \infty$.

D.3 CDF-based Nonparametric Confidence Interval

D.3.1 Confidence Interval And Binomial Distribution

It could be interesting to know the **confidence interval** around each estimate $F(x)$. That is to say, according to a nominal confidence level c, we compute for each value x an interval $[p_l, p_u]$ for which we are sure with a probability c that that the true cumulative distribution function $F(x) \in [p_l, p_u]$.

For each x, the random variable $1\{X_i \leq x\}$ is a Bernoulli random variable with probability $p = F(x)$. Therefore, as $n\hat{F}_n(x)$ is a sum of n Bernoulli random variables, it is has a binomial distribution with parameters n and success probability $F(x)$. Using $n\hat{F}_n(x)$ as a binomial distribution is useful because many techniques have been developed to estimate the confidence intervals for binomial population proportions, in particular with the Beta Distribution Generator, the Clopper & Pearson Approach, the Normal Approximation or the Wald method [25, 24].

The **Beta Distribution Generator** method is presented in the following subsection and was proposed by Ewan Cameron [24]. Indeed, this methods seems the more efficient and robust according to the number of samples n for our problem.

D.3.2 The Beta Distribution Generator Method

To have the same notation that the Cameron's paper [24] for a specific $x \in \mathbb{R}$, we note $k = \sum_{i=1}^{n} 1\{X_i \leq x\} \in [1, n]$, $\hat{p} = \hat{F}_n(x) = \frac{k}{n}$ and $p = F(x)$.

We know that the likelihood of observing the result, $\hat{p}$, for a given value of p is proportional to $p^k(1-p)^{n-k}$. The normalisation of this likelihood function over $0 < p < 1$ defines a **beta distribution** with integer parameters, $a = k + 1$ and $b = n - k + 1$:

$$B(a, b) = \frac{(a+b-1)!}{(a-1)!(b-1)!} p^{a-1}(1-p)^{b-1} \tag{D.1}$$

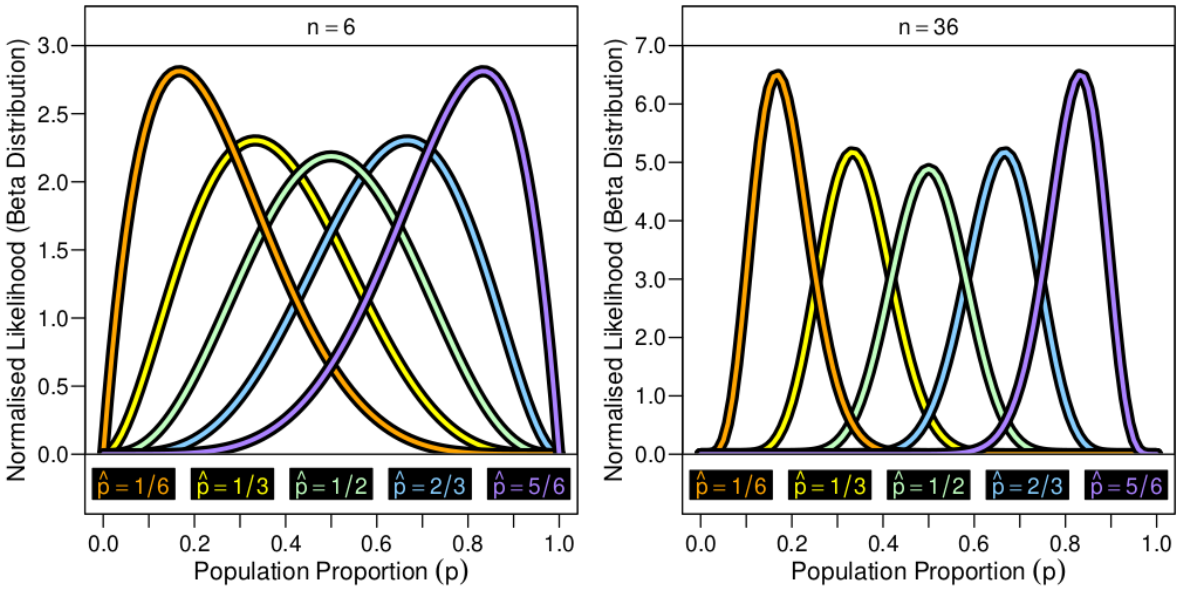


Figure D.1: Example likelihood functions for the true value of the underlying population proportion, p, given five 'measured' success fractions, $\hat{p} = \frac{k}{n}$, for samples of sizes $n = 6$ (left panel) and $n = 36$ (right panel). In each case the shape of the curve is given by the beta distribution with shape parameters as specified by Equation (D.1). The asymmetric nature of this likelihood function in the small sample size regime is clearly evident amongst the $n = 6$ examples, as is its convergence in the intermediate-to-large sample size regime towards a narrower, more symmetric, (pseudo-)normal distribution amongst the $n = 36$ examples. (taken from [24])

This likelihood function reveals that $\hat{p}$ is the maximum likelihood estimator of p. The characteristic shape of the beta distribution likelihood function for p is illustrated in Figure D.1. Given no *a priori* knowledge we may suppose that all values of p are equally probable. Formally, this condition is characterised via the Bayes-Laplace uniform prior, for which $P_{\text{Prior}}(p) = 1, \forall p \in [0, 1]$. Application of the Bayes' theorem under this assumption allows us to treat then the normalised likelihood function for p as a posterior probability distribution. Thus, the quantiles of the beta distribution from Equation (D.1) may be used directly to estimate confidence intervals on the underlying population proportion given the observed data. Specifically, the lower and upper bounds, p_l and p_u, defining an **equal-tailed** (or **central**) interval for p at a nominal confidence level of $c = 1 - \alpha$ are given by the quantiles:

$$\int_0^{p_l} B(a, b)dp = \frac{\alpha}{2} \text{ and } \int_{p_u}^1 B(a, b)dp = \frac{\alpha}{2} \tag{D.2}$$

It is interesting to note that the bounds of this equal-tailed interval is asymmetric about the maximum likelihood value $\hat{p}$ due to the asymmetric nature of the beta distribution likelihood function for p.

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[2] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[5] Sarah Osentoski George Konidaris and Philip Thomas. Value function approximation in reinforcement learning using the fourier basis. 2011.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[7] John Stachurski. *Economic Dynamics: Theory and Computation*, volume 1 of *MIT Press Books*. The MIT Press, December 2009.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[9] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[11] Ronald Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3:241–, 09 1991.

[12] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *IN PROC. 19TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING*, pages 267–274, 2002.

[13] Katerina Fragkiadaki. Deep reinforcement learning and control : Natural policy gradients, trpo, ppo. https://www.win.tue.nl/~rmcastro/AppStat2013/files/lecture1.pdf.

[14] Joshua Achiam. Advanced policy gradient methods, 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf.

[15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[16] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[17] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation, 2017.

[18] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature, 2015.

[19] Savinay Nagendra, Nikhil Podila, Rashmi Ugarakhod, and Koshy George. Comparison of reinforcement learning algorithms applied to the cart pole problem, 2018.

[20] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2017.

[21] Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards, 2018.

[22] E. Schuitema, Lucian Busoniu, Robert Babuska, and Pieter Jonker. Control delay in reinforcement learning for real-time dynamic systems: A memoryless approach. pages 3226 – 3231, 11 2010.

[23] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for stl. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 264–279, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[24] Ewan Cameron. On the estimation of confidence intervals for binomial population proportions in astronomy: The simplicity and superiority of the bayesian approach, 2010.

[25] Rui Castro. Lecture 1 - introduction and the empirical cdf. https://www.win.tue.nl/~rmcastro/AppStat2013/files/lecture1.pdf.

[26] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[27] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[28] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.